DEVice **I**nterface **L**anguage (**DEVIL**)

Boklyn Wong (bw2007@cs.columbia.edu)
Pranay Wilson Tigga/Team Leader (pt2116@cs.columbia.edu)
Vishal Kumar Singh (vs2140@columbia.edu)

Table of Contents

Chapter 1: Introduction

1.1 Introduction

DEVIL is a device configuration language that would generate a script or an executable which when run can perform the actual configuration on the device as specified by DEVIL. Devil is an Easy to Use, Platform independent, Reusable and Extensible.

1.2 Background

Different devices/servers from different vendors are configured in different ways which are generally specific to each individual device. Network administrators need to know every specific detail about the device before they can configure it properly. A network admin capable of configuring a Cisco router might not be able to properly configure a NEC router and need to learn different configuration interfaces for configuring devices from different vendors.

This increases the possibility of error and increases the time it takes for deployment and configuration of network. There is currently no network application that would be able to configure devices across vendors in a generic way. Moreover, different devices might follow different configuration protocols, which make it difficult to develop a common utility to work across different platforms. For example, a device running Linux operating system would be configured differently than devices running windows.

1.3 Motivation

The DEVIL has far-reaching implications and applications in today's world of diverse platforms and devices. One such application would be in the field of Network Management for device configuration. Future generations of Network Administrators would not longer have to be familiar intuitively with all aspects of devices in his/her network. With the creation of DEVIL, A Linux network admin would be able to manage as well as maintain a Window’s based network. Further applications of DEVIL could be in the field of personal computing. DEVIL could possibly grow into a scheduling language.

1.4 Related Work

There has been much attempt to standardize the configuration mechanism which resulted in different interfaces and protocols for configuration e.g. SNMP, SOAP and Web services are being used for device configuration. This has not helped to alleviate the problem but wound up aggravating it, by creating a need to learn more interfaces and support them. Other work has been done by Netconf Inetrnet Engineering Task force (IETF) working group which proposes a request response based mechanism to configure devices.

Our group found out that the main issue of different ways of configuration can be solved by separating the user from device specifics and providing a vendor specific compiler which generates the output code whose target is the vendor's device. The compiler is provided by vendor and he understands the internals of device. This disassociates user from multiplicity of device configuration techniques and provides him with a single and easy to use interface in the form of our language.

1.5 Goals

High Level: Users who uses DEVIL would never have to know or understand the lines of code that went into producing the output that performs his/her desired task.

Syntactically Intuitive: One of the Goals of the DEVIL language is that users who use DEVIL would not need any prior knowledge of the device to be configured but simply an understanding of what he would like the device to perform. In other words DEVIL must be Easy to Use.

Portable: Since DEVIL is text based and written in XML, it is capable of generating the target script for any device. This fact makes DEVIL a highly portable language. DEVIL is analogous to Java in the sense that compilers for either language generate platform specific code.

1.6 Features

High Level**:** Users who uses DEVIL would never have to know or understand the lines of code that went into producing the output that performs his/her desired task. Syntactically Intuitive: One of the Goals of the DEVIL language is that users who use DEVIL would not need any prior knowledge of the device to be configured but simply an understanding of what he would like the device to perform. In other words DEVIL must be Easy to Use.

Portable: Since DEVIL is text based and is written in a very simple syntax, it is capable of generating the target script for any device. This fact makes DEVIL a highly portable language. DEVIL is analogous to Java in the sense that compilers for either language generate platform specific code.

Context Less and Context Aware: There are certain kinds of configurations which are state-full and others are stateless. A state full configuration has a notion of context whereas a stateless configuration can be context less. An example of a context less configuration command is setting IP address and hostname in Linux where there is no dependency on the order of commands being executed, whereas a context aware case of configuration is a router configuration where global commands override local commands e.g. Blocking ICMP packet can be global and Allowing can be local to interface so the context of command being invoked is important.

Easy to use: Devil is very simple in terms of syntax. A simple code can be generated with no hassles of data type handling or memory allocation implementations. In an almost English like language one can program a network device without having to know
the type of router or its intrinsic network details.

Devil follows a C type syntax for control flow and function calls. Without having to understand what IP Addresses or ports is one can program various configurations for network devices from DEVIL.

1.7 Sample Codes

Intro: There are two types of sample code provided below. The first would be an
example on how our language can be used to configuring a network device. The second
would illustrate the full functionality and flexibility of our language.

Sample Code 1:
In this example we will use our language to configure the firewall on a Linux Fedora Core 3 machine to block a specific IP.

A command of iptables –L to display current rules would yield:

```
[root@dhcp14 bw2007]# iptables –L
Chain INPUT (policy ACCEPT)
target prot opt source destination
RH-Firewall-1-INPUT all --anywhere anywhere
Chain FORWARD (policy ACCEPT)
target prot opt source destination
```

```
RH-Firewall-1-INPUT all --anywhere anywhere
Chain OUTPUT (policy ACCEPT)
target prot opt source destination
Chain RH-Firewall-1-INPUT (2 references)
target prot opt source destination
ACCEPT all --anywhere anywhere
ACCEPT icmp --anywhere anywhere icmp any ACCEPT ipv6-crypt--anywhere
anywhere
ACCEPT ipv6-auth--anywhere anywhere
ACCEPT udp --anywhere 224.0.0.251 udp dpt:5353
ACCEPT udp --anywhere anywhere udp dpt:ipp
ACCEPT all --anywhere anywhere state RELATED,ESTABLISHED
ACCEPT tcp --anywhere anywhere state NEW tcp dpt:http
ACCEPT tcp --anywhere anywhere state NEW tcp dpt:https
ACCEPT tcp --anywhere anywhere state NEW tcp dpt:ssh
ACCEPT tcp --anywhere anywhere state NEW tcp dpt:smtp
ACCEPT tcp --anywhere anywhere state NEW tcp dpt:netbios-ssn
ACCEPT tcp --anywhere anywhere state NEW tcp dpt:microsoft-ds
ACCEPT tcp --anywhere anywhere state NEW tcp dpt:5901
ACCEPT tcp --anywhere anywhere state NEW tcp dpt:5902
ACCEPT tcp --anywhere anywhere state NEW tcp dpt:glftpd
REJECT all --anywhere anywhere reject-with icmp-host-prohibited
```

INPUT: block.devil

```
block                                                          218.12.13.4
```

INTERPRETATION

The above input program is parsed and interpreted to generate the following output. The input program sequences are interpreted and mapped to generate output shell script corresponding to the action defined.

e.g. block does blocking of IP Address and
Port.

The input sequence "block" is recognized and interpreter callas appropriate functions to generate the code.

OUTPUT
block.sh:

```
iptables -I INPUT -s 218.12.13.4 -j DROP
```

Sample Code 2:

Chapter 2: Lexical Conventions

2.1 Comments

The character # introduces a comment. Everything on that line will be considered as a comment

2.2 Identifiers

Identifier is a sequence of letter and digit with first character must be letter.

2.3 Keywords

The following identifiers are reserved for the language use and may not be used otherwise:

Break
else
if
while

2.4 Separators

( ) [ ] { } ;

Separators which are Ignored
Newline \n
Tab \t
Carriage \r
White Space

2.5 Types and Variables

Strong type controlled language. Mismatched types will not be automatically resolved or allowed by the compiler.

1. Data Types

   No Types.
   The compiler decides variable types only at back end on run time.
   There are no arrays but , there are K-ary Tuples, these data structures can support different data types in the array.
2. Variables

   The variables are of type DevilDataType.

The scope can be

I. Global.

II. Local.

3. Initialization

Initialization , causes the declaration and determines the scope.

An un-initialized variable will not be considered by compiler.

4. Conversions

Since, all data types are implicit and supported at back end, the type conversions are done appropriately internally to handle it, assignments and reassignments.

2.6 Operators, Declarations, Expressions, Statements and Blocks

2.6.1 Operators

The following is the list of Operators in our language.

```
< > <= >= == NOT
+ -* / %
++ --AND OR
=
```

1. 1. Operator + : Addition
2. 2. Operator - : Subtraction
3. 3. Operator * : Multiplication
4. 4. Operator / : Division
5. 5. Operator % : remainder of division
6. 6. AND : Logical AND operation
7. 7. OR : Logical OR
8. 8. NOT : Not operation
9. 9. ++ : Increment
10. 10. --: Decrement
11. 11. = : Assignment
12. 12. < : Less then
13. 13. > : Greater then
14. 14. <= : Less then equal to.
15. 15. >= : Greater then equal to.
16. 16. == : equal to

Postfix increment: (x ++)
Prefix increment: (++x)
Postfix decrement: (x--)
Prefix decrement: (--x) Relational Operators
> Greater then
< Lesser then
>= greater then equal to

<= lesser then equal to

*expression * expression*
expression + expression
expression -expression
expression / expression
expression % expression

expression++
expression--
++expression
--expression

expression < expression
expression > expression
expression <= expression
expression >= expression
expression == expression

Identifier =expression

Operator Precedence:

LOGIAL.
*, /, % ,
++ , --, + , -
< , > , <= , >= , ==
=

2.6.2 Declarations

1. Array Creation
   A[1] = data;
   Creates memory block and stores index., does not allocate N blocks before hand.

2. Array Indexing
   Read
   Identifier[expr];

Store

Identifier[expr] = expr;

2.6.3 Statements

Program Consists of Statements and Function definitions.
Statements can also be function call..

1. Expression Statements

    Most statements are expressions.
    Statements are executed in sequence.
    Successful evaluation of expression completes the statements.

2. Conditional Statement

    There are 2 conditional statements

1. *1. if (condition) { statement }*
2. *2. if (condition) {statement1 } else {statement2}*

The *condition* in above 2 statements is an evaluation of *expression.*
The statement in 1 is executed if condition is “true”.
The statement1 in 2 is executed if condition is “true” and statement2 is executed if condition is “false”.

The *else* is connected to innermost *if.*

3. Loop Statement

➢ while statement
while (condition) statement
Condition is evaluation of an *expression*. When the expression becomes “false” the loop exits.

2.7 Functions

The form of function definition is described below.

function_definition:
function_declaration function_body

function_declaration:
function_name (parameter_list)

function_body:
function_statements function_statements:
statement_list

statement_list:
  statement statement_list

statement:
  expression

2.8 Future work: Ontology

The team is working to devise a way to represent the knowledge which a device has and other devices can reuse using language constructs which are intuitive to network admins. This may include representing things like relationship between different kind of devices and there interactions. E.g. a Linux based router can inherit knowledge from a generic router. But the same router can have ports and can also act as firewall device. The relationship between Linux Router and Firewall is of Association, relationship between port and Linux router is composition and relationship between Linux Router and Generic Router is of Inheritance. Establishing these relationships will enable information sharing
between devices.

2.8.1 Sample Code

Prg1.dv

```
# This program sets the hostname of a linux machine

Object LinuxMachine {

static string platform = “Linux”;
volatile string hostname;

}

boolean set_hostname(string hostname)
{
        string command = “sethostname”;
        genshell( command , hostname);
        return true;
}

Main( )
{
LinuxMachine lm ;
                                set_hostname (“PLT_LAB”);

}
```

O/p is a file with following entry :
sethostname PLT_LAB

Prg2.dv : Enable firewall on Linux

```
Object LinuxMachine {

static string platform = “Linux”;
volatile string hostname;

}
Object LinuxFirewall;
LinuxFirewall -> LinuxMachine;

LinuxFirewall {
    boolean enabled = false;
}



boolean set_hostname(string hostname)
{
                            string command = “sethostname”;
                             genshell( command , hostname);
        return true;
}
boolean enable_firewall( )
{
    string command = “ipchain enable firewall”;
    genshell(command);
}

Main( )
{

        LinuxFirewall lm;
                               set_hostname (“PLT_LAB”);
         enable_firewall ( );
} O/P
```

Sethostname PLT_LAB

Ipchain enable firewall

3 Project Plan: -
Here we describe our design, plan and implementation for DEVIL.

3.1 Plan and Timeline

TIMELINE:
White Paper 9/28
LRM 10/21
Complete Lexer and Parser 10/21
Complete Testing 12/21
Complete Changes 12/23
Complete Documentation 12/23
Submission 12/23

RESPONSIBILITIES
BokLyn Wong : Front End, Documentation and Backend modifications
Pranay Tigga: Front End, Backend and Documentation.
Vishal Kumar Singh: Front End, Backend and Backend modifications

3.2ENVIRONMENT
DEVIL language was developed on top of Suns Java 4 using ATLR 2.7.4. ANTLR was used to write the Lexer, Parser and Tree walker for DEVIL. The output components were generated in Shell script to configure Linux 2.8.6.1 IP table.

Testing was done on a Fedora CORE 3 Linux Box.

3.3Coding Style
The following is a list of coding standards, conventions, and guidelines used during the development on the PSP language.

Naming conventions
Variable Names: Variables are C style variables which can be represented as a string of alphabets and digits starting with an alphabet. White spaces are prohibited in order
to
avoid confusion for the Lexer.
Example:
X =10;
Ip1= "129.12.235.3";
As DEVIL is an un-typed language, variables are declared with assignment only.

.
Function Names & Parameters Names:
The same conventions are same as variable names. See the section on variable names for details.

Commenting
Comments are extremely important as they should provide readers with useful information about your source code. Ideally should be above the line of code that the comment pertains to.

CHAPTER 4

4.1 Architecture:

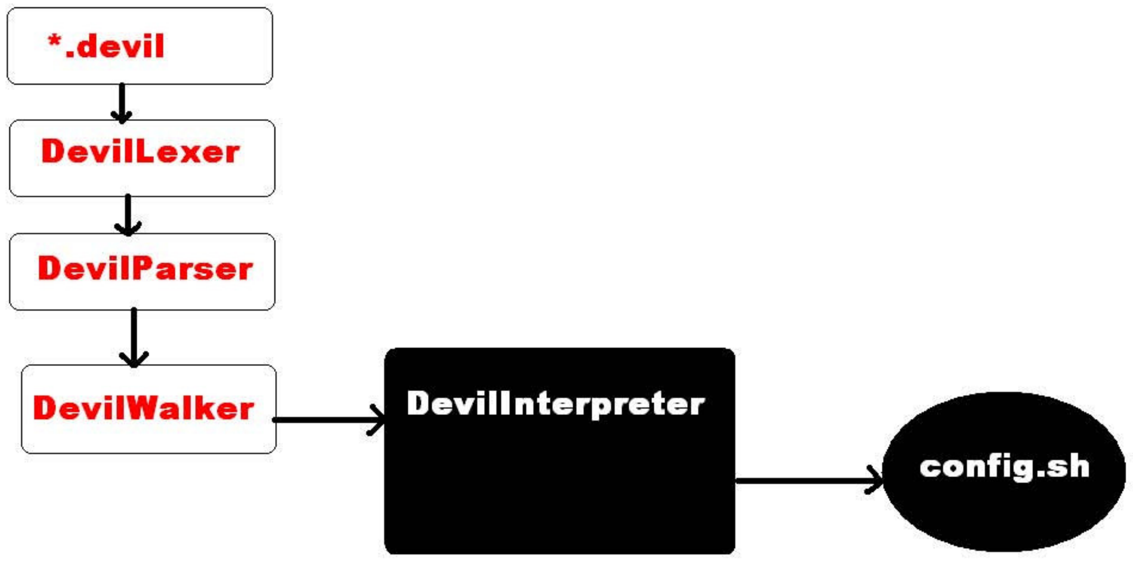


fig 4.1

***.devil:**

The input file has an extension of .devil. Extension check is made before compilations.

DevilLexer:

The Lexer takes the input stream from the .devil file and tokenizes it to give an output to the parser.
Some of the resultant tokens are:

```
INCREMENT: "++";
DECREMENT: "--";
PLUS: '+';
DASH: '-';
LOGICALNOT: '!';
STAR: '*';
FORWARDSLASH: '/';
MODULUS: '%';
GT: '>';
GE: ">=";
LS: '<';
LE: "<=";
NOTE: "!=";
E: "==";
AND: "&&";
OR: "||";
QMARK: '?';
COLON: ':';
ASSIGN: '=';
ASSIGNPLUS: "+=";
ASSIGNMINUS: "-=";
ASSIGNMULT: "*=";
ASSIGNDIV:                                                  "/=";

SEMICOLON:                                                  ';';
```

DevilParser:

This is the Parser which performs lexical analysis on the input stream provided by the tokens provided by the Lexer. The major components of the Parser are the tokens provided to the Walker.
Tokens are as follows:

```
tokens{
UPL;
UMN;
ULNOT;
LINC;
LDEC;
RINC;
RDEC;
STATEMENT;
CONCAT;
TEXT;
LABEL;
FOR_COND;
INDEX;
RINC_STATEMENT;
LINC_STATEMENT;
RDEC_STATEMENT;
LDEC_STATEMENT;

FOR_CONDITION; FOR_EXP;
ARRAY;
ARGLISTS;
FUNC_CALL;
FUNC_DEF;
}
```

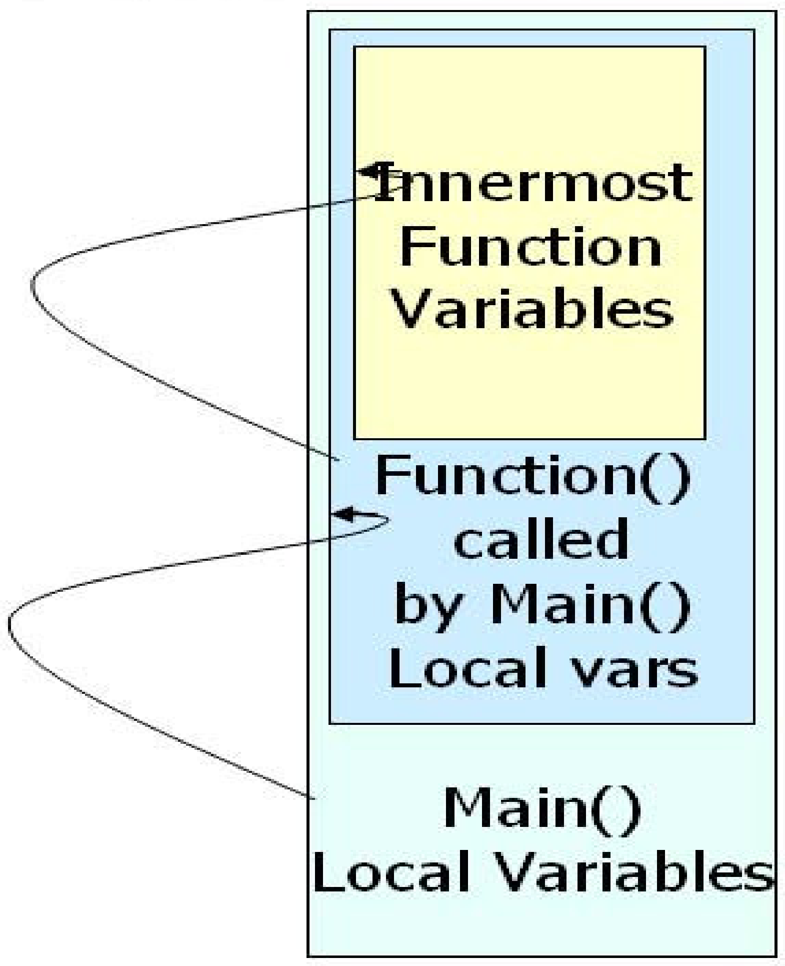
All the rules start from “program”. The majority of statements are analyzed by the “statement” rule.
Some of the rules are given below:

```
program
    :
                    (function_def | statement )* EOF! {#program = #([STATEMENT,
"PROGRAM"],program);}
    ;


statement
    :
        label_statement
       | break_statement SEMICOLON!
        | cont_statement SEMICOLON!
       | do_statement SEMICOLON!
        | if_statement
       | for_statement
        | io_statement SEMICOLON!
        | while_statement
       | assigment_statement SEMICOLON!
        | inc_dec_statement SEMICOLON!
        | statement_block
        | block_func SEMICOLON!
        | unblock_func SEMICOLON!
        | allow_func SEMICOLON!
        | function_stmt
    ;
```

DevilWalker:
This is one of the most crucial components of the DEVIL language. The Walker takes the AST from the Parser and defines rules of how to handle these tokens.
Some of the actions are given below:

```
expr returns [ DevilDataType x ]
{
    DevilDataType                                                                     a,b,c;

    x = null_data;

        String out = "";
    String[] vars;
    Vector params;
    DevilDataType[] stack;
}

:#(OR a=expr b=expr) { x=interpreter.or(a,b);

        //System.out.println("<WALKER> OR " );

                                                }
      | #(AND a=expr b=expr)            {
                                                       x=interpreter.and(a,b);
                                            }
          | #(ULNOT a=expr)                        { x = interpreter.logicalnot(a);

                  }
       | #(GT a=expr b=expr)                       { x = interpreter.gt(a, b);
                                                   //

        System.out.println("<WALKER> GT" );
                                                              }
        | #(GE a=expr b=expr) { x = interpreter.ge(a, b); }
        | #(LS a=expr b=expr) { x = interpreter.ls(a, b); }
        | #(LE a=expr b=expr) { x = interpreter.le(a, b); }
                    | #(NOTE a=expr b=expr) { x = interpreter.note(a, b); }
        | #(CONCAT a=expr b=expr) {x = interpreter.concat(a,b);}
        | #(E a=expr b=expr) {
                                                              x = interpreter.equals(a, b);

                }
        | #(PLUS a=expr b=expr) {
                                                       x = interpreter.plus(a, b);

                                                       }
                   | #(DASH a=expr b=expr) { x = interpreter.dash(a, b); }
                    | #(STAR a=expr b=expr) { x = interpreter.star(a, b); }
        | #(FORWARDSLASH a=expr b=expr) { x = interpreter.div(a, b); }
        | #(MODULUS a=expr b=expr) { x = interpreter.modulus(a, b); }
        | #(ASSIGN a=expr b=expr) {
                                                       x = interpreter.assign(a,b);
                                               }
             | #(ASSIGNPLUS a=expr b=expr) { x = interpreter.assignplus(a, b); }
           | #(ASSIGNMINUS a=expr b=expr) { x = interpreter.assignminus(a, b); }
             | #(ASSIGNMULT a=expr b=expr) { x = interpreter.assignmult(a, b); }
        | #(ASSIGNDIV a=expr b=expr) { x = interpreter.assigndiv(a, b); }
        | #(RINC a=expr) { x = interpreter.rincrement(a); }
        | #(LINC a=expr ) {x = interpreter.lincrement(a); }
        | #(RDEC a=expr ) { x = interpreter.rdecrement(a); }
        | #(LDEC a=expr ) { x = interpreter.ldecrement(a); }
        | #(INCREMENT a=expr) { x = interpreter.inc_stat(a); }
                        | #(DECREMENT a=expr) { x = interpreter.dec_stat(a); }
        | #(NOTHING a=expr b=expr) { x = interpreter.concat(a,b);}

        | #(ARRAY a=expr b=expr) { //allocate array[index] = expr
```

DevilInterpreter:
This is the most crucial component of the whole architecture of the DEVIL language.
The Interpreter has all the actions defined in the Walker. It performs Symbol Table and Activation Stack loading and offloading tasks. These tasks are the very foundation of implementing Data Types and Functions with scoping.
Following is a code piece which shows how variables are loaded into the Symbol Table and type definitions take place in the Symbol Table. The plus action performs type check and Symbol Table lookup/ modification to perform the addition task on the variable/ expressions. The Interpreter loads the variables into the Symbol Table and functions into


Innermost Function Variables
Function() called by Main() Local vars
Main() Local Variables

The Implementations of plus action is demonstrated below with a code snippet:
.

```
  public DevilDataType plus(DevilDataType a, DevilDataType b){
          String returnString = new String();
          DevilString init_str = new DevilString("");

          DevilDataType                          dta                    =                    null;

          DevilDataType dtb = null; DevilString xs = null;
          DevilInteger xi = null;
          DevilDouble xd = null;
          String s1 = new String(), s2 = new String();
          double d1 = 0, d2 = 0, returnDouble;
          int i1 = 0, i2 = 0, returnInt;
          boolean var1 = false, var2 = false;

          try                                                                                    {


              DevilDataType valueA =null;
              DevilDataType valueB =null;
              String varNameA = null;
              String typeA = null;
              String varNameB = null;
              String typeB = null;

              valueA = getDevilData(a);
              valueB = getDevilData(b);



               if (valueA.typename() == null && valueB.typename() == null) {
          //System.out.println("<PLUS> returning ");

                          return a;

                      }



               if (valueA.typename() == valueB.typename()){
                      if(valueA.typename() == "Integer"){
                              returnInt = ((DevilInteger)valueA).getValue()+
((DevilInteger)valueB).getValue();
                                      xi = new DevilInteger(returnInt);
                              xi.setType("Integer");
                          return xi;
                      }
                      else {
                                      if(valueA.typename() == "Double"){
                                        returnDouble = ((DevilDouble)valueA).getValue()+
((DevilDouble)valueB).getValue();
                                        xd = new DevilDouble(returnDouble);
```

```
                        xd.setType("Double");
                        return xd;
                    }
            else{
                        if(valueA.typename() == "String"){
                            return concat(a,b);
                        }
                    }
                }
            }
           else if(valueA.typename()!=valueB.typename())
           {
                if ((valueA.typename()=="String") || valueB.typename()!="String"){
                    return concat(a,b);
}



              if ((valueA.typename()!="String")&&(valueB.typename()!="String"))
                {
                    if(valueA.typename()=="Integer")
                    {
                        d1 =((DevilInteger)valueA).getValue();
                    } else {
                        d1 =((DevilDouble)valueA).getValue();
                    }
                    if(valueB.typename()=="Integer")
                    {
                        d2 =((DevilInteger)valueB).getValue();
                    } else {
                        d2 =((DevilDouble)valueB).getValue();
                    }

                    double retval = d1 + d2;
                    xd = new
                    DevilDouble(retval);
                    xd.setType("Double")
                    ;
                    return xd;

                }

           }                         else                         {


                 System.out.println("Compiler error Mismathced data types\n");
                }

        }
        catch(Exception ex)
           {
               System.out.println("\nException occurred in Interpreter PLUS"+ex);
                ex.printStackTrace();
           }
        return null;
    }
```

config.sh:
The output code is generated in shell script. This shell script has all the required assignments of and the Symbol Table assignments as well as the Function Stack pointers to perform the required tasks. The script performs actual device configuration.

CHAPTER 5

Lessons Learnt:

BokLyn Wong:-
Team coordination is the key here. Division of work can actually be a deciding factor on how much work actually gets done. Most importantly trust building has to there so that no one shrieks from their responsibly or leaves in the middle of the course.

Pranay Tigga:-
During this project, careful planning was crucial and we lacked in that field. We started off very ambitiously and had to make a lot of changes at the end. Crucial thing is to identify the difficult parts first and finish them earliest. Rest of the structure can be built
upon it. All teammates proved to be very accommodating when they were required to, but we had our share of frustrations.

Vishal Kumar Singh:-
All the functionalities of the output should always be kept in mind. I learnt that ignoring output implementation can cost dearly and modifications at the end are really painful.
The whole work should be modular but each team member should know the exactly does the other member has done. Teammates should be chosen judicially so that no one backs off at the time of real work, which we had to regretfully experience, leaving us short of
manpower.

CODE LISTING:

Grammar:

Devil.g

```
/* grammmar at 12:51 PM
boklyn */
{
import java.util.*;
}

class DevilParser extends Parser;
options {
     k=2;
     buildAST = true;
     exportVocab = Devil;
     defaultErrorHandler = false;
}

tokens{
UPL;
UMN;
ULNOT;
LINC;
LDEC;
RINC;
RDEC;
STATEMENT;
CONCAT;
TEXT;
LABEL;
FOR_COND;
INDEX;
RINC_STATEMENT;
LINC_STATEMENT;
RDEC_STATEMENT;
LDEC_STATEMENT;
FOR_CONDITION;
FOR_EXP;
ARRAY;
ARGLISTS;
FUNC_CALL;
FUNC_DEF;
// after changes
}

/* here is where it starts*/
program
     :
                              (function_def | statement )* EOF! {#program = #([STATEMENT, "PROGRAM"],program);}
     ;

statement
     :
          label_statement
        | break_statement SEMICOLON!
         | cont_statement SEMICOLON!
         | do_statement SEMICOLON!
         | if_statement
        | for_statement
         | io_statement SEMICOLON!
         | while_statement
        | assigment_statement SEMICOLON!
         | inc_dec_statement SEMICOLON!
         | statement_block
         | block_func SEMICOLON!
         | unblock_func SEMICOLON!
                    | allow_func SEMICOLON!
         | function_stmt
     ;


arg_list
     :
        ((ID | Integer | d_array) (COMMA! (ID | Integer | d_array))*)?
        {#arg_list = #([ARGLISTS,"ARGLISTS"],arg_list);}
      ;

function_stmt
            :
                        function_call SEMICOLON!
            ;

function_call
            :
                        ID LPAREN! arg_list RPAREN!
                        {#function_call = #([FUNC_CALL,"FUNC_CALL"], function_call);}
            ;

function_def
            :
              "func"! ID LPAREN! arg_list RPAREN! statement_block
                        {#function_def = #([FUNC_DEF,"FUNC_DEF"], function_def);}
        ;

block_func
            :
        "block"^ (exp (COLON! exp)?)
      ;

unblock_func
            :
        "unblock"^ (exp (COLON! exp)?)
      ;

allow_func
            :
        "allow"^ (exp (COLON! exp)?)
      ;

statement_block
      :
                        LCUR! (statement)* RCUR!
                                                    {#statement_block = #([STATEMENT,"STATEMENT"],statement_block); }
            ;
```

```
label_statement
    :
       ID COLON! statement
      {#label_statement = #([LABEL, "LABEL"], label_statement);}
           ;

break_statement  : "break"^ LBRACKET! ID RBRACKET!
    ;

cont_statement
    :
       "continue"^ LBRACKET! ID RBRACKET!
    ;

do_statement
    :
       "do"^ statement "while"! LPAREN! exp RPAREN!
    ;

if_statement
    :
       "if"^ LPAREN! exp RPAREN! statement
      (options {greedy = true;}: "else"! statement)?
    ;

for_statement
    :
       "for"^ LPAREN! for_condition RPAREN! statement
    ;

for_condition
    :
             assigment_statement SEMICOLON! exp SEMICOLON! exp
    ;

io_statement
    :
       "print"^ io
    | "printf"^ LPAREN! (ID | STRINGLITERAL | (LPAREN! exp RPAREN!)) io RPAREN!
    ;
```

```
io
     :
                           exp (COMMA^ exp)*
     ;

while_statement
     :
         "while"^ LPAREN! exp RPAREN! statement
     ;


assigment_statement
     :
         (ID | d_array) (ASSIGN^ | ASSIGNPLUS^ | ASSIGNMINUS^ | ASSIGNMULT^ | ASSIGNDIV^) exp
     ;

inc_dec_statement
     :
         ID INCREMENT^
     | INCREMENT^ ID
     | ID DECREMENT^
     | DECREMENT^ ID
     ;

exp
     :  cond_exp (QMARK^ exp COLON! exp )?
    ;

cond_exp
     :
             logic_term (OR^ logic_term)*
     ;

logic_term
     :
             logic_factor (AND^ logic_factor)*
     ;

logic_factor
     :
             rel_term ("in"^ rel_term)*
     ;

rel_term
     :
       concat_term ((GT^ | GE^ | LS^ | LE^ | E^ | NOTE^) concat_term)*
     ;

concat_term
     :
                         aritm_term ( {#concat_term = #([CONCAT, "CONCAT"], concat_term);} aritm_term)*
     ;

aritm_term
    :
       mult_term (options {greedy=true;}: (PLUS^ | DASH^ ) mult_term)*
     ;

mult_term
    :
       unary_term ((STAR^ | FORWARDSLASH^ | MODULUS^) unary_term)*
     ;

unary_term
     :
       PLUS! postpos_term
       { #unary_term = #([UPL,"UPL"], unary_term); }
     | DASH! postpos_term
        { #unary_term = #([UMN,"UMN"], unary_term); }
    | LOGICALNOT! postpos_term
       { #unary_term = #([ULNOT,"ULNOT"], unary_term);}
     | postpos_term
     ;

postpos_term
     :
       prepos_term (options {greedy=true;}: INCREMENT!
       {#postpos_term = #([RINC,"RINC"],postpos_term);}
    | DECREMENT!
        {#postpos_term = #([RDEC,"RDEC"],postpos_term);})?
     ;
```

```
prepos_term
     :
        INCREMENT! atom
        { #prepos_term = #([LINC,"LINC"], prepos_term);}
     | DECREMENT! atom
 { #prepos_term = #([LDEC,"LDEC"], prepos_term);}
     | atom
     ;

atom
     :
         STRINGLITERAL
    | ID
     | DoubleConst
    | Integer
    | (LPAREN! exp RPAREN!)
    | d_array
     ;

d_array
     :
        ID LBRACKET! exp RBRACKET!
                                                                                           {#d_array = #([ARRAY, "ARRAY"],d_array);}
              ;


/* LEXER */
class DevilLexer extends Lexer;

options {
     charVocabulary = '\u0003'..'\uFFFF';
     exportVocab = Devil;
     k=2;
     testLiterals = false;
}


INCREMENT: "++";
DECREMENT: "--";
PLUS: '+';
DASH: '-';
```

```
LOGICALNOT: '!';
STAR: '*';
FORWARDSLASH: '/';
MODULUS: '%';
GT: '>';
GE: ">=";
LS: '<';
LE: "<=";
NOTE: "!=";
E: "==";
AND: "&&";
OR: "||";
QMARK: '?';
COLON: ':';
ASSIGN: '=';
ASSIGNPLUS: "+=";
ASSIGNMINUS: "-=";
ASSIGNMULT: "*=";
ASSIGNDIV: "/=";
SEMICOLON: ';';


protected
PERIOD: '.';
WS: (' ' | '\t' | '\n' {newline();} | '\r')
                                                              {$setType(Token.SKIP);};
COMMA: ",";

ID_PATTERN
    :
    '\"'!
     ((LETTER)(LETTER|DIGIT|COLON|PERIOD|DASH|'_')*)
     '\"'!
    ;

ID
options{ testLiterals = true; }
   :
    LETTER
     (LETTER | '_' | (DIGIT))*
    ;

STRINGLITERAL
    :
      '"'!
       ( (ESCAPE |~('"' | '\n' | '\r'|'\\' )) | ('"'! '"'))*
       '"'!
    ;

ESCAPE
    :
       '\\'
       ( 'n' { $setText('\n'); }
       | 'r' { $setText('\r'); }
       | 't' { $setText('\t'); }
       | 'b' { $setText('\b'); }
       | '"' { $setText('"'); }
       | 'f' { $setText('\f'); }
       | '\\'{ $setText('\\'); }
       | '\''{ $setText('\''); }
       | {char unicode;} 'u' unicode = UNICODE {$setText(unicode);}
       )
    ;

LBRACKET :'[';
RBRACKET :']';
LPAREN :'(';
RPAREN :')';
LCUR :'{';
RCUR :'}';

COMMENT
    : '#' (options {greedy=false;}:
            ~('\n'|'\r'|'\uFFFF')
       )*(('\n'{newline();}|'\r') { $setType(Token.SKIP);} | ('\uFFFF' {$setType(Token.EOF_TYPE);}))
    ;

NUMBER
    :
       (DIGIT)+
       (((PERIOD)(DIGIT)*(EXPONENT)?){ $setType(DoubleConst); }
        |(EXPONENT) {$setType(DoubleConst); }
        |/*nothing*/ {$setType(Integer); }
        )
    ;

FRACTIONALNUMBER
    :  ((PERIOD)(DIGIT)*(EXPONENT)?){ $setType(DoubleConst); }
    ;

protected
UNICODE returns [char unicodeChar] { String unicode = new String();}
       :(h1:HEXDIGIT!){unicode += h1.getText(); }(h2:HEXDIGIT!){unicode +=
h2.getText();}(h3:HEXDIGIT!){unicode += h3.getText();}(h4:HEXDIGIT!){unicode += h3.getText();
unicodeChar = (char)(java.lang.Integer.parseInt(unicode, 16)); };

protected
DIGIT
    :
        '0'..'9'
    ;

protected
LETTER
    :
        'A'..'Z'
   | 'a'..'z'
    ;

protected
HEXDIGIT
    :
        'a'..'f'
    | 'A'..'F'
    | DIGIT
    ;

protected
EXPONENT
    :
                                                          ('e'|'E')('+'|'-')? (DIGIT)+
                                                           ;
```

```
private DevilInterpreter interpreter = new DevilInterpreter();
            private DevilDataType null_data = new DevilDataType();
            private boolean if_cond = true;
            private boolean run = false;
}

expr returns [ DevilDataType x ]
{

// System.out.println("<WALKER> expr");
     DevilDataType a,b,c;
     x = null_data;

//TODO nullify retwurn
            String out = "";
     String[] vars;
     //String[] params;
     Vector params;
     DevilDataType[] stack;
}

            :#(OR a=expr b=expr) { x=interpreter.or(a,b);
                                                                                          //System.out.println("<WALKER>
OR " );
}
            | #(AND a=expr b=expr) { x=interpreter.and(a,b);
                                                                   //System.out.println("<WALKER> AND" );
                                                                                                    }
            | #(ULNOT a=expr) { x = interpreter.logicalnot(a);
                                                                                          //System.out.println("<WALKER>
ULNOT" );
}
| #(GT a=expr b=expr)          { x = interpreter.gt(a, b);
                                //   System.out.println("<WALKER>
GT"
);
                                }
| #(GE a=expr b=expr)          { x = interpreter.ge(a, b); }
| #(LS a=expr b=expr)          { x = interpreter.ls(a, b); }
| #(LE a=expr b=expr)          { x = interpreter.le(a, b); }
| #(NOTE a=expr b=expr)        { x = interpreter.note(a, b); }
| #(CONCAT a=expr b=expr)            {x = interpreter.concat(a,b);}
| #(E a=expr b=expr)           {
                                    x = interpreter.equals(a, b);
//System.out.println("<WALKER> E"
);// not done
}
| #(PLUS a=expr b=expr) {
 x = interpreter.plus(a, b);
//System.out.println("<WALKER> PLUS
RETURNED \n"+x.name ); // done
 }
| #(DASH a=expr b=expr) { x = interpreter.dash(a, b); }
| #(STAR a=expr b=expr) { x = interpreter.star(a, b); }
| #(FORWARDSLASH a=expr b=expr) { x = interpreter.div(a, b); }
| #(MODULUS a=expr b=expr) { x = interpreter.modulus(a, b); }
| #(ASSIGN a=expr b=expr) {
//System.out.println("<WALKER>
ASSIGN\n"+a.name+ " = "+ b.name ) ;//done
 x = interpreter.assign(a,b);
//System.out.println("<WALKER>
ASSIGN\n"+x.name) ;//done
                                                        }
| #(ASSIGNPLUS a=expr b=expr) { x = interpreter.assignplus(a, b); }
| #(ASSIGNMINUS a=expr b=expr) { x = interpreter.assignminus(a, b); }
| #(ASSIGNMULT a=expr b=expr) { x = interpreter.assignmult(a, b); }
| #(ASSIGNDIV a=expr b=expr) { x = interpreter.assigndiv(a, b); }
| #(RINC a=expr) { x = interpreter.rincrement(a); }
| #(LINC a=expr ) {x = interpreter.lincrement(a); }
                    | #(RDEC a=expr ) { x = interpreter.rdecrement(a); }
                    | #(LDEC a=expr ) { x = interpreter.ldecrement(a); }
| #(INCREMENT a=expr) { x = interpreter.inc_stat(a); }
| #(DECREMENT a=expr) { x = interpreter.dec_stat(a); }
```

```
        | #(NOTHING a=expr b=expr) { x = interpreter.concat(a,b);}
        | #(ARRAY a=expr b=expr) { //allocate array[index] = expr
                                  //System.out.println("ARRAY Found"+a + " " + b);
                                  //System.out.println("ARRAY names "+a.name + " " +
b.name);

                                  x = interpreter.allocate(a,b);

                                  // Allocate an array of name a
                                  // size b:77

                              }

        | #(QMARK cond_exp:. stat1:. stat2:.)
            {
                                          if (interpreter.getForCondition(expr(#cond_exp)))
    x = expr(#stat1);
    else
    x = expr(#stat2);
            }

            //System.out.println("atom intnum 1
              x = interpreter.getNumber(
                //System.out.println("atom intnum 2"+
                    //System.out.println("atom intnum 3"+
            );

                              | doublenum:DoubleConst { x = interpreter.getNumber( doublenum.getText() ); }
                              | str:STRINGLITERAL { x = interpreter.getString( str.getText() ); }
                              | id:ID {
                              x = interpreter.getVariable(
                              id.getText() );
                              }
                              | #(INDEX a=expr) {
                              //System.out.println(a);
                              }
                             ~| #(STATEMENT (statement:. { x =
                              expr(#statement);})*)

                              /*| #("for" a=expr forstat:.) {while(run)
                              x=expr(#forstat);}
```

```
| #(SEMICOLON a=expr relexpr:. increxpr:.)
        {
                                        while(interpreter.getForCondition(expr(#relexpr)))
                {
                        expr(#increxpr);
                        run = true;
                        System.out.println("I ");
                }
                run = false;
        }

| #("in" a=expr b=expr) {System.out.println("In expression");}*/


| #("while" wexp:. whilestat:. )
        {
                                        while(interpreter.getForCondition(expr(#wexp)))
                {
                        x=expr(#whilestat);
                }
        }

| #("do" dostat:. exp:.)
        {
                while(interpreter.getForCondition(expr(#exp)))
                {
                        x=expr(#dostat);
                }
        }

| #("for" a=expr relexpr:. increxpr:. forstat:.)
```

```
                    {
                                          while(interpreter.getForCondition(expr(#relexpr)))
{
                                                expr(#increxpr);x=expr(#forstat);
                         }
                    }

               | #("if" a=expr ifstat:. (elsestat:.)?)
                    {
                         if (a instanceof DevilInteger)
                         {
                                             int val = ((DevilInteger)a).getValue();
                                               if (val==0 && #elsestat!=null)
                              {
                                                     x=expr(#elsestat);
                              }
                              if (val==1)
                              {
                                                      x=expr(#ifstat);
                              }
                         }
                         else
                              throw new DevilException("Not a correct if condition");
               }

               | #("print" a=expr)
                                   {
                                        //System.out.println("\nPRINT "+"Name "+a.name);
                                        interpreter.printOutput(a);
                                        interpreter.println();
                                   }
```

```
        | #(COMMA a=expr x=expr)
                                  {interpreter.printOutput(a);}
        | #("block" a= expr )
                        {
                                          // System.out.println("<block> " + a.name);
                             interpreter.block(a);
                             interpreter.println();
                        }
        | #("unblock" a= expr )
                        {
        // System.out.println("<unblock> " + a.name);
        // interpreter.unblock(a);
                        }
        | #("allow" a= expr )
                        {
                                          // System.out.println("<allow> " + a.name);
                        }

        | #(FUNC_CALL fname:ID params=get_expr )

                   {
                   // stack=get_var
   // stack = new DevilDataType[5];
                   System.out.println("FUNC Invoked\n");

                   interpreter.call_func(fname.getText(),params);
                   }


        | #(FUNC_DEF f1name:ID params=get_expr func_body:.)
        {
```

```
                                                    System.out.println("FUNC_DEF FOUND\n");
                        interpreter.new_func(this, f1name.getText(),params,#func_body);
            }


            ;


get_expr returns [ Vector v ]
{

  DevilDataType a;
            v = null;
}
        : #(ARGLISTS { v = new Vector(); }
             ( a=expr { v.add( a ); }
            )*
         ) { ;
                                        }
        ;




get_var returns[String[] params]
{
// AST first;
            params = null;
            Vector v;
}
```

```
            : #(ARGLISTS {v = new Vector();}
                (first:ID {v.add(first.getText());})*)
            {
                    params = interpreter.VectoString(v);
                }

            ;

get_vars returns [String[] var_list]
{
            Vector v;
        var_list = null;
}
            :
              #(ARGLISTS {v = new Vector();}
                    (first:ID {v.add(first.getText());})*)
             {
                            var_list = interpreter.VectoString(v);
                    }
        ;
INTERPRETER
                                                DevilInterpreter.java

/* block impemented 1.07 am*/

import java.util.*;
import java.io.*;
import antlr.CommonAST;
import antlr.collections.AST;
import antlr.RecognitionException;
import antlr.TokenStreamException;
import antlr.TokenStreamIOException;

public class DevilInterpreter {
 private DevilDataType xdt;
   public LinkedList symbolTable = new LinkedList();
                                             public FileOutputStream out; // declare a file output object
   public PrintStream p; // declare a print stream object
   public boolean firsttime = false;

  public StringBuffer FileString; // to write on the .sh file
 public StringBuffer blockCommand = new StringBuffer("block ");

   public static HashMap functiontable = new HashMap();

   public DevilStack record = new DevilStack(symbolTable);
   public DevilInterpreter() {
        try{
                xdt = new DevilDataType();
                                                        out = new FileOutputStream("myfile.txt");
                                                      // Connect print stream to the output stream
                p = new PrintStream( out );
        } catch(Exception e){
                e.printStackTrace();
        }

   }

   public DevilInterpreter(String name) {
        try {
        xdt = new DevilDataType(name);
```

```
          out = new FileOutputStream("outputFile.sh");

          // Connect print stream to the output stream
          p = new PrintStream( out );
          } catch(Exception e){
                    e.printStackTrace();
          }
    }

    public DevilDataType allocate(DevilDataType a, DevilDataType b) {


// System.out.println("Interpreter called Allocate \n");
          int i = ((DevilInteger)b).getValue();

          Vector vb = new Vector();
          vb.add(0,b);

          // System.out.println("Name is "+((DevilVariable) a).getVarName());

          a.name ="array";
          a.setObj(vb);

          return a;
          // return dtp;
    }




public DevilDataType assign(DevilDataType a, DevilDataType b) {
          DevilDataType dtpa = null;
          boolean var =false;
          boolean error =false;
          boolean added = false;
          int value1 =0;
          String value2=null;
          double value3 =0;
          String type = null; DevilString init_str = new DevilString("");
          DevilInteger b1 = null;
          DevilDouble b2 = null;
          DevilString b3 = null;
// System.out.println(" Assign called \n");
          String varName = null;
          DevilDataType dtp = null;

          System.out.println("A = >"+a);
          System.out.println("B = >"+b);
          try {

          if(b instanceof DevilVariable)
          {
                    // get b from Symbol table

                    dtp =record.lookup(((DevilVariable)b).getVarName());
                    System.out.println("Look up in symbol table for =>"+((DevilVariable)b).getVarName() +"
returned "+ dtp+"\n");

                    // if not found in symbol table , give error
                    if( dtp ==null)
                    {
                              System.out.println("Compilation error occurred, Right hand variable not
previously assigned,\n");
```

```
                        System.out.println("Compilation error occurred, Right hand variable not in symbol
table\n");
                        error = true;
                } else {

                        if (dtp instanceof DevilInteger)
                        {
                        type= "Integer";
                        value1 = ((DevilInteger)dtp).getValue();
                        } else if (dtp instanceof DevilDouble )
                        {
                        type= "Double";
                        value3 = ((DevilDouble)dtp).getValue();
                        } else if (dtp instanceof DevilString)
                        {
                        type= "String";
                        value2 = ((DevilString)dtp).getValue();
                        } else {
                        error = true;
                        System.out.println("Compilation error occurred, Unknown data type\n");
                        }


                }
        }
        else {
                // b is not variable but an atom
                // get type and

                if (b instanceof DevilInteger)
                {
                        type= "Integer";
                        value1 = ((DevilInteger)b).getValue();
                } else if (b instanceof DevilDouble )
                {
type= "Double";
                        value3 = ((DevilDouble)b).getValue();
                } else if (b instanceof DevilString)
                {
                        type= "String";
                        value2 = ((DevilString)b).getValue();
                }
                else {
                        error = true;
                        System.out.println("Compilation error occurred, Unknown data type\n");
                }


                // get value

        }


        if(error!=true)
        {
                if(a instanceof DevilVariable)
                {
                // check if a in symbol table or not.

                                                    if ((a.name!=null)&& (a.name.equals("array")))
                         {
```

```
                                        Vector vb = (Vector)a.getObj();

                           DevilDataType dtp1 = (DevilDataType)vb.firstElement();

                            varName =((DevilVariable)a).getVarName();

                           DevilDataType ddtp = record.lookup(varName);
                           if(ddtp==null)
                           {
                                // not found in symbol table
                                HashMap hm = new HashMap();
                                Vector v =(Vector)a.getObj();
                                                v.add(hm);
                                                 ddtp =a;
                           }

                           Vector v =((Vector)ddtp.getObj());
                           HashMap hv = (HashMap)v.get(1);

                           hv.put(new Integer(((DevilInteger)dtp1).getValue()),b);


                           ddtp.setObj(v);
                                        record.update(varName,ddtp);


                      } else {


                                   varName = ((DevilVariable)a).getVarName();
                       dtpa =record.lookup(varName);
                                  System.out.println("Look up in symbol table for
"+((DevilVariable)a).getVarName() +" returned "+ dtpa+"\n");
```

```
if(dtpa==null)
                         {
                                 // dtpa doesn exist in symbol table need to add it
                                         // Update with new Object
                                 if(type=="String")
                                                       dtpa = new DevilString(value2);
                                 else if(type=="Integer")
                                                      dtpa = new DevilInteger(value1);
                                 else if(type=="Double")
                                                      dtpa = new DevilDouble(value3);

                                                       record.update(varName,dtpa);


                         }
                           else {
                         if(dtpa.typename()==type){
                                 if(type=="String")
                                                    ((DevilString)dtpa).setValue(value2);
                                 else if(type=="Integer")
                                                   ((DevilInteger)dtpa).setValue(value1);
                                 else if(type=="Double")
                                                   ((DevilDouble)dtpa).setValue(value3);

                         } else {
                                 // Update with new Object
                                 if(type=="String")
                                                       dtpa = new DevilString(value2);
                                 else if(type=="Integer")
                                                      dtpa = new DevilInteger(value1);
```

```
                              else if(type=="Double")
                                             dtpa = new DevilDouble(value3);

                                              record.update(varName,dtpa);

                         }
                         }

                    // update a with b's value
                    // if Object Type are different replace it by creating anew one



                         }
                    } else {

                    // wrong l value;
                         System.out.println("COMPILER ERROR :Left side of assignment not a variable
but a constant\n");
                    }

               } else {

                    System.out.println("COMPILER ERROR : Error occurred in getting right side of
assignment\n");

               } }
catch( Exception ex)
   {
          ex.printStackTrace();
   }

          return dtpa;
```

```
}
    public DevilDataType assignplus(DevilDataType a, DevilDataType b) {
          DevilDataType aux = null;
          aux = plus(a,b);
          return assign(a, aux);
    }
    public DevilDataType assignmult(DevilDataType a, DevilDataType b) {
          DevilDataType aux = null;
          aux = star(a,b);
          return assign(a, aux);
    }
    public DevilDataType assigndiv(DevilDataType a, DevilDataType b) {
          DevilDataType aux = null;
          aux = div(a,b);
          return assign(a, aux);
    }
    public DevilDataType assignminus(DevilDataType a, DevilDataType b) {
          DevilDataType aux = null;
          aux = dash(a,b);
          return assign(a, aux);
    }

    public DevilDataType getDevilData(DevilDataType a)
    {
                boolean error = false;
                DevilDataType valueA =null;
                DevilDataType dta = null;
                String varNameA = null;
                String typeA = null;
                if( a instanceof DevilVariable )
```

```
            {
                // look up in symbl table
                varNameA = ((DevilVariable)a).getVarName();
                dta = record.lookup(varNameA);
                if(dta==null)
                {
                        System.out.println("Compiler Error, Variable "+ varNameA + " not assigned or
inited\n");
                } else {

                        if (dta instanceof DevilInteger)
                        {
                                typeA= "Integer";
                                                        valueA = ((DevilInteger)dta);
                        } else if (dta instanceof DevilDouble )
                        {
                                typeA= "Double";
                                                        valueA = ((DevilDouble)dta);
                        } else if (dta instanceof DevilString)
{
                                typeA= "String";
                                valueA = ((DevilString)dta);
                        }
                        else {
                                error = true;
                                System.out.println("Compilation error occurred, Unknown data type\n");
                                valueA = null;
                        }

                }
            } else {
                dta =a;

                        if (dta instanceof DevilInteger)
                        {
                                typeA= "Integer";
                                valueA = ((DevilInteger)dta);
                        } else if (dta instanceof DevilDouble )
                        {
                                typeA= "Double";
                                valueA = ((DevilDouble)dta);
                        } else if (dta instanceof DevilString)
                        {
                                typeA= "String";
                                valueA = ((DevilString)dta);
                        }
                        else {
                                error = true;
                                System.out.println("Compilation error occurred, Unknown data type\n");
                                valueA = null;
                        }


            }


        return valueA;

    }

    public DevilDataType plus(DevilDataType a, DevilDataType b){
        String returnString = new String();
        DevilString init_str = new DevilString("");

        DevilDataType dta = null;
        DevilDataType dtb = null;


        DevilString xs = null;
        DevilInteger xi = null;
        DevilDouble xd = null;
        String s1 = new String(), s2 = new String();
        double d1 = 0, d2 = 0, returnDouble;
        int i1 = 0, i2 = 0, returnInt;
        boolean var1 = false, var2 = false;

        try {  DevilDataType valueA =null;
                DevilDataType valueB =null;
                String varNameA = null;
                String typeA = null;
                String varNameB = null;
                String typeB = null;

                valueA = getDevilData(a);
                valueB = getDevilData(b);


                if (valueA.typename() == null && valueB.typename() == null) // empty str + empty str = empty str,
so we simply return a
                        { //System.out.println("<PLUS> returning ");
                            return a;

                        }
                if (valueA.typename() == valueB.typename()){
                        if(valueA.typename() == "Integer"){
                                returnInt = ((DevilInteger)valueA).getValue()+ ((DevilInteger)valueB).getValue();
                                xi = new DevilInteger(returnInt);
                                xi.setType("Integer");
                            return xi;
                        }
                        else {
                                if(valueA.typename() == "Double"){
                                        returnDouble = ((DevilDouble)valueA).getValue()+
((DevilDouble)valueB).getValue();
                                                          xd = new DevilDouble(returnDouble);
                                        xd.setType("Double");
                                        return xd;
                                }
                      else{
                                                            if(valueA.typename() == "String"){
                                                                    return concat(a,b);
                                        }
                                }
                        }
                }
               else if(valueA.typename()!=valueB.typename())
               {
                        if ((valueA.typename()=="String") || valueB.typename()!="String"){
                                return concat(a,b);
                        }

                                         if ((valueA.typename()!="String")&&(valueB.typename()!="String"))
                        {
```

```
                        if(valueA.typename()=="Integer")
                        {
                                        d1 =((DevilInteger)valueA).getValue();
                        } else {
                                        d1 =((DevilDouble)valueA).getValue();
                        }
                        if(valueB.typename()=="Integer")
                        {
                                        d2 =((DevilInteger)valueB).getValue();
                        } else {
                                        d2 =((DevilDouble)valueB).getValue();
                        }
double retval = d1 + d2;
                      xd = new DevilDouble(retval);
                      xd.setType("Double");
                      return xd;


                }

          } else {

                System.out.println("Compiler error Mismathced data types\n");
                }

        }
        catch(Exception ex)
          {
                                System.out.println("\nException occurred in Interpreter PLUS"+ex);
                ex.printStackTrace();
          }
        return null;
    }
    /**
     *
    .* @param a
    . * @param b
    . * @return

     */
    public DevilDataType concat(DevilDataType a, DevilDataType b){
        //TODO: Concatenation
        String returnString = new String();
        DevilString init_str = new DevilString("");
        DevilString xs = null;
        DevilInteger xi = null;
        DevilDouble xd = null;
        DevilString sd1 = null;
        DevilString sd2 = null;
        DevilString sd3 = null;
        String s1 = new String(), s2 = new String();
        double d1 = 0, d2 = 0, returnDouble;
        int i1 = 0, i2 = 0, returnInt;
        boolean var1 = false, var2 = false;
        String ConcatS = new String();
        String stringvalue = new String();
        String stringvaluedouble = new String();
        String stringvaluetwo = new String();
        String stringvaluedoubletwo = new String();

        try {
```

```
        DevilDataType valueA =null;
        DevilDataType valueB =null;
        String varNameA = null;
        String typeA = null;

        String varNameB = null;
        String typeB = null;

        valueA = getDevilData(a);
        valueB = getDevilData(b);

        if(valueA.typename() == "String" && valueB.typename() == "String"){
            ConcatS = ((DevilString)valueA).getValue() + ((DevilString)valueB).getValue();
            sd1 = new DevilString(ConcatS);
sd1.setType("String");
            return sd1;
        }
        else{
            if(valueA.typename() == "Integer"){
                ConcatS = ((DevilInteger)valueA).getValue() + ((DevilString)valueB).getValue();
                sd1 = new DevilString(ConcatS);
                sd1.setType("String");
                return sd1;
            }
            if(valueB.typename() == "Integer"){
                ConcatS = ((DevilString)valueA).getValue() + ((DevilInteger)valueB).getValue();
                sd1 = new DevilString(ConcatS);
                sd1.setType("String");
                return sd1;
            }
            if(valueA.typename() == "Double"){
                    ConcatS = ((DevilDouble)valueA).getValue() + ((DevilString)valueB).getValue();
                    sd1 = new DevilString(ConcatS);
                    sd1.setType("String");
                    return sd1;
                }
                if(valueB.typename() == "Double"){
                    ConcatS = ((DevilString)valueA).getValue() + ((DevilDouble)valueB).getValue();
                    sd1 = new DevilString(ConcatS);
                    sd1.setType("String");
                    return sd1;
                }
            }
        }
        catch(Exception ex)
            {
                System.out.println("\nException occurred in Interpreter PLUS"+ex);
                ex.printStackTrace();
            }
        return null;

    }
    /**
     *
    .* @param a
    . * @param b
    . * @return

     */
    public DevilDataType dash(DevilDataType a, DevilDataType b){
        String returnString = new String();
        DevilString init_str = new DevilString("");
        DevilString xs = null;
        DevilInteger xi = null;
        DevilDouble xd = null;
        String s1 = new String(), s2 = new String();
        double d1 = 0, d2 = 0, returnDouble;
        int i1 = 0, i2 = 0, returnInt;
        boolean var1 = false, var2 = false;


        try {
           DevilDataType valueA =null;
           DevilDataType valueB =null;
           String varNameA = null;
           String typeA = null;
           String varNameB = null;  String typeB = null;

             valueA = getDevilData(a);
             valueB = getDevilData(b);



             if (valueA.typename() == null && valueB.typename() == null) // empty str + empty str = empty str,
so we simply return a
                  { //System.out.println("<PLUS> returning ");
                     return a;

                  }

             if (valueA.typename() == valueB.typename()){
                  if(valueA.typename() == "Integer"){
                          returnInt = ((DevilInteger)valueA).getValue()- ((DevilInteger)valueB).getValue();
```

```
                        xi = new DevilInteger(returnInt);
                        xi.setType("Integer");
                    return xi;
                }
                else if(valueB.typename() == "Double"){
                                        returnDouble = ((DevilDouble)valueA).getValue()-
((DevilDouble)valueB).getValue();
                        xd = new DevilDouble(returnDouble);
                        xd.setType("Double");
                        return xd;
                }
            }

            else if(valueA.typename()!=valueB.typename())
            {
                                if ((valueA.typename()!="String")&&(valueB.typename()!="String"))
                {
                        if(valueA.typename()=="Integer")
                        {
                                        d1 =((DevilInteger)valueA).getValue();
                        } else {
                                        d1 =((DevilDouble)valueA).getValue();
                        }
                        if(valueB.typename()=="Integer")
                        {
                                        d2 =((DevilInteger)valueB).getValue();
                        } else {
                                        d2 =((DevilDouble)valueB).getValue();
                        }

                        double retval = d1 -d2;
                        xd = new DevilDouble(retval);
                        xd.setType("Double");
                        return xd;


                    }

            } else {

                    System.out.println("Compiler error Mismathced data types\n");
                    }

          }
          catch(Exception ex)  {
                                System.out.println("\nException occurred in Interpreter PLUS"+ex);
                    ex.printStackTrace();
             }
          return null;
     }
     /**
      *
 .* @param a
 . * @param b
 . * @return

      */
     public DevilDataType div(DevilDataType a, DevilDataType b){
          String returnString = new String();
          DevilString init_str = new DevilString("");
          DevilString xs = null;
          DevilInteger xi = null;
```

```
        DevilDouble xd = null;
        String s1 = new String(), s2 = new String();
        double d1 = 0, d2 = 0, returnDouble;
        int i1 = 0, i2 = 0, returnInt;
        boolean var1 = false, var2 = false;


        try {
            DevilDataType valueA =null;
            DevilDataType valueB =null;
            String varNameA = null;
            String typeA = null;
            String varNameB = null;
            String typeB = null;

            valueA = getDevilData(a);
            valueB = getDevilData(b);



            if (valueA.typename() == null && valueB.typename() == null) // empty str + empty str = empty str,
so we simply return a
                { //System.out.println("<PLUS> returning ");
                    return a;

                }

            if (valueA.typename() == valueB.typename()){
                if(valueA.typename() == "Integer"){
                        returnInt = ((DevilInteger)valueA).getValue()/ ((DevilInteger)valueB).getValue();
                        xi = new DevilInteger(returnInt);
                        xi.setType("Integer");
                        return xi;
                    }
                    else if(valueB.typename() == "Double"){
                                                    returnDouble = ((DevilDouble)valueA).getValue()/
((DevilDouble)valueB).getValue();
                             xd = new DevilDouble(returnDouble);
                             xd.setType("Double");
                             return xd;
                    }
              }

              else if(valueA.typename()!=valueB.typename())
              {
if ((valueA.typename()!="String")&&(valueB.typename()!="String"))
                {
                                                          if(valueA.typename()=="Integer")
                          {
                                 d1 =((DevilInteger)valueA).getValue();
                          } else {
                                 d1 =((DevilDouble)valueA).getValue();
                          }
                                                          if(valueB.typename()=="Integer")
                          {
                                 d2 =((DevilInteger)valueB).getValue();
                          } else {
                                 d2 =((DevilDouble)valueB).getValue();
                          }

                          double retval = d1 / d2;
                                                            xd = new DevilDouble(retval);
                          xd.setType("Double");
                        return xd;

                }

            } else {

                System.out.println("Compiler error Mismathced data types\n");
                }

        }
        catch(Exception ex)
          {
                System.out.println("\nException occurred in Interpreter PLUS"+ex);
                ex.printStackTrace();
          }
        return null;
}
/**
 *
□.* @param a
□. * @param b
□. * @return

 */
public DevilDataType star(DevilDataType a, DevilDataType b){
        String returnString = new String();
        DevilString init_str = new DevilString("");
        DevilString xs = null;
        DevilInteger xi = null;
        DevilDouble xd = null;
        String s1 = new String(), s2 = new String();
        double d1 = 0, d2 = 0, returnDouble;
        int i1 = 0, i2 = 0, returnInt;
        boolean var1 = false, var2 = false;


        try {
            DevilDataType valueA =null;
            DevilDataType valueB =null;
            String varNameA = null;
            String typeA = null;
            String varNameB = null;
            String typeB = null;  valueA = getDevilData(a);
              valueB = getDevilData(b);



              if (valueA.typename() == null && valueB.typename() == null) // empty str + empty str = empty str,
so we simply return a
                   { //System.out.println("<PLUS> returning ");
                      return a;

                   }

              if (valueA.typename() == valueB.typename()){
                   if(valueA.typename() == "Integer"){
                           returnInt = ((DevilInteger)valueA).getValue()* ((DevilInteger)valueB).getValue();
                           xi = new DevilInteger(returnInt);
                           xi.setType("Integer");
                      return xi;
                   }
                   else if(valueB.typename() == "Double"){
```

```
                                                returnDouble = ((DevilDouble)valueA).getValue()*
((DevilDouble)valueB).getValue();
                        xd = new DevilDouble(returnDouble);
                        xd.setType("Double");
                        return xd;
                }
         }

        else if(valueA.typename()!=valueB.typename())
        {
                                if ((valueA.typename()!="String")&&(valueB.typename()!="String"))
                {
                        if(valueA.typename()=="Integer")
                        {
                                        d1 =((DevilInteger)valueA).getValue();
                        } else {
                                        d1 =((DevilDouble)valueA).getValue();
                        }
                        if(valueB.typename()=="Integer")
                        {
                                        d2 =((DevilInteger)valueB).getValue();
                        } else {
                                        d2 =((DevilDouble)valueB).getValue();
                        }

                        double retval = d1 * d2;
                        xd = new DevilDouble(retval);
                        xd.setType("Double");
                        return xd;
```

```
                    }

                } else {

                    System.out.println("Compiler error Mismathced data types\n");
                    }

            }
            catch(Exception ex)
               {
                                        System.out.println("\nException occurred in Interpreter PLUS"+ex);
ex.printStackTrace();
               }
            return null;
    }
    /**
     *
.* @param a
. * @param b
. * @return

     */
    public DevilDataType modulus(DevilDataType a, DevilDataType b){
            DevilInteger xi = null;
            DevilDouble xd = null;
            double d1 = 0, d2 = 0, returnDouble;
            int i1 = 0, i2 = 0, returnInt;
            boolean var1 = false, var2 = false;

            try {
               DevilDataType valueA =null;
            DevilDataType valueB =null;
            String varNameA = null;
            String typeA = null;
            String varNameB = null;
            String typeB = null;

            valueA = getDevilData(a);
            valueB = getDevilData(b);



            if (valueA.typename() == null && valueB.typename() == null) // empty str + empty str = empty str,
so we simply return a
                 { //System.out.println("<PLUS> returning ");
                    return a;

                 }

            if (valueA.typename() == valueB.typename()){
                 if(valueA.typename() == "Integer"){
                          returnInt = ((DevilInteger)valueA).getValue()% ((DevilInteger)valueB).getValue();
                          xi = new DevilInteger(returnInt);
                          xi.setType("Integer");
                    return xi;
                 }
                 else if(valueB.typename() == "Double"){
                                              returnDouble = ((DevilDouble)valueA).getValue()%
((DevilDouble)valueB).getValue();
                          xd = new DevilDouble(returnDouble);
                          xd.setType("Double");
                          return xd;
                 }
```

```
         }

        else if(valueA.typename()!=valueB.typename())
        {
                                if ((valueA.typename()!="String")&&(valueB.typename()!="String"))
              {
                   if(valueA.typename()=="Integer")
                   {
                                          d1 =((DevilInteger)valueA).getValue();
                   } else {
                                          d1 =((DevilDouble)valueA).getValue();
}
                                            if(valueB.typename()=="Integer")
                  {
                         d2 =((DevilInteger)valueB).getValue();
                  } else {
                         d2 =((DevilDouble)valueB).getValue();
                  }

                  double retval = d1 % d2;
                                             xd = new DevilDouble(retval);
                  xd.setType("Double");
                  return xd;


             }

      } else {

             System.out.println("Compiler error Mismathced data types\n");
             }
```

```
    }
    catch(Exception ex)
      {
          System.out.println("\nException occurred in Interpreter PLUS"+ex);
          ex.printStackTrace();
      }
    return null;
}
/**
 *
□.* @param a
□. * @return

 */
public DevilDataType Iincrement(DevilDataType a){
    // check if a is in the symbol table
    DevilDataType valueA =null;
    valueA = getDevilData(a);
          if (valueA.typename() == "Integer"){
            //set the value of a, but use the old value in the expression
                                  int aux = ((DevilInteger)valueA).getValue();
            aux++;
            ((DevilInteger)valueA).setValue(aux);
            return valueA;
          }
          //TODO we have to check if we can do this on doubles too
          else{
          if(a.typename() == "Double"){
            //set the value of a, but use the old value in the expression
                                double aux = ((DevilDouble)valueA).getValue();
```

```
                    aux = aux++;
                    ((DevilDouble)valueA).setValue(aux);
                    return valueA;
                  }
         else
              System.err.println("This operator can't be aplied on this type");
         }
             return null;

  }
  /**
   *
. * @param a
. * @return

     */
    public DevilDataType inc_stat(DevilDataType a){
          if(!(a instanceof DevilVariable)){
             System.err.println("ERROR: You can't apply this operator on a costant.");
          }

                   for (int j= 0; j<symbolTable.size();j++){
                            if (((SymbolTableElement) symbolTable.get(j)).getName().equals(((DevilVariable)
a).getVarName())){
                   if (((SymbolTableElement) symbolTable.get(j)).getValue() instanceof DevilInteger){
                      int aux = ((DevilInteger)((SymbolTableElement)
symbolTable.get(j)).getValue()).getValue();
                      ((DevilInteger)((SymbolTableElement) symbolTable.get(j)).getValue()).setValue(aux +
1);
                      return a;
                   }//end if
```

```
                if (((SymbolTableElement) symbolTable.get(j)).getValue() instanceof DevilDouble){
                   DevilDouble xd = new DevilDouble(1);
                   xd.setType("Double");
                   return assignplus(a,xd);

                }//end if
                if (((SymbolTableElement) symbolTable.get(j)).getValue() instanceof DevilString){
                              System.err.println("ERROR: You can't apply this operator on a string.");
                }
            }//end if
        }//end for
        return null;
    }
    /**
     *
 .* @param a
 . * @return

     */
    public DevilDataType ldecrement(DevilDataType a){
        // check if a is in the symbol table
        DevilDataType valueA =null;
        valueA = getDevilData(a);
                if (valueA.typename() == "Integer"){
                   //set the value of a, but use the old value in the expression
                   int aux = ((DevilInteger)valueA).getValue();
                   aux--;
                   ((DevilInteger)valueA).setValue(aux);
                   return valueA;
                }
                //TODO we have to check if we can do this on doubles too
                else{
                if(a.typename() == "Double"){
                   //set the value of a, but use the old value in the expression
                   double aux = ((DevilDouble)valueA).getValue();
                   aux = aux--;
                   ((DevilDouble)valueA).setValue(aux);
                   return valueA;
                }
          else
                System.err.println("This operator can't be aplied on this type");
          }
              return null;
    }  public DevilDataType rincrement(DevilDataType a){
// check if a is in the symbol table
        DevilDataType valueA =null;
        valueA = getDevilData(a);
                if (valueA.typename() == "Integer"){
                   //set the value of a, but use the old value in the expression
                   int aux = ((DevilInteger)valueA).getValue();
                   aux++;
                                                        ((DevilInteger)valueA).setValue(aux);
                   return valueA;
                }
                //TODO we have to check if we can do this on doubles too
                else{
                if(a.typename() == "Double"){
                   //set the value of a, but use the old value in the expression
                   double aux = ((DevilDouble)valueA).getValue();
                   aux = aux++;
```

```
                                                    ((DevilDouble)valueA).setValue(aux);
                         return valueA;
                      }
              else
                    System.err.println("This operator can't be aplied on this type");
              }
                 return null;
          }


              /**
           *
        .* @param a
        . * @return

           */
          public DevilDataType rdecrement(DevilDataType a){
               // check if a is in the symbol table
               DevilDataType valueA =null;
               valueA = getDevilData(a);
                      if (valueA.typename() == "Integer"){
                         //set the value of a, but use the old value in the expression
                         int aux = ((DevilInteger)valueA).getValue();
                         aux--;
                                                    ((DevilInteger)valueA).setValue(aux);
                         return valueA;
                      }
                      //TODO we have to check if we can do this on doubles too
                      else{
                      if(a.typename() == "Double"){
                         //set the value of a, but use the old value in the expression
                    double aux = ((DevilDouble)valueA).getValue();
                    aux = aux--;
                                                   ((DevilDouble)valueA).setValue(aux);
                    return valueA;
                  }
           else
                 System.err.println("This operator can't be aplied on this type");
           }
               return null;
     }
     /**
      *
   .* @param a
   . * @return

      */
     public DevilDataType dec_stat(DevilDataType a){

          if(!(a instanceof DevilVariable)){
             System.err.println("ERROR: You can't apply this operator on a constant.");
          }

          for (int j= 0; j<symbolTable.size();j++){
                         if (((SymbolTableElement) symbolTable.get(j)).getName().equals(((DevilVariable)
a).getVarName())){
                 if (((SymbolTableElement) symbolTable.get(j)).getValue() instanceof DevilInteger){
                    int aux = ((DevilInteger)((SymbolTableElement)
symbolTable.get(j)).getValue()).getValue();
                    ((DevilInteger)((SymbolTableElement) symbolTable.get(j)).getValue()).setValue(aux -
1);
                    return a;
```

```
            } //end if
            if (((SymbolTableElement) symbolTable.get(j)).getValue() instanceof DevilDouble){
              DevilDouble xd = new DevilDouble(1);
              xd.setType("Double");
              return assignminus(a,xd);

            } //end if
            if (((SymbolTableElement) symbolTable.get(j)).getValue() instanceof DevilString){
                        System.err.println("ERROR: You can't apply this operator on a string.");
            }
          } //end if
        } //end for
        return null;
    }
    /**
     *
  □.* @param a
  □. * @param b
  □. * @return

     */
    public DevilDataType equals(DevilDataType a, DevilDataType b){
        boolean returnVal = false;
        DevilString xs = null;
        DevilInteger xi = null;
        DevilDouble xd = null;
        String s1 = new String(), s2 = new String();
        double d1 = 0, d2 = 0;
        int i1 = 0, i2 = 0, returnInt;
        boolean var1 = false, var2 = false;
```

```
try {
        DevilDataType valueA =null;
        DevilDataType valueB =null;
        String varNameA = null;
        String typeA = null;
        String varNameB = null;
        String typeB = null;

        valueA = getDevilData(a);
        valueB = getDevilData(b);
        if(valueA.typename() == "String"){
              s1 = ((DevilString)valueA).getValue();
        }
        if(valueB.typename() == "String"){
              s2 = ((DevilString)valueB).getValue();
        }
        if(valueA.typename() == "Integer"){
              i1 = ((DevilInteger)valueA).getValue();
        }
  if(valueB.typename() == "Integer"){
              i2 = ((DevilInteger)valueB).getValue();
   }
        if(valueA.typename() == "Double"){
              d1 = ((DevilDouble)valueA).getValue();
        }
  if(valueB.typename() == "Double"){
              d2 = ((DevilDouble)valueB).getValue();
   }

        if(valueA.typename() == valueB.typename()){
              if(valueA.typename() == "String"){
                    returnVal = s1.equals(s2);

                                                                                         }
            if(valueA.typename() == "Integer"){
                    returnVal = i1 == i2;
}
            if(valueA.typename() == "Double"){
                    returnVal = d1 == d2;
            }

                    }
    if(valueA.typename() != valueB.typename()){
            if(valueA.typename() == "Double" && valueB.typename() == "Integer"){
            returnVal = d1 == i2;
            }
            if(valueA.typename() == "Integer" && valueB.typename() == "Integer"){
            returnVal = i1 == d2;
            }
    }

            if (returnVal){
                    xi = new DevilInteger(1);
                    xi.setType("Integer");
                    return xi;
            }
            else {
                    xi = new DevilInteger(0);
                    xi.setType("Integer");
                    return xi;
            }
```

```
          }
          catch(Exception ex)
              {
                      System.out.println("\nException occurred in Interpreter PLUS"+ex);
                      ex.printStackTrace();
              }
          System.err.println("ERROR:Type mismatch");
          return null;
}
    /**
     *
.* @param a
. * @param b
. * @return

     */  public DevilDataType ge(DevilDataType a, DevilDataType b){
        boolean returnVal = false;
        DevilString xs = null;
        DevilInteger xi = null;
        DevilDouble xd = null;
        String s1 = new String(), s2 = new String();
        double d1 = 0, d2 = 0;
        int i1 = 0, i2 = 0, returnInt;
        boolean var1 = false, var2 = false;

        try {
            DevilDataType valueA =null;
            DevilDataType valueB =null;
            String varNameA = null;
            String typeA = null;
```

```
      String varNameB = null;
      String typeB = null;

      valueA = getDevilData(a);
      valueB = getDevilData(b);
if(valueA.typename() == "String"){
       s1 = ((DevilString)valueA).getValue();
      }
if(valueB.typename() == "String"){
       s2 = ((DevilString)valueB).getValue();
      }
if(valueA.typename() == "Integer"){
       i1 = ((DevilInteger)valueA).getValue();
      }
if(valueB.typename() == "Integer"){
       i2 = ((DevilInteger)valueB).getValue();
 }
if(valueA.typename() == "Double"){
       d1 = ((DevilDouble)valueA).getValue();
      }
if(valueB.typename() == "Double"){
       d2 = ((DevilDouble)valueB).getValue();
 }

if(valueA.typename() == valueB.typename()){
       if(valueA.typename() == "String"){
               returnVal = s1.equals(s2);
       }
       if(valueA.typename() == "Integer"){
               returnVal = i1 >= i2;
    }
```

```
            if(valueA.typename() == "Double"){
                    returnVal = d1 >= d2;
            }

                    }
        if(valueA.typename() != valueB.typename()){
            if(valueA.typename() == "Double" && valueB.typename() == "Integer"){
            returnVal = d1 >= i2;
            }
            if(valueA.typename() == "Integer" && valueB.typename() == "Double"){
            returnVal = i1 >= d2;
            }
           }

            if (returnVal){
                    xi = new DevilInteger(1);
xi.setType("Integer");
                        return xi;
                }
                else {
                        xi = new DevilInteger(0);
                        xi.setType("Integer");
                        return xi;
                }

          }
          catch(Exception ex)
             {
                System.out.println("\nException occurred in Interpreter PLUS"+ex);
                ex.printStackTrace();
             }
        System.err.println("ERROR:Type mismatch");
        return null;
}

/**
 *
.* @param a
. * @param b
. * @return

 */
public DevilDataType Is(DevilDataType a, DevilDataType b){
        boolean returnVal = false;
        DevilString xs = null;
        DevilInteger xi = null;
        DevilDouble xd = null;
        String s1 = new String(), s2 = new String();
        double d1 = 0, d2 = 0;
        int i1 = 0, i2 = 0, returnInt;
        boolean var1 = false, var2 = false;

        try {
            DevilDataType valueA =null;
            DevilDataType valueB =null;
            String varNameA = null;
            String typeA = null;
            String varNameB = null;
            String typeB = null;

            valueA = getDevilData(a);
            valueB = getDevilData(b);
```

```
    if(valueA.typename() == "String"){
        s1 = ((DevilString)valueA).getValue();
    }
    if(valueB.typename() == "String"){
        s2 = ((DevilString)valueB).getValue();
    }
    if(valueA.typename() == "Integer"){
        i1 = ((DevilInteger)valueA).getValue();
    }
if(valueB.typename() == "Integer"){
        i2 = ((DevilInteger)valueB).getValue();
 }
    if(valueA.typename() == "Double"){
        d1 = ((DevilDouble)valueA).getValue();
    }
if(valueB.typename() == "Double"){
        d2 = ((DevilDouble)valueB).getValue();
 }

    if(valueA.typename() == valueB.typename()){
        if(valueA.typename() == "String"){
              returnVal = !(s1.equals(s2));
        }
        if(valueA.typename() == "Integer"){
              returnVal = i1 < i2;
   }
        if(valueA.typename() == "Double"){
              returnVal = d1 < d2;
        }

             }
```

```
            if(valueA.typename() != valueB.typename()){
                if(valueA.typename() == "Double" && valueB.typename() == "Integer"){
                returnVal = d1 < i2;
                }
                if(valueA.typename() == "Integer" && valueB.typename() == "Double"){
                returnVal = i1 < d2;
                }
            }

                if (returnVal){
                        xi = new DevilInteger(1);
                        xi.setType("Integer");
                        return xi;
                }
                else {
                        xi = new DevilInteger(0);
                        xi.setType("Integer");
                        return xi;
                }

        }
        catch(Exception ex)
            {
                System.out.println("\nException occurred in Interpreter PLUS"+ex);
                ex.printStackTrace();
            }
        System.err.println("ERROR:Type mismatch");
        return null;
    }

    /**
```

```
 *
.* @param a
. * @param b
. * @return

 */
public DevilDataType le(DevilDataType a, DevilDataType b){
       boolean returnVal = false;
       DevilString xs = null;
       DevilInteger xi = null;
       DevilDouble xd = null;
       String s1 = new String(), s2 = new String();
       double d1 = 0, d2 = 0;
       int i1 = 0, i2 = 0, returnInt;
       boolean var1 = false, var2 = false;


       try {  DevilDataType valueA =null;
     DevilDataType valueB =null;
     String varNameA = null;
     String typeA = null;
     String varNameB = null;
     String typeB = null;

     valueA = getDevilData(a);
     valueB = getDevilData(b);
     if(valueA.typename() == "String"){
           s1 = ((DevilString)valueA).getValue();
     }
     if(valueB.typename() == "String"){
           s2 = ((DevilString)valueB).getValue();
```

```
        }
        if(valueA.typename() == "Integer"){
                i1 = ((DevilInteger)valueA).getValue();
        }
    if(valueB.typename() == "Integer"){
                i2 = ((DevilInteger)valueB).getValue();
     }
        if(valueA.typename() == "Double"){
                d1 = ((DevilDouble)valueA).getValue();
        }
    if(valueB.typename() == "Double"){
                d2 = ((DevilDouble)valueB).getValue();
     }

        if(valueA.typename() == valueB.typename()){
                if(valueA.typename() == "String"){
                        returnVal = s1.equals(s2);
               }
                if(valueA.typename() == "Integer"){
                        returnVal = i1 <= i2;
       }
                if(valueA.typename() == "Double"){
                        returnVal = d1 <= d2;
               }

                       }
        if(valueA.typename() != valueB.typename()){
                if(valueA.typename() == "Double" && valueB.typename() == "Integer"){
                returnVal = d1 <= i2;
               }
                if(valueA.typename() == "Integer" && valueB.typename() == "Double"){
```

```
            returnVal = i1 <= d2;
          }
      }

          if (returnVal){
                      xi = new DevilInteger(1);
                      xi.setType("Integer");
                      return xi;
          }
          else {
                      xi = new DevilInteger(0);
                      xi.setType("Integer");
                      return xi;
          }

  }
  catch(Exception ex)  {
                    System.out.println("\nException occurred in Interpreter PLUS"+ex);
                    ex.printStackTrace();
              }
          System.err.println("ERROR:Type mismatch");
          return null;
}
    /**
     *
.* @param a
. * @param b
. * @return

     */
    public DevilDataType gt(DevilDataType a, DevilDataType b){
```

```
 boolean returnVal = false;
 DevilString xs = null;
 DevilInteger xi = null;
 DevilDouble xd = null;
 String s1 = new String(), s2 = new String();
 double d1 = 0, d2 = 0;
 int i1 = 0, i2 = 0, returnInt;
 boolean var1 = false, var2 = false;


 try {
     DevilDataType valueA =null;
     DevilDataType valueB =null;
     String varNameA = null;
     String typeA = null;
     String varNameB = null;
     String typeB = null;

     valueA = getDevilData(a);
     valueB = getDevilData(b);
     if(valueA.typename() == "String"){
           s1 = ((DevilString)valueA).getValue();
     }
     if(valueB.typename() == "String"){
           s2 = ((DevilString)valueB).getValue();
     }
     if(valueA.typename() == "Integer"){
           i1 = ((DevilInteger)valueA).getValue();
     }
if(valueB.typename() == "Integer"){
           i2 = ((DevilInteger)valueB).getValue();
```

```
         }
            if(valueA.typename() == "Double"){
                 d1 = ((DevilDouble)valueA).getValue();
            }
        if(valueB.typename() == "Double"){
                 d2 = ((DevilDouble)valueB).getValue();
         }

            if(valueA.typename() == valueB.typename()){
                 if(valueA.typename() == "String"){
                                                      returnVal = !(s1.equals(s2));
                 }
                 if(valueA.typename() == "Integer"){
                         returnVal = i1 > i2;
          }
                 if(valueA.typename() == "Double"){
                         returnVal = d1 > d2;
}

                          }
          if(valueA.typename() != valueB.typename()){
               if(valueA.typename() == "Double" && valueB.typename() == "Integer"){
               returnVal = d1 > i2;
               }
               if(valueA.typename() == "Integer" && valueB.typename() == "Double"){
               returnVal = i1 > d2;
               }
          }

               if (returnVal){
                      xi = new DevilInteger(1);
```

```
                        xi.setType("Integer");
                        return xi;
                }
                else {
                        xi = new DevilInteger(0);
                        xi.setType("Integer");
                        return xi;
                }

        }
        catch(Exception ex)
          {
                System.out.println("\nException occurred in Interpreter PLUS"+ex);
                ex.printStackTrace();
          }
        System.err.println("ERROR:Type mismatch");
        return null;
    }

    /**
     *
    */
    public DevilDataType note(DevilDataType a, DevilDataType b){
        boolean returnVal = false;
        DevilString xs = null;
        DevilInteger xi = null;
        DevilDouble xd = null;
        String s1 = new String(), s2 = new String();
        double d1 = 0, d2 = 0;
        int i1 = 0, i2 = 0, returnInt;
        boolean var1 = false, var2 = false;
```

```
  try {
      DevilDataType valueA =null;
      DevilDataType valueB =null;
      String varNameA = null;
      String typeA = null;
      String varNameB = null;
      String typeB = null;

      valueA = getDevilData(a);
      valueB = getDevilData(b);
      if(valueA.typename() == "String"){
            s1 = ((DevilString)valueA).getValue();
      }
      if(valueB.typename() == "String"){
            s2 = ((DevilString)valueB).getValue();
      }
      if(valueA.typename() == "Integer"){
            i1 = ((DevilInteger)valueA).getValue();
      }
if(valueB.typename() == "Integer"){
            i2 = ((DevilInteger)valueB).getValue();
 }
      if(valueA.typename() == "Double"){
            d1 = ((DevilDouble)valueA).getValue();
      }
if(valueB.typename() == "Double"){
            d2 = ((DevilDouble)valueB).getValue();

 }

      if(valueA.typename() == valueB.typename()){
        if(valueA.typename() == "String"){
                returnVal = !(s1.equals(s2));
        }
        if(valueA.typename() == "Integer"){
                returnVal = i1 != i2;
}
        if(valueA.typename() == "Double"){
                returnVal = d1 > d2;
        }

                }
  if(valueA.typename() != valueB.typename()){
        if(valueA.typename() == "Double" && valueB.typename() == "Integer"){
        returnVal = d1 != i2;
        }
        if(valueA.typename() == "Integer" && valueB.typename() == "Double"){
        returnVal = i1 != d2;
        }
  }

        if (returnVal){
                xi = new DevilInteger(1);
                xi.setType("Integer");
                return xi;
        }
        else {
                xi = new DevilInteger(0);
                xi.setType("Integer");
                return xi;
        }
        }
        catch(Exception ex)
            {
                    System.out.println("\nException occurred in Interpreter PLUS"+ex);
                    ex.printStackTrace();
            }
        System.err.println("ERROR:Type mismatch");
        return null;
}
/**
 *
□.* @param a
□. * @return

 */
public DevilDataType logicalnot(DevilDataType a) {
        DevilInteger xi = null; int val_a, val_b;

           if (a instanceof DevilInteger){
               val_a = ((DevilInteger)a).getValue();
               if (val_a == 1){
                       xi = new DevilInteger(0);
                       xi.setType("Integer");
                       return xi;
               }
               else {
                       xi = new DevilInteger(1);
                       xi.setType("Integer");
                       return xi;
               }
        }
        else if (a instanceof DevilVariable)
            for (int j= 0; j<symbolTable.size();j++){
                  if (((SymbolTableElement) symbolTable.get(j)).getName().equals(((DevilVariable)
a).getVarName())){
                      a.setType(((SymbolTableElement)symbolTable.get(j)).getType());
                      if (a.typename() == "Integer"){
                            val_a =
((DevilInteger)((SymbolTableElement)symbolTable.get(j)).getValue()).getValue();
                            if (val_a == 0){
                                xi = new DevilInteger(1);
                                xi.setType("Integer");
                                return xi;
                            }
                            else{
                                xi = new DevilInteger(0);
                                xi.setType("Integer");
                                return xi;
                            }
                      }
                                              else throw new DevilException("Not the correct type");
                  }

            }
        return null;

    }

    /**
     *
□.* @param a
```

```
. * @param b
. * @return

  */
  public DevilDataType and(DevilDataType a, DevilDataType b) {
        DevilInteger xi = null;
        int val_a, val_b;

        if (a instanceof DevilInteger){
            val_a = ((DevilInteger)a).getValue();
            if (val_a == 0){
                  xi = new DevilInteger(0);
                  xi.setType("Integer");
                  return xi;
            }
        }
        else if (a instanceof DevilVariable)
            for (int j= 0; j<symbolTable.size();j++){
if (((SymbolTableElement) symbolTable.get(j)).getName().equals(((DevilVariable)
a).getVarName())){
                        a.setType(((SymbolTableElement)symbolTable.get(j)).getType());
                        if (a.typename() == "Integer"){
                              val_a =
((DevilInteger)((SymbolTableElement)symbolTable.get(j)).getValue()).getValue();
                              if (val_a == 0){
                                  xi = new DevilInteger(0);
                                  xi.setType("Integer");
                                  return xi;
                              }
                        }
```

```
                                        else throw new DevilException("Not the correct type");
                    }

                }

            if (b instanceof DevilInteger){
                val_b = ((DevilInteger)b).getValue();
                if (val_b == 0){
                    xi = new DevilInteger(0);
                    xi.setType("Integer");
                    return xi;
                }
            }
            else if (b instanceof DevilVariable)
                for (int j= 0; j<symbolTable.size();j++){
                    if (((SymbolTableElement) symbolTable.get(j)).getName().equals(((DevilVariable)
b).getVarName())){
                        b.setType(((SymbolTableElement)symbolTable.get(j)).getType());
                        if (b.typename() == "Integer"){
                            val_b =
((DevilInteger)((SymbolTableElement)symbolTable.get(j)).getValue()).getValue();
                            if (val_b == 0){
                                xi = new DevilInteger(0);
                                xi.setType("Integer");
                                return xi;
                            }
                        }
                                        else throw new DevilException("Not the correct type");
                    }

                }
```

```
          xi = new DevilInteger(1);
          xi.setType("Integer");
          return xi;
    }
    /**
     *
.* @param a
. * @param b
. * @return

     */
    public DevilDataType or(DevilDataType a, DevilDataType b) {
          DevilInteger xi = null;
          int val_a, val_b;

          if (a instanceof DevilInteger){
             val_a = ((DevilInteger)a).getValue();
             if (val_a == 1){
                   xi = new DevilInteger(1);
xi.setType("Integer");
                   return xi;
             }
          }
          else if (a instanceof DevilVariable)
             for (int j= 0; j<symbolTable.size();j++){
                   if (((SymbolTableElement) symbolTable.get(j)).getName().equals(((DevilVariable)
a).getVarName())){
                      a.setType(((SymbolTableElement)symbolTable.get(j)).getType());
                      if (a.typename() == "Integer"){
                            val_a =
((DevilInteger)((SymbolTableElement)symbolTable.get(j)).getValue()).getValue();
                                if (val_a == 1){
                                    xi = new DevilInteger(1);
                                    xi.setType("Integer");
                                    return xi;
                                }
                          }
                                                     else throw new DevilException("Not the correct type");
                     }

               }

           if (b instanceof DevilInteger){
               val_b = ((DevilInteger)b).getValue();
               if (val_b == 1){
                     xi = new DevilInteger(1);
                     xi.setType("Integer");
                     return xi;
               }
           }
           else if (b instanceof DevilVariable)
               for (int j= 0; j<symbolTable.size();j++){
                     if (((SymbolTableElement) symbolTable.get(j)).getName().equals(((DevilVariable)
b).getVarName())){
                          b.setType(((SymbolTableElement)symbolTable.get(j)).getType());
                          if (b.typename() == "Integer"){
                                val_b =
((DevilInteger)((SymbolTableElement)symbolTable.get(j)).getValue()).getValue();
                                if (val_b == 1){
                                    xi = new DevilInteger(1);
                                    xi.setType("Integer");
```

```
                            return xi;
                         }
                   }
                                          else throw new DevilException("Not the correct type");
                }

           }

        xi = new DevilInteger(0);
        xi.setType("Integer");
        return xi;

   }
   /**
    *
□.* @param s
□. * @return

    */
   public DevilDataType getNumber(String s) {
        String type=""; int aux=100;
      double aux1=0.0;
      // System.out.println("Got String DevilDataType: getNumber"+s);
      try{
         aux = Integer.parseInt(s);
         type = "Integer";
      }catch (NumberFormatException e1){
         //System.out.println("It is not a integer"+e1);
                                      System.out.println(" e1 -> DevilInterpreter.getNumber"+e1 );
        // e1.printStackTrace();
```

```
            try {
                  aux1 = Double.parseDouble(s);
                  type = "Double";
            }catch (NumberFormatException e2){
                                            System.out.println(" e2 -> DevilInterpreter.getNumber"+e2 );
                  //e2.printStackTrace();
            }
        }

        // System.out.println("DevilDataType: type"+type+" val "+aux);

        if (type.equals("Integer")){
            //System.out.println(Integer.parseInt(s));
            //System.out.println("DevilDataType -Creating DevilInteger");
            DevilInteger devilInteger= new DevilInteger(Integer.parseInt(s));
            devilInteger.setType("Integer");
            //System.out.println("DevilDataType -DevilInteger"+aux);
            return devilInteger;
        }
        if (type.equals("Double")){
            //System.out.println(Double.parseDouble(s));
            DevilDouble devilDouble= new DevilDouble(Double.parseDouble(s));
            devilDouble.setType("Double");
            //System.out.println("DevilDataType -DevilDouble"+aux1);
            return devilDouble;
        }
        return null;
    }
    /**
     *
    .* @param s
```

```
□. * @return

 */
public DevilDataType getVariable(String s) {
        DevilVariable var = new DevilVariable(s);
        return var;
}
/**
 *
□.* @param s
□. * @return

 */
public DevilDataType getString(String s) {
        DevilString xs = new DevilString(s);
        xs.setType("String");
        return xs;
}
/**
 *
*
 */  public void println(){
           System.out.println();
    }
    /**
     *
□.* @param b
□. * @return

     */
    public boolean getForCondition(DevilDataType b){
            int val = 0;
            DevilInteger xi = null;
            if (b instanceof DevilInteger){
                val = ((DevilInteger)b).getValue();
            }
            if (val==1) return true;
            return false;

    }


public boolean writeToFile(String a)
{
            FileOutputStream ostream = null;
            try {
            ostream = new FileOutputStream("devil.sh",firsttime);
            if(firsttime==false)
            {
                        firsttime = true;
            }
            PrintWriter pw = new PrintWriter(ostream);
            pw.println(a);
            pw.close();
            ostream.close();
            } catch(Exception ex)
            {
                        ex.printStackTrace();
                        return false;
            }
            return true;
```

```
}

public void block(DevilDataType param)
{
          int value;
          String varName="";
          //FileOutputStream ostream = null;
          boolean searchtable=false;
          boolean notfound = true;
          StringBuffer blockString= new StringBuffer();
          System.out.println(" inside block function typename and obj "+param.typename()+" "+param);
          try {

          //ostream = new FileOutputStream("devil.sh",firsttime);
          //if(firsttime==false)
          //{
          // firsttime = true;
          //}
          //PrintWriter pw = new PrintWriter(ostream);


          if( param instanceof DevilVariable)
          {
varName = ((DevilVariable)param).getVarName();
                    searchtable = true;

          }

          //String varName = ((DevilVariable)param).getVarName();
          //System.out.println("\n\n\t\t <INTERPRETER> printOut\n\n");
          if(param.typename() == "array")
          {
```

```
    // System.out.println("I got here searching \n"+varName+" size "+symbolTable.size());
        for (int i = 0; i<symbolTable.size();i++){

            if (((SymbolTableElement) symbolTable.get(i)).getName().equals(varName)){
            notfound = false;

            SymbolTableElement ste = (SymbolTableElement)symbolTable.get(i);
            DevilDataType dtp1 = ste.getValue();
            HashMap v =((HashMap)dtp1.getObj());

            Vector vb = (Vector)param.getObj();

                    DevilDataType dtp = (DevilDataType)vb.firstElement();

            value = ((DevilInteger) dtp).getValue();
            value = ((DevilInteger) dtp).getValue();

            DevilDataType dd = ((DevilDataType)v.get(new Integer(value)));

            if (dd.typename() == "String"){
                        System.out.println("Nowblocking the ip:
"+((DevilString)(param)).getValue()+"-.");
                 blockString.append("iptables -I INPUT -s ");
                 blockString.append(((DevilString)(param)).getValue());
                           blockString.append("-j DROP");
            }
            else if (dd.typename() == "Integer"){
                System.out.println("So the final output is => ="+((DevilInteger)v.get(new
Integer(value))).getValue());
                blockString.append(" ="+((DevilInteger)v.get(new
Integer(value))).getValue());
            }
```

```
                              else if (dd.typename() == "Double"){
                                    System.out.println("So the final output is =>
="+((DevilDouble)v.get(new Integer(value))).getValue());
                                    blockString.append(" ="+((DevilDouble)v.get(new


                        }
                         if(notfound==true)
                              System.out.println("DEVIL COMPILER GENERATED ERROR Not found
\n"+varName +" Looks like used directly");




            }
            else if (param.typename() == "String"){
               System.out.println("Nowblocking the ip: "+((DevilString)(param)).getValue()+"-.");
               blockString.append("iptables -I INPUT -s ");
               blockString.append(((DevilString)(param)).getValue());
                                                                                        ); } }

                                                                                     lue()


                                                                        ))).getVa


                                                                  r(value


                                                              Intege
               blockString.append(" -j DROP");
            }
            else if (param.typename() == "Integer"){
                System.out.println("So the final output in block is => ="+((DevilInteger)(param)).getValue());
                blockString.append(((DevilInteger)(param)).getValue());
            }
            else if (param.typename() == "Double"){
                System.out.println("So the final output in block is => ="+((DevilDouble)(param)).getValue());
                blockString.append(((DevilDouble)(param)).getValue());
            } else if (param.typename()==null) {


                for (int i = 0; i<symbolTable.size();i++){
                        if (((SymbolTableElement) symbolTable.get(i)).getName().equals(varName)){
                            //System.out.println(varName);
                                        if (((SymbolTableElement) symbolTable.get(i)).getType() == "Integer"){
                                //System.out.println("Integer");
                                if (((SymbolTableElement) symbolTable.get(i)).getValue() instanceof
DevilInteger){
                                   // DevilInteger dataType = new
DevilInteger((DevilInteger)((SymbolTableElement) symbolTable.get(i)).getValue());
                                    DevilDataType dtp =
(DevilDataType)((SymbolTableElement)symbolTable.get(i)).getValue();

                                    System.out.print("found in the symbol table object"+dtp);
                                                        System.out.print("found in the symbol table
value"+((DevilInteger)dtp).getValue());
                                    blockString.append(" = "+((DevilInteger)dtp).getValue());
                                                // System.out.print(((DevilInteger)dataType).getValue());
                                }
                            }
                            else if (((SymbolTableElement) symbolTable.get(i)).getType() == "Double"){
                                if (((SymbolTableElement) symbolTable.get(i)).getValue() instanceof
DevilDouble){
```

```
                                  // DevilInteger dataType = new
DevilInteger((DevilInteger)((SymbolTableElement) symbolTable.get(i)).getValue());
                                   DevilDataType dtp =
(DevilDataType)((SymbolTableElement)symbolTable.get(i)).getValue();

                                   System.out.print("found in the symbol table object"+dtp);
                                                   System.out.print("found in the symbol table
value"+((DevilDouble)dtp).getValue());
                                   blockString.append(" = "+((DevilDouble)dtp).getValue());
                                             // System.out.print(((DevilInteger)dataType).getValue());
                                }
                          }//end else if
                          else if (((SymbolTableElement) symbolTable.get(i)).getType() == "String"){
                                if (((SymbolTableElement) symbolTable.get(i)).getValue() instanceof DevilString){
                                  // DevilInteger dataType = new
DevilInteger((DevilInteger)((SymbolTableElement) symbolTable.get(i)).getValue());
                                   DevilDataType dtp =
(DevilDataType)((SymbolTableElement)symbolTable.get(i)).getValue();

                                   System.out.print("found in the symbol table object"+dtp);
                                                   System.out.print("found in the symbol table
value"+((DevilString)dtp).getValue());
                                             blockString.append(" = "+((DevilString)dtp).getValue());
                                             // System.out.print(((DevilInteger)dataType).getValue());
                                }
                          }//end else if
                      }
                  }


              }
```

```
        else {

                if ( param instanceof DevilInteger) {
                System.out.println(" printing =>"+((DevilInteger)param).getValue());
                } else if (param instanceof DevilString ) {
                                System.out.println(" printing =>"+((DevilString)param).getValue());
                } else if (param instanceof DevilDouble ){
                System.out.println(" printing =>"+((DevilDouble)param).getValue());
                }
                else {
                System.out.println("DEVIL COMPILER GENERATED ERROR : Unsupported or
Mismatched type ");
                }

        }




                                        System.out.println("\nCommand Generated is "+blockString.toString());




                                        public void unblock(DevilDataType a)
                                        {

                                        }


                                        public void allow(DevilDataType a)
```

```
"+blockCommand);
ex );
```

```
{


}



public DevilDataType searchGlobalSymbol(DevilDataType dtp)
{
        String varName=null;
        boolean doSearch = false;
        if(dtp instanceof DevilVariable)
        {

                varName = ((DevilVariable)dtp).getVarName();
                doSearch = true;
        }

        if(doSearch == true) {
        for (int i= 0; i<symbolTable.size();i++)
        {
          if (((SymbolTableElement) symbolTable.get(i)).getName().equals(varName))
         {
                                SymbolTableElement ste =((SymbolTableElement)symbolTable.get(i));
                  DevilDataType dtp1 = ste.getValue();
                System.out.println(" Found inn symbol table "+ dtp1);
                return dtp1;
         }
        }
        }
        System.out.println(" Reached End of search and not found \n"+dtp);
        return null;

    }


    public void call_func(String name,Vector params)
    {
        DevilDataType dtp_decl = null;
        DevilDataType dtp_passed = null;
        DevilDataType dtp = null;
        DevilDataType dtp1 = null;
        DevilDataType dtp_f = null;
        Vector arg_passed = null;
        try {

        //System.out.println(" Invoking the function\n"+name);
        //System.out.println("Invoking "+ name+" param list size is "+params.size());

        if(params.size()>0){
            arg_passed = new Vector();
            //dtp_passed = new Vector();
        }


        // the variable list to be passed is read b4 adding the symbol table for the function
        for(int i=0;i<params.size();i++)
        {

            dtp = (DevilDataType)params.get(i);

            //dtp_passed.add(dtp);

            dtp_f = getDevilData(dtp);
```

```
                        arg_passed.add(dtp_f);

                        //System.out.println("\n param is "+params.get(i)+" and mapped Devildata from "+dtp_f);
}

// get the function from function table
DevilFunction functorun = (DevilFunction)functiontable.get(name);

if(functorun.args.size()!=params.size())
{
            System.out.println("Compiler error, Mismathced params in declaration and invokation\n");
            return;
}

// initialization function
// creates func [decl, passed] mapping
// adds symbol table
record.symTableInit(name);


// add args to symbol table
Vector arg_decl = functorun.getArgs();

for(int j=0; j< arg_decl.size();j++)
{
            // arguments from decl list
            dtp_decl =(DevilDataType)arg_decl.get(j);

            dtp_passed = (DevilDataType)params.get(j);


            String decl_var = ((DevilVariable)dtp_decl).getVarName();
            if(dtp_passed instanceof DevilVariable)
            {
                                            String passed_var = ((DevilVariable)dtp_passed).getVarName();

                dtp_passed = (DevilDataType)arg_passed.get(j);

                System.out.println(" Decl List "+decl_var +" Param List= " +passed_var);

                record.addTosymTable(name,dtp_decl,decl_var,dtp_passed,passed_var);

            } else {

                dtp_passed = (DevilDataType)arg_passed.get(j);

                                            System.out.println(" Decl List "+decl_var +" Param List " +null);

                                            record.addTosymTable(name,dtp_decl,decl_var,dtp_passed,null);
                // name is null
                // pass by value

            }

            // added argument to symbol table
            // inited passed values in arg list

        }
        functorun.walker.expr(functorun.funcbody);

            System.out.println("\n Devildatatype function execution completed "+functorun);
        record.removeSymTable();

        // now this need to replace parameter names in function invocation

        } catch (Exception ec)
        {
                ec.printStackTrace();
        }

}


public DevilDataType getDevilDataType(String data)
{
        int a;
        double b;
        String c= null;
        DevilDataType dtp = null;
        dtp = getNumber(data);
        if(dtp==null)
        {

                dtp =(DevilDataType) new DevilVariable(data);

        }


        return dtp;




}
```

```
public void new_func(DevilWalker walker, String name, Vector args, AST body ) {

        try {
        //System.out.println("Name of Function \n"+ name);
        //for(int i=0;i<args.size();i++)
        //System.out.println("Arg list \n"+ (DevilDataType)args.get(i));

        //System.out.println("Body of Function \n"+ body);

        //System.out.println("Number of children of statement"+body.getNumberOfChildren());

        //walker.expr(body);


        //System.out.println("Number of children of statement"+body.getNumberOfChildren());
        //walker.expr(body);



      functiontable.put( name, new DevilFunction( walker, name, args, body, functiontable ) );

        //System.out.println("Stored Func in FunctionTable\n");
        } catch (Exception ex)
        {
                ex.printStackTrace();
        }
   }


public String[] VectoString( Vector vec ) {
        //System.out.println("Called Vectorto String"+vec);
      String[] sarray = new String[ vec.size() ];
      for ( int i=0; i<vec.size(); i++ ){
        sarray[i] = (String) vec.elementAt( i );
           //System.out.println("\nNext elemnt ="+sarray[i]);
         }

      return sarray;
    }



public void printOutput(DevilDataType out)
{

        boolean searchtable = false;
        DevilDataType dd = null;
        String varName="";
        try {
        if( out instanceof DevilVariable)
        {
                varName = ((DevilVariable)out).getVarName();
                searchtable = true;
                dd = record.lookup(varName);


        } else {

        dd = out;
        }
        if(dd==null)
        {
         dd =out;
        }
        System.out.println("Inside print Out");
```

```
            if ( dd instanceof DevilInteger) {
                        System.out.println(" printing =>"+((DevilInteger)dd).getValue());
            } else if (dd instanceof DevilString ) {
                        System.out.println(" printing =>"+((DevilString)dd).getValue());

            } else if (dd instanceof DevilDouble ){
                        System.out.println(" printing =>"+((DevilDouble)dd).getValue());

            } else if (dd.name=="array" )
            {
//System.out.println("Type is Array"+out);


                        Vector v = (Vector)dd.getObj();


            }
            else {
                        System.out.println("DEVIL COMPILER GENERATED ERROR : Unsupported or
Mismatched type ");

            }
            } catch (Exception e)
            {
                        e.printStackTrace();
            }

            //return;

}

}
```

DevilDataType.java

```
/*                              DevilDataType    */

public class DevilDataType {
        public String name;
        public Object value; // drake
            public DevilDataType(){
                    name = null;
                    value = null;
        }
            public DevilDataType(String name){
            this.name = name;
            value= null;
            }
            public DevilDataType(DevilDataType ddt){
                                                            this.name = ddt.name;
            value= null;
            }

            public Object getObj() {
             return this.value;
            }
            public void setObj(Object obj) {
            value = obj;

            }
            /* Parsing the string type and return correct DevilDataType*/
            public static DevilDataType parse(DevilString dstr){
    DevilInteger b1 = null;
    DevilDouble b2 = null;
    String str = dstr.getValue();
    double tempval = 0.0 ; // initialize to 0.0
    int i = str.length()-1;

    if(str.length() == 0) {
            b2 = new DevilDouble(tempval);
            return b2;
    }
    while (i>0) {
            try {
```

```
                                                tempval = Double.parseDouble(str);
                                if(str.indexOf(".") == -1 && str.indexOf("e") ==-1 && str.indexOf("E")==-
1){
                                                b1 = new DevilInteger((int)tempval);
                                        return b1;
                                }
                                else {
                                        b2 = new DevilDouble(tempval);
                                        return b2;
                                }

                        }
                        catch (Exception e){
                                str = str.substring(0,i);
                                i--;
                        }
                }
                b2 = new DevilDouble(tempval);
                return b2;
        }

        /* type name */
        public String typename(){
                return this.name;
        }

        public DevilDataType copy() {
                return new DevilDataType();
        }

        public void setType(String name){
                this.name = name;
        }

        public DevilDataType error (String msg) {
                throw new DevilException ("illegal operation: " +msg
.+ "( <" +typename() + ">"
.+ (name != null ? name : "<?>")
.+ ")"

                                                );
        }

        public DevilDataType error(DevilDataType b, String msg ){
                if (b==null ) return error(msg);
                throw new DevilException ("illegal operation: " + msg + "( <" + typename() + "> " + ( name
!= null ? name : "<?>" ) + " and " + "<" + typename() + "> "
     + ( name != null ? name : "<?>" ) + " )" );

        }

        public DevilDataType plus(DevilDataType b) {
                return error(b , "+");

        }
}
```

DevilDouble.java

```
/* DevilDouble */

import java.util.*;
public class DevilDouble extends DevilDataType{
	double doubleNumber;

	public DevilDouble(double val){
		doubleNumber = val;
	}

	public DevilDouble(DevilDouble val){
		doubleNumber = val.getValue();
	}
	public String typename(){
		this.name = "Double";
		return "Double";
	}

	public double getValue(){
		return doubleNumber;
	}

	public void setValue(double val){
		doubleNumber = val;
	}

} /* DevilException */
class DevilException extends RuntimeException {
	DevilException( String msg ) {
		super(msg);
	}
}



/* DevilInterger */
import java.util.*;
public class DevilInteger extends DevilDataType{
	int intNumber;

	public DevilInteger(int val){
		intNumber = val;
	}

	public DevilInteger(DevilInteger xint){
		intNumber = ((DevilInteger)xint).getValue();
	}

	public int getValue(){
		return intNumber;
	}

	public String typename(){
		return "Integer";
	}

	public void setValue(int val){
		intNumber = val;
	}
}
```

```
/* DevilVarable */

public class DevilVariable extends DevilDataType{
	String varName;

	public DevilVariable(String var){
		varName = var;
	}

	public String getVarName(){
		return varName;
	}
}


public class DevilString extends DevilDataType{
	String string;
	public DevilString(String s){
		string = s;
}

	public DevilString(DevilString s){
		string = s.getValue();
	}

							public String typename(){
		this.name = "String";
		return "String";
	}

							public String getValue(){
	return string;
```

```
          }
          public void setValue(String val){
                    string = val;
                    }
}
```

DevilFunction.java

```
import java.io.*;
import java.util.*;
import antlr.collections.AST;

class DevilFunction extends DevilDataType {
    Vector args;
    AST funcbody;
   HashMap tree;
   public DevilWalker walker;
   int id;

   public DevilFunction( DevilWalker walker , String fname, Vector args,
                AST funcbody, HashMap functionSymbolTable) {
      super( fname );
      this.args = args;
      this.funcbody = funcbody;
      this.tree = functionSymbolTable;
          this.walker = walker;
    }
    public DevilFunction( String fname, int id ) {
      super( fname );
      this.args = null;
      this.id = id;
      tree = null;
      funcbody = null;
    }

    public final boolean isInternal() {
      return funcbody == null;
```

```
}

public final int getInternalId() {
  return id;
}

public String typename() {
  return "function";
}


public Vector getArgs() {
  return args;
}

public HashMap getFunctionSymbolTable() {
  return tree;
}

          public AST getfuncBody() {
rn funcbody;

DevilInterpreter.java

block impemented 1.07 am*/

import java.util.*;
ort java.io.*;

          import antlr.CommonAST;
          import antlr.collections.AST;
          import antlr.RecognitionException;
                                     import antlr.TokenStreamException;
                                    import antlr.TokenStreamIOException;
          public class DevilInterpreter {
           private DevilDataType xdt;
           public LinkedList symbolTable = new LinkedList();
           public FileOutputStream out; // declare a file output object
           public PrintStream p; // declare a print stream object
```

```
 public boolean firsttime = false;

public StringBuffer FileString; // to write on the .sh file
 public StringBuffer blockCommand = new StringBuffer("block ");

 public static HashMap functiontable = new HashMap();

 public DevilStack record = new DevilStack(symbolTable);
 public DevilInterpreter() {
        try{
                xdt = new DevilDataType();
                                                        out = new FileOutputStream("myfile.txt");
                                                        // Connect print stream to the output stream
                p = new PrintStream( out );
        } catch(Exception e){
                e.printStackTrace();
        }

 }

 public DevilInterpreter(String name) {
        try {
        xdt = new DevilDataType(name);

        out = new FileOutputStream("outputFile.sh");

        // Connect print stream to the output stream
        p = new PrintStream( out );
        } catch(Exception e){
                e.printStackTrace();
        }
 }
```

```
    public DevilDataType allocate(DevilDataType a, DevilDataType b) {


// System.out.println("Interpreter called Allocate \n");
          int i = ((DevilInteger)b).getValue();

          Vector vb = new Vector();
          vb.add(0,b);

          // System.out.println("Name is "+((DevilVariable) a).getVarName());

          a.name ="array";
          a.setObj(vb);

          return a;
          // return dtp;
    } public DevilDataType assign(DevilDataType a, DevilDataType b) {
          DevilDataType dtpa = null;
          boolean var =false;
          boolean error =false;
          boolean added = false;
          int value1 =0;
          String value2=null;
          double value3 =0;
          String type = null;
          DevilString init_str = new DevilString("");
          DevilInteger b1 = null;
          DevilDouble b2 = null;
          DevilString b3 = null;
// System.out.println(" Assign called \n");
          String varName = null;
          DevilDataType dtp = null;
```

```
        System.out.println("A = >"+a);
        System.out.println("B = >"+b);
        try {

        if(b instanceof DevilVariable)
        {
                // get b from Symbol table

                dtp =record.lookup(((DevilVariable)b).getVarName());
                System.out.println("Look up in symbol table for =>"+((DevilVariable)b).getVarName() +"
returned "+ dtp+"\n");

                // if not found in symbol table , give error
                if( dtp ==null)
                {
                        System.out.println("Compilation error occurred, Right hand variable not
previously assigned,\n");
                        System.out.println("Compilation error occurred, Right hand variable not in symbol
table\n");
                        error = true;
                } else {


                        if (dtp instanceof DevilInteger)
                        {
                        type= "Integer";
                        value1 = ((DevilInteger)dtp).getValue();
                        } else if (dtp instanceof DevilDouble )
                        {
                        type= "Double";
                        value3 = ((DevilDouble)dtp).getValue();
```

```
                    } else if (dtp instanceof DevilString)
                    {
                    type= "String";
                    value2 = ((DevilString)dtp).getValue();
                    } else {
                    error = true;
                    System.out.println("Compilation error occurred, Unknown data type\n");
                    }


            }
 }
 else { // b is not variable but an atom
// get type and

if (b instanceof DevilInteger)
{
        type= "Integer";
        value1 = ((DevilInteger)b).getValue();
} else if (b instanceof DevilDouble )
{
        type= "Double";
        value3 = ((DevilDouble)b).getValue();
} else if (b instanceof DevilString)
{
        type= "String";
        value2 = ((DevilString)b).getValue();
}
else {
        error = true;
```

```
                System.out.println("Compilation error occurred, Unknown data type\n");
        }

        // get value

}

if(error!=true)
{
        if(a instanceof DevilVariable)
        {
        // check if a in symbol table or not.

                if ((a.name!=null)&& (a.name.equals("array")))
                {
                                                Vector vb = (Vector)a.getObj();

                        DevilDataType dtp1 = (DevilDataType)vb.firstElement();


                         varName =((DevilVariable)a).getVarName();

                        DevilDataType ddtp = record.lookup(varName);
                        if(ddtp==null)
                        {
                                // not found in symbol table
                                HashMap hm = new HashMap();
                                Vector v =(Vector)a.getObj();
                                v.add(hm);
```

```
                    ddtp =a;
                }

                                           Vector v =((Vector)ddtp.getObj());
                                          HashMap hv = (HashMap)v.get(1);

                hv.put(new Integer(((DevilInteger)dtp1).getValue()),b);


                ddtp.setObj(v);
                                            record.update(varName,ddtp);
} else {


                varName = ((DevilVariable)a).getVarName();
                 dtpa =record.lookup(varName);
                                     System.out.println("Look up in symbol table for
"+((DevilVariable)a).getVarName() +" returned "+ dtpa+"\n");

                if(dtpa==null)
                {
                    // dtpa doesn exist in symbol table need to add it
                        // Update with new Object
                    if(type=="String")
                                           dtpa = new DevilString(value2);
                    else if(type=="Integer")
                                          dtpa = new DevilInteger(value1);
                    else if(type=="Double")
                                          dtpa = new DevilDouble(value3);

                                            record.update(varName,dtpa);
```

```
            }
             else {
            if(dtpa.typename()==type){
                        if(type=="String")
                                                ((DevilString)dtpa).setValue(value2);
                        else if(type=="Integer")
                                               ((DevilInteger)dtpa).setValue(value1);
                        else if(type=="Double")
                                               ((DevilDouble)dtpa).setValue(value3);

            } else {
                        // Update with new Object
                        if(type=="String")
                                                   dtpa = new DevilString(value2);
                        else if(type=="Integer")
                                                  dtpa = new DevilInteger(value1);
                        else if(type=="Double")
                                                  dtpa = new DevilDouble(value3);

                                                    record.update(varName,dtpa);

            }
            }

// update a with b's value
// if Object Type are different replace it by creating anew one




            }
} else {
```

```
                    // wrong l value;
                              System.out.println("COMPILER ERROR :Left side of assignment not a variable
but a constant\n");
                    }
          } else {

                    System.out.println("COMPILER ERROR : Error occurred in getting right side of
assignment\n");

          }







}
catch( Exception ex)
    {
          ex.printStackTrace();
    }

          return dtpa;
}
    public DevilDataType assignplus(DevilDataType a, DevilDataType b) {
          DevilDataType aux = null;
          aux = plus(a,b);
          return assign(a, aux);
    }
    public DevilDataType assignmult(DevilDataType a, DevilDataType b) {
          DevilDataType aux = null;
          aux = star(a,b);
          return assign(a, aux);
     }
     public DevilDataType assigndiv(DevilDataType a, DevilDataType b) {
          DevilDataType aux = null;
          aux = div(a,b);
          return assign(a, aux);
     }
     public DevilDataType assignminus(DevilDataType a, DevilDataType b) {
          DevilDataType aux = null;
          aux = dash(a,b);
          return assign(a, aux);
     }

     public DevilDataType getDevilData(DevilDataType a)
     {
               boolean error = false;
               DevilDataType valueA =null;
               DevilDataType dta = null;
               String varNameA = null;
               String typeA = null;
               if( a instanceof DevilVariable )
               {
                    // look up in symbl table
                    varNameA = ((DevilVariable)a).getVarName();
                    dta = record.lookup(varNameA);
                    if(dta==null)
                    {
                              System.out.println("Compiler Error, Variable "+ varNameA + " not assigned or
inited\n");
```

```
                        } else {
if (dta instanceof DevilInteger)
                            {
                                typeA= "Integer";
                                valueA = ((DevilInteger)dta);
                            } else if (dta instanceof DevilDouble )
                            {
                                typeA= "Double";
                                valueA = ((DevilDouble)dta);
                            } else if (dta instanceof DevilString)
                            {
                                typeA= "String";
                                valueA = ((DevilString)dta);
                            }
                            else {
                                error = true;
                                System.out.println("Compilation error occurred, Unknown data type\n");
                                valueA = null;
                            }

                      }
                } else {
                    dta =a;

                            if (dta instanceof DevilInteger)
                            {
                                typeA= "Integer";
                                valueA = ((DevilInteger)dta);
                            } else if (dta instanceof DevilDouble )
                            {
                                typeA= "Double";
                        valueA = ((DevilDouble)dta);
                } else if (dta instanceof DevilString)
                {
                        typeA= "String";
                        valueA = ((DevilString)dta);
                }
                else {
                        error = true;
                        System.out.println("Compilation error occurred, Unknown data type\n");
                        valueA = null;
                }


        }


    return valueA;

}

public DevilDataType plus(DevilDataType a, DevilDataType b){
    String returnString = new String();
    DevilString init_str = new DevilString("");

    DevilDataType dta = null;
    DevilDataType dtb = null;


    DevilString xs = null; DevilInteger xi = null;
      DevilDouble xd = null;
```

```
        String s1 = new String(), s2 = new String();
        double d1 = 0, d2 = 0, returnDouble;
        int i1 = 0, i2 = 0, returnInt;
        boolean var1 = false, var2 = false;

        try {


            DevilDataType valueA =null;
            DevilDataType valueB =null;
            String varNameA = null;
            String typeA = null;
            String varNameB = null;
            String typeB = null;

            valueA = getDevilData(a);
            valueB = getDevilData(b);



            if (valueA.typename() == null && valueB.typename() == null) // empty str + empty str = empty str,
so we simply return a
                    { //System.out.println("<PLUS> returning ");
                        return a;

                    }

            if (valueA.typename() == valueB.typename()){
                    if(valueA.typename() == "Integer"){
                            returnInt = ((DevilInteger)valueA).getValue()+ ((DevilInteger)valueB).getValue();
                            xi = new DevilInteger(returnInt);
                            xi.setType("Integer");
                        return xi;
                    }
                    else {
                              if(valueA.typename() == "Double"){
                                        returnDouble = ((DevilDouble)valueA).getValue()+
((DevilDouble)valueB).getValue();
                                                                xd = new DevilDouble(returnDouble);
                                        xd.setType("Double");
                                        return xd;
                              }
                  else{
                                                                  if(valueA.typename() == "String"){
                                                                            return concat(a,b);
                                        }
                              }
                    }
              }
             else if(valueA.typename()!=valueB.typename())
             {
                    if ((valueA.typename()=="String") || valueB.typename()!="String"){
                              return concat(a,b);
                    }

                                           if ((valueA.typename()!="String")&&(valueB.typename()!="String"))
                    {
                              if(valueA.typename()=="Integer")
                              {
                                                                d1 =((DevilInteger)valueA).getValue();
} else {
                                     d1 =((DevilDouble)valueA).getValue();
                           }
```

```
                                        if(valueB.typename()=="Integer")
                    {
                        d2 =((DevilInteger)valueB).getValue();
                    } else {
                        d2 =((DevilDouble)valueB).getValue();
                    }

                    double retval = d1 + d2;
                                         xd = new DevilDouble(retval);
                    xd.setType("Double");
                    return xd;


                }

           } else {

                System.out.println("Compiler error Mismathced data types\n");
                }

          }
          catch(Exception ex)
             {
                System.out.println("\nException occurred in Interpreter PLUS"+ex);
                ex.printStackTrace();
             }
          return null;
       }
       /**
        *
       .* @param a
       . * @param b
```

```
. * @return

 */
public DevilDataType concat(DevilDataType a, DevilDataType b){
        //TODO: Concatenation
        String returnString = new String();
        DevilString init_str = new DevilString("");
        DevilString xs = null;
        DevilInteger xi = null;
        DevilDouble xd = null;
        DevilString sd1 = null;
        DevilString sd2 = null;
        DevilString sd3 = null;
        String s1 = new String(), s2 = new String();
        double d1 = 0, d2 = 0, returnDouble;
        int i1 = 0, i2 = 0, returnInt;
        boolean var1 = false, var2 = false;
        String ConcatS = new String();
        String stringvalue = new String();
        String stringvaluedouble = new String();
        String stringvaluetwo = new String();
        String stringvaluedoubletwo = new String();

        try {
            DevilDataType valueA =null;
            DevilDataType valueB =null;
            String varNameA = null;
            String typeA = null;  String varNameB = null;
            String typeB = null;

            valueA = getDevilData(a);
```

```
valueB = getDevilData(b);

if(valueA.typename() == "String" && valueB.typename() == "String"){
	ConcatS = ((DevilString)valueA).getValue() + ((DevilString)valueB).getValue();
	sd1 = new DevilString(ConcatS);
	sd1.setType("String");
	return sd1;
}
else{
	if(valueA.typename() == "Integer"){
		ConcatS = ((DevilInteger)valueA).getValue() + ((DevilString)valueB).getValue();
		sd1 = new DevilString(ConcatS);
		sd1.setType("String");
		return sd1;
	}
	if(valueB.typename() == "Integer"){
		ConcatS = ((DevilString)valueA).getValue() + ((DevilInteger)valueB).getValue();
		sd1 = new DevilString(ConcatS);
		sd1.setType("String");
		return sd1;
	}
	if(valueA.typename() == "Double"){
		ConcatS = ((DevilDouble)valueA).getValue() + ((DevilString)valueB).getValue();
		sd1 = new DevilString(ConcatS);
		sd1.setType("String");
		return sd1;
	}
	if(valueB.typename() == "Double"){
		ConcatS = ((DevilString)valueA).getValue() + ((DevilDouble)valueB).getValue();
		sd1 = new DevilString(ConcatS);
```

```
                        sd1.setType("String");
                        return sd1;
                    }
                }
            }
            catch(Exception ex)
                {
                    System.out.println("\nException occurred in Interpreter PLUS"+ex);
                    ex.printStackTrace();
                }
            return null;

    }
    /**
     *
    .* @param a
    . * @param b
    . * @return

     */
    public DevilDataType dash(DevilDataType a, DevilDataType b){
            String returnString = new String();
            DevilString init_str = new DevilString("");
            DevilString xs = null;
            DevilInteger xi = null;
            DevilDouble xd = null;
            String s1 = new String(), s2 = new String();
            double d1 = 0, d2 = 0, returnDouble;
            int i1 = 0, i2 = 0, returnInt; boolean var1 = false, var2 = false;
```

```
        try {
            DevilDataType valueA =null;
            DevilDataType valueB =null;
            String varNameA = null;
            String typeA = null;
            String varNameB = null;
            String typeB = null;

            valueA = getDevilData(a);
            valueB = getDevilData(b);


            if (valueA.typename() == null && valueB.typename() == null) // empty str + empty str = empty str,
so we simply return a
                  { //System.out.println("<PLUS> returning ");
                     return a;

                  }

            if (valueA.typename() == valueB.typename()){
                  if(valueA.typename() == "Integer"){
                        returnInt = ((DevilInteger)valueA).getValue()- ((DevilInteger)valueB).getValue();
                        xi = new DevilInteger(returnInt);
                        xi.setType("Integer");
                     return xi;
                  }
                  else if(valueB.typename() == "Double"){
                                          returnDouble = ((DevilDouble)valueA).getValue()-
((DevilDouble)valueB).getValue();
                        xd = new DevilDouble(returnDouble);
                        xd.setType("Double");
```

```
                    return xd;
                }
            }

            else if(valueA.typename()!=valueB.typename())
            {
                                if ((valueA.typename()!="String")&&(valueB.typename()!="String"))
                {
                    if(valueA.typename()=="Integer")
                    {
                                        d1 =((DevilInteger)valueA).getValue();
                    } else {
                                        d1 =((DevilDouble)valueA).getValue();
                    }
                    if(valueB.typename()=="Integer")
                    {
                                        d2 =((DevilInteger)valueB).getValue();
                    } else {
                                        d2 =((DevilDouble)valueB).getValue();
                    }

                    double retval = d1 -d2;
                    xd = new DevilDouble(retval);
                    xd.setType("Double");
                    return xd;
        }

            } else {

                System.out.println("Compiler error Mismathced data types\n");
                }
        }
        catch(Exception ex)
            {
                                                    System.out.println("\nException occurred in Interpreter PLUS"+ex);
                    ex.printStackTrace();
            }
        return null;
    }
    /**
     *
.* @param a
. * @param b
. * @return

     */
    public DevilDataType div(DevilDataType a, DevilDataType b){
        String returnString = new String();
        DevilString init_str = new DevilString("");
        DevilString xs = null;
        DevilInteger xi = null;
        DevilDouble xd = null;
        String s1 = new String(), s2 = new String();
        double d1 = 0, d2 = 0, returnDouble;
        int i1 = 0, i2 = 0, returnInt;
        boolean var1 = false, var2 = false;


        try {
            DevilDataType valueA =null;
            DevilDataType valueB =null;
            String varNameA = null;
```

```
                String typeA = null;
                String varNameB = null;
                String typeB = null;

                valueA = getDevilData(a);
                valueB = getDevilData(b);



                if (valueA.typename() == null && valueB.typename() == null) // empty str + empty str = empty str,
so we simply return a
                        { //System.out.println("<PLUS> returning ");
                            return a;

                        }

                if (valueA.typename() == valueB.typename()){
                        if(valueA.typename() == "Integer"){
                                    returnInt = ((DevilInteger)valueA).getValue()/ ((DevilInteger)valueB).getValue();
                                    xi = new DevilInteger(returnInt);
                                    xi.setType("Integer");
                            return xi;
                        }
                        else if(valueB.typename() == "Double"){
returnDouble = ((DevilDouble)valueA).getValue()/
((DevilDouble)valueB).getValue();
                                    xd = new DevilDouble(returnDouble);
                                    xd.setType("Double");
                                    return xd;
                        }
                }
```

```
   else if(valueA.typename()!=valueB.typename())
   {
          if ((valueA.typename()!="String")&&(valueB.typename()!="String"))
          {
                                                   if(valueA.typename()=="Integer")
                    {
                              d1 =((DevilInteger)valueA).getValue();
                    } else {
                              d1 =((DevilDouble)valueA).getValue();
                    }
                                                   if(valueB.typename()=="Integer")
                    {
                              d2 =((DevilInteger)valueB).getValue();
                    } else {
                              d2 =((DevilDouble)valueB).getValue();
                    }

                    double retval = d1 / d2;
                                                     xd = new DevilDouble(retval);
                    xd.setType("Double");
                    return xd;


          }

   } else {

          System.out.println("Compiler error Mismathced data types\n");
          }

}
catch(Exception ex)
```

```
            {
                System.out.println("\nException occurred in Interpreter PLUS"+ex);
                ex.printStackTrace();
            }
        return null;
    }
    /**
     *
 .* @param a
 . * @param b
 . * @return

     */
    public DevilDataType star(DevilDataType a, DevilDataType b){
        String returnString = new String();
        DevilString init_str = new DevilString("");
        DevilString xs = null;
        DevilInteger xi = null;
        DevilDouble xd = null;
        String s1 = new String(), s2 = new String();
        double d1 = 0, d2 = 0, returnDouble;
        int i1 = 0, i2 = 0, returnInt;
        boolean var1 = false, var2 = false;
        try {
            DevilDataType valueA =null;
            DevilDataType valueB =null;
            String varNameA = null;
            String typeA = null;
            String varNameB = null;
            String typeB = null;
```

```
                valueA = getDevilData(a);
                valueB = getDevilData(b);



                if (valueA.typename() == null && valueB.typename() == null) // empty str + empty str = empty str,
so we simply return a
                        { //System.out.println("<PLUS> returning ");
                            return a;

                        }

                if (valueA.typename() == valueB.typename()){
                        if(valueA.typename() == "Integer"){
                                returnInt = ((DevilInteger)valueA).getValue()* ((DevilInteger)valueB).getValue();
                                xi = new DevilInteger(returnInt);
                                xi.setType("Integer");
                            return xi;
                        }
                        else if(valueB.typename() == "Double"){
                                                        returnDouble = ((DevilDouble)valueA).getValue()*
((DevilDouble)valueB).getValue();
                                xd = new DevilDouble(returnDouble);
                                xd.setType("Double");
                                return xd;
                        }
                }

               else if(valueA.typename()!=valueB.typename())
               {
                                                if ((valueA.typename()!="String")&&(valueB.typename()!="String"))
                        {
```

```
                if(valueA.typename()=="Integer")
                {
                                        d1 =((DevilInteger)valueA).getValue();
                } else {
                                        d1 =((DevilDouble)valueA).getValue();
                }
                if(valueB.typename()=="Integer")
                {
                                        d2 =((DevilInteger)valueB).getValue();
                } else {
                                        d2 =((DevilDouble)valueB).getValue();
                }

                double retval = d1 * d2;
                xd = new DevilDouble(retval);
                xd.setType("Double");
                return xd;


            }
     } else {

            System.out.println("Compiler error Mismathced data types\n");
            }

    }
    catch(Exception ex)
      {
                            System.out.println("\nException occurred in Interpreter PLUS"+ex);
            ex.printStackTrace();
      }
```

```
        return null;
    }
    /**
     *
.* @param a
. * @param b
. * @return

     */
    public DevilDataType modulus(DevilDataType a, DevilDataType b){
        DevilInteger xi = null;
        DevilDouble xd = null;
        double d1 = 0, d2 = 0, returnDouble;
        int i1 = 0, i2 = 0, returnInt;
        boolean var1 = false, var2 = false;

        try {
            DevilDataType valueA =null;
            DevilDataType valueB =null;
            String varNameA = null;
            String typeA = null;
            String varNameB = null;
            String typeB = null;

            valueA = getDevilData(a);
            valueB = getDevilData(b);


            if (valueA.typename() == null && valueB.typename() == null) // empty str + empty str = empty str,
so we simply return a
```

```
                    { //System.out.println("<PLUS> returning ");
                       return a;

                    }

                if (valueA.typename() == valueB.typename()){
                    if(valueA.typename() == "Integer"){
                            returnInt = ((DevilInteger)valueA).getValue()% ((DevilInteger)valueB).getValue();
                            xi = new DevilInteger(returnInt);
                            xi.setType("Integer");
                       return xi;
                    }
                    else if(valueB.typename() == "Double"){
                                                  returnDouble = ((DevilDouble)valueA).getValue()%
((DevilDouble)valueB).getValue();
                            xd = new DevilDouble(returnDouble);
                            xd.setType("Double");
                            return xd;
                    }
                }
 else if(valueA.typename()!=valueB.typename())
          {
                  if ((valueA.typename()!="String")&&(valueB.typename()!="String"))
                  {
                                                            if(valueA.typename()=="Integer")
                        {
                                d1 =((DevilInteger)valueA).getValue();
                        } else {
                                d1 =((DevilDouble)valueA).getValue();
                        }
                                                            if(valueB.typename()=="Integer")
                        {
                                d2 =((DevilInteger)valueB).getValue();
                        } else {
                                d2 =((DevilDouble)valueB).getValue();
                        }

                        double retval = d1 % d2;
                                                                xd = new DevilDouble(retval);
                        xd.setType("Double");
                        return xd;


                }

        } else {

                System.out.println("Compiler error Mismathced data types\n");
                }

        }
        catch(Exception ex)
            {
                System.out.println("\nException occurred in Interpreter PLUS"+ex);
                ex.printStackTrace();
            }
        return null;
  }
 /**
  *
 .* @param a
 . * @return
 */
public DevilDataType lincrement(DevilDataType a){
        // check if a is in the symbol table
        DevilDataType valueA =null;
        valueA = getDevilData(a);
                    if (valueA.typename() == "Integer"){
                        //set the value of a, but use the old value in the expression
                                                                    int aux = ((DevilInteger)valueA).getValue();
                        aux++;
                        ((DevilInteger)valueA).setValue(aux);
                        return valueA;
                    }
                    //TODO we have to check if we can do this on doubles too
                    else{
                    if(a.typename() == "Double"){
                        //set the value of a, but use the old value in the expression
                                                                 double aux = ((DevilDouble)valueA).getValue();
                        aux = aux++;
                        ((DevilDouble)valueA).setValue(aux);
return valueA;
                         }
            else
                     System.err.println("This operator can't be aplied on this type");
            }
                  return null;

    }
    /**
     *
 .* @param a
```

```
. * @return

     */
    public DevilDataType inc_stat(DevilDataType a){
        if(!(a instanceof DevilVariable)){
            System.err.println("ERROR: You can't apply this operator on a costant.");
        }

                for (int j= 0; j<symbolTable.size();j++){
                        if (((SymbolTableElement) symbolTable.get(j)).getName().equals(((DevilVariable)
a).getVarName())){
                if (((SymbolTableElement) symbolTable.get(j)).getValue() instanceof DevilInteger){
                    int aux = ((DevilInteger)((SymbolTableElement)
symbolTable.get(j)).getValue()).getValue();
                    ((DevilInteger)((SymbolTableElement) symbolTable.get(j)).getValue()).setValue(aux +
1);
                    return a;
                }//end if
                if (((SymbolTableElement) symbolTable.get(j)).getValue() instanceof DevilDouble){
                    DevilDouble xd = new DevilDouble(1);
                    xd.setType("Double");
                    return assignplus(a,xd);

                }//end if
                if (((SymbolTableElement) symbolTable.get(j)).getValue() instanceof DevilString){
                            System.err.println("ERROR: You can't apply this operator on a string.");
                }
            }//end if
        }//end for
        return null;
    }
    /**
     *
.* @param a
. * @return

     */
    public DevilDataType ldecrement(DevilDataType a){
        // check if a is in the symbol table
        DevilDataType valueA =null;
        valueA = getDevilData(a);
                if (valueA.typename() == "Integer"){
                    //set the value of a, but use the old value in the expression
                    int aux = ((DevilInteger)valueA).getValue();
                    aux--;
                    ((DevilInteger)valueA).setValue(aux);
                    return valueA;
                }
                //TODO we have to check if we can do this on doubles too
                else{
                if(a.typename() == "Double"){
                    //set the value of a, but use the old value in the expression
                    double aux = ((DevilDouble)valueA).getValue();
aux = aux--;
                                                        ((DevilDouble)valueA).setValue(aux);
                    return valueA;
                }
         else
               System.err.println("This operator can't be aplied on this type");
         }
              return null;
    }


  public DevilDataType rincrement(DevilDataType a){
// check if a is in the symbol table
          DevilDataType valueA =null;
          valueA = getDevilData(a);
                    if (valueA.typename() == "Integer"){
                        //set the value of a, but use the old value in the expression
                        int aux = ((DevilInteger)valueA).getValue();
                        aux++;
                                                            ((DevilInteger)valueA).setValue(aux);
                        return valueA;
                    }
                    //TODO we have to check if we can do this on doubles too
                    else{
                    if(a.typename() == "Double"){
                        //set the value of a, but use the old value in the expression
                        double aux = ((DevilDouble)valueA).getValue();
                        aux = aux++;
                                                            ((DevilDouble)valueA).setValue(aux);
                        return valueA;
                    }
           else
                  System.err.println("This operator can't be aplied on this type");
           }
                return null;
    }
```

```
          /**
     *
.* @param a
. * @return

     */
   public DevilDataType rdecrement(DevilDataType a){
         // check if a is in the symbol table
         DevilDataType valueA =null;
         valueA = getDevilData(a);
                    if (valueA.typename() == "Integer"){
                        //set the value of a, but use the old value in the expression
                        int aux = ((DevilInteger)valueA).getValue();
                        aux--;
                                                  ((DevilInteger)valueA).setValue(aux);
                        return valueA;
                    }
                    //TODO we have to check if we can do this on doubles too
                    else{
                    if(a.typename() == "Double"){
                        //set the value of a, but use the old value in the expression
                        double aux = ((DevilDouble)valueA).getValue();
                        aux = aux--;
                                                  ((DevilDouble)valueA).setValue(aux);
return valueA;
                    }
          else
                  System.err.println("This operator can't be aplied on this type");
          }
               return null;
```

```
    }
    /**
     *
.* @param a
. * @return

     */
    public DevilDataType dec_stat(DevilDataType a){

            if(!(a instanceof DevilVariable)){
                System.err.println("ERROR: You can't apply this operator on a constant.");
            }

            for (int j= 0; j<symbolTable.size();j++){
                                    if (((SymbolTableElement) symbolTable.get(j)).getName().equals(((DevilVariable)
a).getVarName())){
                        if (((SymbolTableElement) symbolTable.get(j)).getValue() instanceof DevilInteger){
                            int aux = ((DevilInteger)((SymbolTableElement)
symbolTable.get(j)).getValue()).getValue();
                            ((DevilInteger)((SymbolTableElement) symbolTable.get(j)).getValue()).setValue(aux -
1);
                            return a;
                        } //end if
                        if (((SymbolTableElement) symbolTable.get(j)).getValue() instanceof DevilDouble){
                            DevilDouble xd = new DevilDouble(1);
                            xd.setType("Double");
                            return assignminus(a,xd);

                        } //end if
                        if (((SymbolTableElement) symbolTable.get(j)).getValue() instanceof DevilString){
                                        System.err.println("ERROR: You can't apply this operator on a string.");
```

```
                    }
                } //end if
            } //end for
            return null;
    }
    /**
     *
.* @param a
. * @param b
. * @return

     */
    public DevilDataType equals(DevilDataType a, DevilDataType b){
            boolean returnVal = false;
            DevilString xs = null;
            DevilInteger xi = null;
            DevilDouble xd = null;
            String s1 = new String(), s2 = new String();
            double d1 = 0, d2 = 0;
            int i1 = 0, i2 = 0, returnInt;
            boolean var1 = false, var2 = false;

      try {
                DevilDataType valueA =null;
                DevilDataType valueB =null;
                String varNameA = null;
                String typeA = null;  String varNameB = null;
     String typeB = null;

     valueA = getDevilData(a);
     valueB = getDevilData(b);
```

```
        if(valueA.typename() == "String"){
                s1 = ((DevilString)valueA).getValue();
        }
        if(valueB.typename() == "String"){
                s2 = ((DevilString)valueB).getValue();
        }
        if(valueA.typename() == "Integer"){
                i1 = ((DevilInteger)valueA).getValue();
        }
if(valueB.typename() == "Integer"){
                i2 = ((DevilInteger)valueB).getValue();
 }
        if(valueA.typename() == "Double"){
                d1 = ((DevilDouble)valueA).getValue();
        }
if(valueB.typename() == "Double"){
                d2 = ((DevilDouble)valueB).getValue();
 }

        if(valueA.typename() == valueB.typename()){
                if(valueA.typename() == "String"){
                        returnVal = s1.equals(s2);

                                                }
                if(valueA.typename() == "Integer"){
                        returnVal = i1 == i2;
      }
                if(valueA.typename() == "Double"){
                        returnVal = d1 == d2;
              }
```

```
                    }
      if(valueA.typename() != valueB.typename()){
             if(valueA.typename() == "Double" && valueB.typename() == "Integer"){
             returnVal = d1 == i2;
            }
             if(valueA.typename() == "Integer" && valueB.typename() == "Integer"){
             returnVal = i1 == d2;
            }
      }

            if (returnVal){
                        xi = new DevilInteger(1);
                        xi.setType("Integer");
                        return xi;
            }
            else {
                        xi = new DevilInteger(0);
                        xi.setType("Integer");
                        return xi;
            }

  }
  catch(Exception ex)
      {
             System.out.println("\nException occurred in Interpreter PLUS"+ex);
             ex.printStackTrace();
 }
                                                              System.err.println("ERROR:Type mismatch");
            return null;
}
   /**
```

```
     *
.* @param a
. * @param b
. * @return

     */
    public DevilDataType ge(DevilDataType a, DevilDataType b){
          boolean returnVal = false;
          DevilString xs = null;
          DevilInteger xi = null;
          DevilDouble xd = null;
                                                  String s1 = new String(), s2 = new String();
          double d1 = 0, d2 = 0;
          int i1 = 0, i2 = 0, returnInt;
          boolean var1 = false, var2 = false;

          try {
             DevilDataType valueA =null;
             DevilDataType valueB =null;
             String varNameA = null;
             String typeA = null;
             String varNameB = null;
             String typeB = null;

             valueA = getDevilData(a);
             valueB = getDevilData(b);
             if(valueA.typename() == "String"){
                   s1 = ((DevilString)valueA).getValue();
             }
             if(valueB.typename() == "String"){
                   s2 = ((DevilString)valueB).getValue();
```

```
                }
                if(valueA.typename() == "Integer"){
                        i1 = ((DevilInteger)valueA).getValue();
                }
            if(valueB.typename() == "Integer"){
                        i2 = ((DevilInteger)valueB).getValue();
             }
                if(valueA.typename() == "Double"){
                        d1 = ((DevilDouble)valueA).getValue();
                }
            if(valueB.typename() == "Double"){
                        d2 = ((DevilDouble)valueB).getValue();
             }

                if(valueA.typename() == valueB.typename()){
                        if(valueA.typename() == "String"){
                                returnVal = s1.equals(s2);
                        }
                        if(valueA.typename() == "Integer"){
                                returnVal = i1 >= i2;
              }
                        if(valueA.typename() == "Double"){
                                returnVal = d1 >= d2;
                        }

                                }
                if(valueA.typename() != valueB.typename()){
if(valueA.typename() == "Double" && valueB.typename() == "Integer"){
                      returnVal = d1 >= i2;
                      }
                      if(valueA.typename() == "Integer" && valueB.typename() == "Double"){
                        returnVal = i1 >= d2;
                        }
                }

                        if (returnVal){
                                xi = new DevilInteger(1);
                                xi.setType("Integer");
                                return xi;
                        }
                        else {
                                xi = new DevilInteger(0);
                                xi.setType("Integer");
                                return xi;
                        }

            }
            catch(Exception ex)
                {
                        System.out.println("\nException occurred in Interpreter PLUS"+ex);
                        ex.printStackTrace();
                }
            System.err.println("ERROR:Type mismatch");
            return null;
    }

    /**
     *
    .* @param a
    . * @param b
    . * @return
```

```
 */
public DevilDataType ls(DevilDataType a, DevilDataType b){
        boolean returnVal = false;
        DevilString xs = null;
        DevilInteger xi = null;
        DevilDouble xd = null;

        String s1 = new String(), s2 = new String();
        double d1 = 0, d2 = 0;
        int i1 = 0, i2 = 0, returnInt;
        boolean var1 = false, var2 = false;

        try {
            DevilDataType valueA =null;
            DevilDataType valueB =null;
            String varNameA = null;
            String typeA = null;
            String varNameB = null;
            String typeB = null;

            valueA = getDevilData(a);
            valueB = getDevilData(b);
            if(valueA.typename() == "String"){
                    s1 = ((DevilString)valueA).getValue();
            }
            if(valueB.typename() == "String"){
                    s2 = ((DevilString)valueB).getValue();
            }
            if(valueA.typename() == "Integer"){
i1 = ((DevilInteger)valueA).getValue();
            }
```

```
if(valueB.typename() == "Integer"){
            i2 = ((DevilInteger)valueB).getValue();
 }
      if(valueA.typename() == "Double"){
            d1 = ((DevilDouble)valueA).getValue();
      }
if(valueB.typename() == "Double"){
            d2 = ((DevilDouble)valueB).getValue();
 }

      if(valueA.typename() == valueB.typename()){
            if(valueA.typename() == "String"){
                      returnVal = !(s1.equals(s2));
            }
            if(valueA.typename() == "Integer"){
                      returnVal = i1 < i2;
    }
            if(valueA.typename() == "Double"){
                      returnVal = d1 < d2;
            }

                      }
      if(valueA.typename() != valueB.typename()){
            if(valueA.typename() == "Double" && valueB.typename() == "Integer"){
            returnVal = d1 < i2;
            }
            if(valueA.typename() == "Integer" && valueB.typename() == "Double"){
            returnVal = i1 < d2;
            }
      }
```

```
                if (returnVal){
                        xi = new DevilInteger(1);
                        xi.setType("Integer");
                        return xi;
                }
                else {
                        xi = new DevilInteger(0);
                        xi.setType("Integer");
                        return xi;
                }

        }
        catch(Exception ex)
           {
                System.out.println("\nException occurred in Interpreter PLUS"+ex);
                ex.printStackTrace();
           }
        System.err.println("ERROR:Type mismatch");
        return null;
}

/**
 *
.* @param a
. * @param b
. * @return

 */
public DevilDataType le(DevilDataType a, DevilDataType b){
        boolean returnVal = false; DevilString xs = null;
 DevilInteger xi = null;
```

```
  DevilDouble xd = null;
  String s1 = new String(), s2 = new String();
  double d1 = 0, d2 = 0;
  int i1 = 0, i2 = 0, returnInt;
  boolean var1 = false, var2 = false;


  try {
      DevilDataType valueA =null;
      DevilDataType valueB =null;
      String varNameA = null;
      String typeA = null;
      String varNameB = null;
      String typeB = null;

      valueA = getDevilData(a);
      valueB = getDevilData(b);
      if(valueA.typename() == "String"){
            s1 = ((DevilString)valueA).getValue();
      }
      if(valueB.typename() == "String"){
            s2 = ((DevilString)valueB).getValue();
      }
      if(valueA.typename() == "Integer"){
            i1 = ((DevilInteger)valueA).getValue();
      }
if(valueB.typename() == "Integer"){
            i2 = ((DevilInteger)valueB).getValue();
 }
      if(valueA.typename() == "Double"){
            d1 = ((DevilDouble)valueA).getValue();
      }
if(valueB.typename() == "Double"){
            d2 = ((DevilDouble)valueB).getValue();
 }

      if(valueA.typename() == valueB.typename()){
            if(valueA.typename() == "String"){
                      returnVal = s1.equals(s2);
           }
            if(valueA.typename() == "Integer"){
                      returnVal = i1 <= i2;
    }
            if(valueA.typename() == "Double"){
                      returnVal = d1 <= d2;
           }

                      }
      if(valueA.typename() != valueB.typename()){
            if(valueA.typename() == "Double" && valueB.typename() == "Integer"){
            returnVal = d1 <= i2;
           }
            if(valueA.typename() == "Integer" && valueB.typename() == "Double"){
            returnVal = i1 <= d2;
           }
      }

           if (returnVal){
                      xi = new DevilInteger(1);
                      xi.setType("Integer");
return xi;
```

```
                }
                else {
                        xi = new DevilInteger(0);
                        xi.setType("Integer");
                        return xi;
                }

        }
        catch(Exception ex)
           {
                System.out.println("\nException occurred in Interpreter PLUS"+ex);
                ex.printStackTrace();
           }
        System.err.println("ERROR:Type mismatch");
        return null;
}
   /**
    *
.* @param a
. * @param b
. * @return

    */
   public DevilDataType gt(DevilDataType a, DevilDataType b){
        boolean returnVal = false;
        DevilString xs = null;
        DevilInteger xi = null;
        DevilDouble xd = null;
        String s1 = new String(), s2 = new String();
        double d1 = 0, d2 = 0;
```

```
 int i1 = 0, i2 = 0, returnInt;
 boolean var1 = false, var2 = false;


 try {
     DevilDataType valueA =null;
     DevilDataType valueB =null;
     String varNameA = null;
     String typeA = null;
     String varNameB = null;
     String typeB = null;

     valueA = getDevilData(a);
     valueB = getDevilData(b);
     if(valueA.typename() == "String"){
            s1 = ((DevilString)valueA).getValue();
     }
     if(valueB.typename() == "String"){
            s2 = ((DevilString)valueB).getValue();
     }
     if(valueA.typename() == "Integer"){
            i1 = ((DevilInteger)valueA).getValue();
     }
if(valueB.typename() == "Integer"){
            i2 = ((DevilInteger)valueB).getValue();
 }
     if(valueA.typename() == "Double"){
            d1 = ((DevilDouble)valueA).getValue();
     }
if(valueB.typename() == "Double"){
            d2 = ((DevilDouble)valueB).getValue();
```

```
          }
  if(valueA.typename() == valueB.typename()){
                  if(valueA.typename() == "String"){
                            returnVal = !(s1.equals(s2));
                  }
                  if(valueA.typename() == "Integer"){
                            returnVal = i1 > i2;
          }
                  if(valueA.typename() == "Double"){
                            returnVal = d1 > d2;
                  }

                            }
            if(valueA.typename() != valueB.typename()){
                  if(valueA.typename() == "Double" && valueB.typename() == "Integer"){
                  returnVal = d1 > i2;
                  }
                  if(valueA.typename() == "Integer" && valueB.typename() == "Double"){
                  returnVal = i1 > d2;
                  }
            }

                  if (returnVal){
                            xi = new DevilInteger(1);
                            xi.setType("Integer");
                            return xi;
                  }
                  else {
                            xi = new DevilInteger(0);
                            xi.setType("Integer");
                            return xi;
```

```
                }

        }
        catch(Exception ex)
           {
                System.out.println("\nException occurred in Interpreter PLUS"+ex);
                ex.printStackTrace();
           }
        System.err.println("ERROR:Type mismatch");
        return null;
 }

 /**
  *
 */
 public DevilDataType note(DevilDataType a, DevilDataType b){
        boolean returnVal = false;
        DevilString xs = null;
        DevilInteger xi = null;
        DevilDouble xd = null;
        String s1 = new String(), s2 = new String();
        double d1 = 0, d2 = 0;
        int i1 = 0, i2 = 0, returnInt;
        boolean var1 = false, var2 = false;


        try {
           DevilDataType valueA =null;
           DevilDataType valueB =null;
           String varNameA = null;
           String typeA = null;  String varNameB = null;
        String typeB = null;

        valueA = getDevilData(a);
        valueB = getDevilData(b);
        if(valueA.typename() == "String"){
                s1 = ((DevilString)valueA).getValue();
        }
        if(valueB.typename() == "String"){
                s2 = ((DevilString)valueB).getValue();
        }
        if(valueA.typename() == "Integer"){
                i1 = ((DevilInteger)valueA).getValue();
        }
if(valueB.typename() == "Integer"){
                i2 = ((DevilInteger)valueB).getValue();
 }
        if(valueA.typename() == "Double"){
                d1 = ((DevilDouble)valueA).getValue();
        }
if(valueB.typename() == "Double"){
                d2 = ((DevilDouble)valueB).getValue();
 }

        if(valueA.typename() == valueB.typename()){
                if(valueA.typename() == "String"){
                        returnVal = !(s1.equals(s2));
               }
                if(valueA.typename() == "Integer"){
                        returnVal = i1 != i2;
     }
                if(valueA.typename() == "Double"){
                        returnVal = d1 > d2;
            }

                        }
        if(valueA.typename() != valueB.typename()){
              if(valueA.typename() == "Double" && valueB.typename() == "Integer"){
              returnVal = d1 != i2;
            }
              if(valueA.typename() == "Integer" && valueB.typename() == "Double"){
              returnVal = i1 != d2;
            }
        }

            if (returnVal){
                        xi = new DevilInteger(1);
                        xi.setType("Integer");
                        return xi;
            }
            else {
                        xi = new DevilInteger(0);
                        xi.setType("Integer");
                        return xi;
            }

    }
    catch(Exception ex)
        {
              System.out.println("\nException occurred in Interpreter PLUS"+ex);
              ex.printStackTrace();
        }
  System.err.println("ERROR:Type mismatch");
```

```
            return null;
    }
    /**
     *
□.* @param a
□. * @return

     */
    public DevilDataType logicalnot(DevilDataType a) {
            DevilInteger xi = null;
            int val_a, val_b;

            if (a instanceof DevilInteger){
                val_a = ((DevilInteger)a).getValue();
                if (val_a == 1){
                        xi = new DevilInteger(0);
                        xi.setType("Integer");
                        return xi;
                }
                else {
                        xi = new DevilInteger(1);
                        xi.setType("Integer");
                        return xi;
                }
            }
            else if (a instanceof DevilVariable)
                for (int j= 0; j<symbolTable.size();j++){
                        if (((SymbolTableElement) symbolTable.get(j)).getName().equals(((DevilVariable)
a).getVarName())){
                            a.setType(((SymbolTableElement)symbolTable.get(j)).getType());
                        if (a.typename() == "Integer"){
                            val_a =
((DevilInteger)((SymbolTableElement)symbolTable.get(j)).getValue()).getValue();
                            if (val_a == 0){
                                xi = new DevilInteger(1);
                                xi.setType("Integer");
                                return xi;
                            }
                            else{
                                xi = new DevilInteger(0);
                                xi.setType("Integer");
                                return xi;
                            }
                        }
                                                else throw new DevilException("Not the correct type");
                    }

                }
            return null;

    }

    /**
     *
.* @param a
. * @param b
. * @return

     */
    public DevilDataType and(DevilDataType a, DevilDataType b) {
            DevilInteger xi = null;
            int val_a, val_b; if (a instanceof DevilInteger){
                val_a = ((DevilInteger)a).getValue();
                if (val_a == 0){
                        xi = new DevilInteger(0);
                        xi.setType("Integer");
                        return xi;
                }
            }
            else if (a instanceof DevilVariable)
                for (int j= 0; j<symbolTable.size();j++){
                        if (((SymbolTableElement) symbolTable.get(j)).getName().equals(((DevilVariable)
a).getVarName())){
                            a.setType(((SymbolTableElement)symbolTable.get(j)).getType());
                            if (a.typename() == "Integer"){
                                    val_a =
((DevilInteger)((SymbolTableElement)symbolTable.get(j)).getValue()).getValue();
                                    if (val_a == 0){
                                        xi = new DevilInteger(0);
                                        xi.setType("Integer");
                                        return xi;
                                    }
                            }
                                                            else throw new DevilException("Not the correct type");
                        }

                }

            if (b instanceof DevilInteger){
                val_b = ((DevilInteger)b).getValue();
                if (val_b == 0){
                        xi = new DevilInteger(0);
                xi.setType("Integer");
                return xi;
            }
        }
        else if (b instanceof DevilVariable)
            for (int j= 0; j<symbolTable.size();j++){
                if (((SymbolTableElement) symbolTable.get(j)).getName().equals(((DevilVariable)
b).getVarName())){
                    b.setType(((SymbolTableElement)symbolTable.get(j)).getType());
                    if (b.typename() == "Integer"){
                        val_b =
((DevilInteger)((SymbolTableElement)symbolTable.get(j)).getValue()).getValue();
                        if (val_b == 0){
                            xi = new DevilInteger(0);
                            xi.setType("Integer");
                            return xi;
                        }
                    }
                                                else throw new DevilException("Not the correct type");
                }

            }

        xi = new DevilInteger(1);
        xi.setType("Integer");
        return xi;
    }
    /**
     *
□.* @param a
□. * @param b
```

```
. * @return

 */
 public DevilDataType or(DevilDataType a, DevilDataType b) {
        DevilInteger xi = null;
        int val_a, val_b;

        if (a instanceof DevilInteger){
            val_a = ((DevilInteger)a).getValue();
            if (val_a == 1){
                    xi = new DevilInteger(1);
                    xi.setType("Integer");
                    return xi;
            }
        }
        else if (a instanceof DevilVariable)
            for (int j= 0; j<symbolTable.size();j++){
                    if (((SymbolTableElement) symbolTable.get(j)).getName().equals(((DevilVariable)
a).getVarName())){
                        a.setType(((SymbolTableElement)symbolTable.get(j)).getType());
                        if (a.typename() == "Integer"){
                                val_a =
((DevilInteger)((SymbolTableElement)symbolTable.get(j)).getValue()).getValue();
                                if (val_a == 1){
                                    xi = new DevilInteger(1);
                                    xi.setType("Integer");
                                    return xi;
                                }
                        }
                                                else throw new DevilException("Not the correct type");
                    }

                }

            if (b instanceof DevilInteger){
                val_b = ((DevilInteger)b).getValue();
                if (val_b == 1){
                        xi = new DevilInteger(1);
                        xi.setType("Integer");
                        return xi;
                }
            }
            else if (b instanceof DevilVariable)
                for (int j= 0; j<symbolTable.size();j++){
                        if (((SymbolTableElement) symbolTable.get(j)).getName().equals(((DevilVariable)
b).getVarName())){
                            b.setType(((SymbolTableElement)symbolTable.get(j)).getType());
                            if (b.typename() == "Integer"){
                                    val_b =
((DevilInteger)((SymbolTableElement)symbolTable.get(j)).getValue()).getValue();
                                    if (val_b == 1){
                                        xi = new DevilInteger(1);
                                        xi.setType("Integer");
                                        return xi;
                                    }
                            }
                                                        else throw new DevilException("Not the correct type");
                    }

                }
            xi = new DevilInteger(0);
            xi.setType("Integer");
        return xi;

}
/**
 *
.* @param s
. * @return

 */
public DevilDataType getNumber(String s) {
        String type="";
        int aux=100;
        double aux1=0.0;
        // System.out.println("Got String DevilDataType: getNumber"+s);
        try{
            aux = Integer.parseInt(s);
            type = "Integer";
        }catch (NumberFormatException e1){
            //System.out.println("It is not a integer"+e1);
                                                 System.out.println(" e1 -> DevilInterpreter.getNumber"+e1 );
           // e1.printStackTrace();

            try {
                    aux1 = Double.parseDouble(s);
                    type = "Double";
            }catch (NumberFormatException e2){
                                                 System.out.println(" e2 -> DevilInterpreter.getNumber"+e2 );
                    //e2.printStackTrace();
            }
        }

        // System.out.println("DevilDataType: type"+type+" val "+aux);

        if (type.equals("Integer")){
            //System.out.println(Integer.parseInt(s));
            //System.out.println("DevilDataType -Creating DevilInteger");
            DevilInteger devilInteger= new DevilInteger(Integer.parseInt(s));
            devilInteger.setType("Integer");
            //System.out.println("DevilDataType -DevilInteger"+aux);
            return devilInteger;
        }
        if (type.equals("Double")){
            //System.out.println(Double.parseDouble(s));
            DevilDouble devilDouble= new DevilDouble(Double.parseDouble(s));
            devilDouble.setType("Double");
            //System.out.println("DevilDataType -DevilDouble"+aux1);
            return devilDouble;
        }
        return null;
}
/**
 *
.* @param s
. * @return

 */
public DevilDataType getVariable(String s) {
        DevilVariable var = new DevilVariable(s);
        return var;
}
```

```
/**
 *
□.* @param s
□. * @return

     */
    public DevilDataType getString(String s) {
            DevilString xs = new DevilString(s);
            xs.setType("String");
            return xs;
   }
    /**
     *
    *
     */
    public void println(){
            System.out.println();
    }
    /**
     *
□.* @param b
□. * @return

     */
    public boolean getForCondition(DevilDataType b){
            int val = 0;
            DevilInteger xi = null;
            if (b instanceof DevilInteger){
                 val = ((DevilInteger)b).getValue();
            }
            if (val==1) return true;
            return false;

    }


public boolean writeToFile(String a)
{
            FileOutputStream ostream = null;
            try {
            ostream = new FileOutputStream("devil.sh",firsttime);
            if(firsttime==false)
            {
                        firsttime = true;
            }
            PrintWriter pw = new PrintWriter(ostream);
            pw.println(a);
            pw.close();
            ostream.close();
            } catch(Exception ex)
            {
                        ex.printStackTrace();
                        return false;
            }
            return true;
}

public void block(DevilDataType param)
{
            int value;
            String varName="";
            //FileOutputStream ostream = null;
            boolean searchtable=false;
            boolean notfound = true;
            StringBuffer blockString= new StringBuffer();
            System.out.println(" inside block function typename and obj "+param.typename()+" "+param);
            try {
//ostream = new FileOutputStream("devil.sh",firsttime);
            //if(firsttime==false)
            //{
            // firsttime = true;
            //}
            //PrintWriter pw = new PrintWriter(ostream);


            if( param instanceof DevilVariable)
            {
                        varName = ((DevilVariable)param).getVarName();
                        searchtable = true;

            }

            //String varName = ((DevilVariable)param).getVarName();
            //System.out.println("\n\n\t\t <INTERPRETER> printOut\n\n");
            if(param.typename() == "array")
            {
            // System.out.println("I got here searching \n"+varName+" size "+symbolTable.size());
                        for (int i = 0; i<symbolTable.size();i++){

                                    if (((SymbolTableElement) symbolTable.get(i)).getName().equals(varName)){
                                    notfound = false;

                                    SymbolTableElement ste = (SymbolTableElement)symbolTable.get(i);
```

```
                DevilDataType dtp1 = ste.getValue();
                HashMap v =((HashMap)dtp1.getObj());

                Vector vb = (Vector)param.getObj();

                        DevilDataType dtp = (DevilDataType)vb.firstElement();

                value = ((DevilInteger) dtp).getValue();
                value = ((DevilInteger) dtp).getValue();

                DevilDataType dd = ((DevilDataType)v.get(new Integer(value)));

                if (dd.typename() == "String"){
                            System.out.println("Nowblocking the ip:
"+((DevilString)(param)).getValue()+"-.");
                    blockString.append("iptables -I INPUT -s ");
                    blockString.append(((DevilString)(param)).getValue());
                              blockString.append("-j DROP");
                }
                else if (dd.typename() == "Integer"){
                    System.out.println("So the final output is => ="+((DevilInteger)v.get(new
Integer(value))).getValue());
                    blockString.append(" ="+((DevilInteger)v.get(new
Integer(value))).getValue());
                }
                else if (dd.typename() == "Double"){
                    System.out.println("So the final output is =>
="+((DevilDouble)v.get(new Integer(value))).getValue());
                    blockString.append(" ="+((DevilDouble)v.get(new
                    }
                    if(notfound==true)
                            System.out.println("DEVIL COMPILER GENERATED ERROR Not found
\n"+varName +" Looks like used directly");
                                                                                              ); } }

                                                                                          lue()

                                                                              ))).getVa

                                                                         r(value

                                                                    Intege
           }
           else if (param.typename() == "String"){
               System.out.println("Nowblocking the ip: "+((DevilString)(param)).getValue()+"-.");
               blockString.append("iptables -I INPUT -s ");
               blockString.append(((DevilString)(param)).getValue());
               blockString.append(" -j DROP");
           }
           else if (param.typename() == "Integer"){
               System.out.println("So the final output in block is => ="+((DevilInteger)(param)).getValue());
               blockString.append(((DevilInteger)(param)).getValue());
           }
           else if (param.typename() == "Double"){
               System.out.println("So the final output in block is => ="+((DevilDouble)(param)).getValue());
               blockString.append(((DevilDouble)(param)).getValue());
           } else if (param.typename()==null) {


               for (int i = 0; i<symbolTable.size();i++){
                    if (((SymbolTableElement) symbolTable.get(i)).getName().equals(varName)){
                        //System.out.println(varName);
                                        if (((SymbolTableElement) symbolTable.get(i)).getType() == "Integer"){
                                //System.out.println("Integer");
                                if (((SymbolTableElement) symbolTable.get(i)).getValue() instanceof
DevilInteger){
                                  // DevilInteger dataType = new
DevilInteger((DevilInteger)((SymbolTableElement) symbolTable.get(i)).getValue());
                                   DevilDataType dtp =
(DevilDataType)((SymbolTableElement)symbolTable.get(i)).getValue();

                                   System.out.print("found in the symbol table object"+dtp);
                                                     System.out.print("found in the symbol table
value"+((DevilInteger)dtp).getValue());
                                   blockString.append(" = "+((DevilInteger)dtp).getValue());
                                               // System.out.print(((DevilInteger)dataType).getValue());
                                }
                        }
                        else if (((SymbolTableElement) symbolTable.get(i)).getType() == "Double"){
                                if (((SymbolTableElement) symbolTable.get(i)).getValue() instanceof
DevilDouble){
                                  // DevilInteger dataType = new
DevilInteger((DevilInteger)((SymbolTableElement) symbolTable.get(i)).getValue());
                                   DevilDataType dtp =
(DevilDataType)((SymbolTableElement)symbolTable.get(i)).getValue();

                                   System.out.print("found in the symbol table object"+dtp);
                                                     System.out.print("found in the symbol table
value"+((DevilDouble)dtp).getValue());
                                   blockString.append(" = "+((DevilDouble)dtp).getValue());
                                               // System.out.print(((DevilInteger)dataType).getValue());
                                }
                        }//end else if
                        else if (((SymbolTableElement) symbolTable.get(i)).getType() == "String"){
                                if (((SymbolTableElement) symbolTable.get(i)).getValue() instanceof DevilString){
                                  // DevilInteger dataType = new
DevilInteger((DevilInteger)((SymbolTableElement) symbolTable.get(i)).getValue());
                                   DevilDataType dtp =
(DevilDataType)((SymbolTableElement)symbolTable.get(i)).getValue();
 System.out.print("found in the symbol table object"+dtp);
                                                        System.out.print("found in the symbol table
value"+((DevilString)dtp).getValue());
                                   blockString.append(" = "+((DevilString)dtp).getValue());
                                 // System.out.print(((DevilInteger)dataType).getValue());
                                }
                        }//end else if
                }

            }


        }
        else {

                if ( param instanceof DevilInteger) {
                System.out.println(" printing =>"+((DevilInteger)param).getValue());
                } else if (param instanceof DevilString ) {
                                System.out.println(" printing =>"+((DevilString)param).getValue());
                } else if (param instanceof DevilDouble ){
                System.out.println(" printing =>"+((DevilDouble)param).getValue());
                }
```

```
                    else {
                    System.out.println("DEVIL COMPILER GENERATED ERROR : Unsupported or
Mismatched type ");
                    }

          }



                                        System.out.println("\nCommand Generated is "+blockString.toString());



                                        public void unblock(DevilDataType a)
                                        {

                                        }


                                        public void allow(DevilDataType a)
                                        { }


                                        public DevilDataType searchGlobalSymbol(DevilDataType dtp)
                                        {
                                        String varName=null;
                                        boolean doSearch = false;
                                        if(dtp instanceof DevilVariable)
```

```
ommand.append(blockString);
stem.out.println("<block>          "+blockCommand);

println(blockString.toString());
e();
n.close();
Exception ex)
                    {
.out.println("block Exception " +ex );
               ex.printStackTrace();
                    }
File(blockString.toString());
```

```
        {

                varName = ((DevilVariable)dtp).getVarName();
                doSearch = true;
        }

        if(doSearch == true) {
        for (int i= 0; i<symbolTable.size();i++)
        {
          if (((SymbolTableElement) symbolTable.get(i)).getName().equals(varName))
         {
                 SymbolTableElement ste =((SymbolTableElement)symbolTable.get(i));
                 DevilDataType dtp1 = ste.getValue();
                                          System.out.println(" Found inn symbol table "+ dtp1);
                return dtp1;
         }
        }
        }
        System.out.println(" Reached End of search and not found \n"+dtp);


        return null;

}


public void call_func(String name,Vector params)
{
        DevilDataType dtp_decl = null;
        DevilDataType dtp_passed = null;
        DevilDataType dtp = null;
```

```
            DevilDataType dtp1 = null;
            DevilDataType dtp_f = null;
            Vector arg_passed = null;
            try {

            //System.out.println(" Invoking the function\n"+name);
            //System.out.println("Invoking "+ name+" param list size is "+params.size());

            if(params.size()>0){
                        arg_passed = new Vector();
                        //dtp_passed = new Vector();
            }


            // the variable list to be passed is read b4 adding the symbol table for the function
            for(int i=0;i<params.size();i++)
            {

                        dtp = (DevilDataType)params.get(i);
            //dtp_passed.add(dtp);

            dtp_f = getDevilData(dtp);

            arg_passed.add(dtp_f);

            //System.out.println("\n param is "+params.get(i)+" and mapped Devildata from "+dtp_f);


}

// get the function from function table
DevilFunction functorun = (DevilFunction)functiontable.get(name);

if(functorun.args.size()!=params.size())
```

```
{
        System.out.println("Compiler error, Mismathced params in declaration and invokation\n");
        return;
}

// initialization function
// creates func [decl, passed] mapping
// adds symbol table
record.symTableInit(name);


// add args to symbol table
Vector arg_decl = functorun.getArgs();

for(int j=0; j< arg_decl.size();j++)
{
        // arguments from decl list
        dtp_decl =(DevilDataType)arg_decl.get(j);

        dtp_passed = (DevilDataType)params.get(j);


        String decl_var = ((DevilVariable)dtp_decl).getVarName();


        if(dtp_passed instanceof DevilVariable)
        {
                                        String passed_var = ((DevilVariable)dtp_passed).getVarName();


                dtp_passed = (DevilDataType)arg_passed.get(j);

                System.out.println(" Decl List "+decl_var +" Param List= " +passed_var);
```

```
                    record.addTosymTable(name,dtp_decl,decl_var,dtp_passed,passed_var);

            } else {

                    dtp_passed = (DevilDataType)arg_passed.get(j);

                                                System.out.println(" Decl List "+decl_var +" Param List " +null);

                                                record.addTosymTable(name,dtp_decl,decl_var,dtp_passed,null);
                    // name is null
// pass by value

                    }


                    // added argument to symbol table
                    // inited passed values in arg list


            }





            functorun.walker.expr(functorun.funcbody);


            System.out.println("\n Devildatatype function execution completed "+functorun);

            record.removeSymTable();

            // now this need to replace parameter names in function invocation

            } catch (Exception ec)
```

```
        {
                ec.printStackTrace();
        }

}


public DevilDataType getDevilDataType(String data)
{
        int a;
        double b;
        String c= null;
        DevilDataType dtp = null;
        dtp = getNumber(data);
        if(dtp==null)
        {

                dtp =(DevilDataType) new DevilVariable(data);

        }


        return dtp;




}

public void new_func(DevilWalker walker, String name, Vector args, AST body ) {

        try {
            //System.out.println("Name of Function \n"+ name);
            //for(int i=0;i<args.size();i++)
//System.out.println("Arg list \n"+ (DevilDataType)args.get(i));

            //System.out.println("Body of Function \n"+ body);

            //System.out.println("Number of children of statement"+body.getNumberOfChildren());

            //walker.expr(body);


            //System.out.println("Number of children of statement"+body.getNumberOfChildren());
            //walker.expr(body);



                              functiontable.put( name, new DevilFunction( walker, name, args, body, functiontable ) );

            //System.out.println("Stored Func in FunctionTable\n");
            } catch (Exception ex)
            {
                  ex.printStackTrace();
            }
    }


public String[] VectoString( Vector vec ) {
            //System.out.println("Called Vectorto String"+vec);
        String[] sarray = new String[ vec.size() ];
        for ( int i=0; i<vec.size(); i++ ){
          sarray[i] = (String) vec.elementAt( i );
              //System.out.println("\nNext elemnt ="+sarray[i]);
            }
        return sarray;
      }



  public void printOutput(DevilDataType out)
  {

            boolean searchtable = false;
            DevilDataType dd = null;
            String varName="";
            try {
            if( out instanceof DevilVariable)
            {
                        varName = ((DevilVariable)out).getVarName();
                        searchtable = true;
                        dd = record.lookup(varName);


            } else {

            dd = out;
            }
            if(dd==null)
            {
             dd =out;
            }
            System.out.println("Inside print Out"); if ( dd instanceof DevilInteger) {
                                                    System.out.println(" printing =>"+((DevilInteger)dd).getValue());
            } else if (dd instanceof DevilString ) {
                                                     System.out.println(" printing =>"+((DevilString)dd).getValue());
        } else if (dd instanceof DevilDouble ){
                                System.out.println(" printing =>"+((DevilDouble)dd).getValue());

        } else if (dd.name=="array" )
        {
                //System.out.println("Type is Array"+out);


                Vector v = (Vector)dd.getObj();
                HashMap hm = v.get(1);
                int val =
                                System.out.println(" Printing => "+hm.get(new Integer(val)));
        }
        else {
                System.out.println("DEVIL COMPILER GENERATED ERROR : Unsupported or
Mismatched type ");

        }
        } catch (Exception e)
        {
                e.printStackTrace();
        }

        //return;

}

} /*
. * Symbol Table Element
.* Symbol Table maintained as a Linked List

 */
```

```
/* | | | |
. * | type | name | value |
. * | | | |
. * */

public class SymbolTableElement {
	/* Basic Symbol table entries */
	private String type;
	private String name;
	private DevilDataType value;

	/* DevilDataType Constructor */
	public SymbolTableElement(String type, String name, DevilDataType value){
		this.type = type;
		this.name = name;
		this.value = value;
	}
	/* DevilInteger Constructor */
	public SymbolTableElement(String type, String name, DevilInteger value){
		this.type = type;
		this.name = name;
		this.value = value;
	}
	/* SymbolTableElement Constructor */
	public SymbolTableElement(SymbolTableElement element){
		type = element.type;
		name = element.name;
		value = element.value;
	}
	/* GetName */
```

```
        public String getName(){
                return name;
        }
        /* setvalue */
        public void setValue(DevilDataType value){
                this.value = value;
        }
        /* getValue*/
        public DevilDataType getValue(){
                return value;
        }
/*                      getType                 */
        public String getType(){
                return type;
        }
/*                      setType                 */
        public void setType(String t){
                type = t;
        }
}
/*      Main.java*/
import java.io.*;
import antlr.CommonAST;
import antlr.collections.AST;
import antlr.debug.misc.ASTFrame;

class Main {
public static void main(String[] args) {
try {
 DataInputStream input = new DataInputStream(System.in);


 DevilLexer lexer = new DevilLexer(input);

DevilParser parser = new DevilParser(lexer);
 parser.program();

CommonAST parseTree = (CommonAST)parser.getAST();

 //System.out.println(parseTree.toStringList());

 ASTFrame frame = new ASTFrame("\n\nAST from the DEVIL parser\n\n",
parseTree);
 frame.setVisible(true);

 DevilWalker walker = new DevilWalker();
 DevilDataType drake = walker.expr(parseTree);

System.out.println("\n\n\n");


 } catch(Exception e) { System.err.println("Exception: "+e); }
```

```
    }
}
```