EcoSL

Economical Spreadsheet Language

Somenath Das
Satish Srinivas
Lalit K Kanteti

Table of Contents

1. Introduction

1.1 Spreadsheets and EcoSL

Spreadsheets are a well-known and widespread set of language systems. They have been found to be very useful, primarily in the business world, for the manipulation and presentation of financial and other sorts of tabular data. EcoSL (Economical Spreadsheet Language) will be a language for users who want to carry out simple to complex operations on sets of data analogous to what one could do using conventional spreadsheets. Functionally, EcoSL will be more than a basic scientific calculator and close to full scale spreadsheet like Excel or Openoffice.

EcoSL will serve as a good alternative to the current users of commercial spreadsheet software. The most commonly used functions like Mathematical Computations, data interpretation; graphical representation of data such as graphs would be supported by EcoSL. EcoSL is also being designed to support users who log into terminal based systems. Apart from its simplicity and powerful features, EcoSL is completely Free!!! We intend to release it under GPLV3 license.

EcoSL is designed to be a translated language with it target language being Java. Programs written in EcoSL will be parsed and analyzed by ANTLR[1], to Java code and then executed by the JVM. ANTLR is a tool in Java that provides a framework for constructing translators. ANTLR generates a lexer, parser and tree walker.

1.2 Basic features

- No types: EcoSL is a type less language. That is, user don't have to specify the type of a variable. A user can directly use a variable. Variable are dynamically bound. Also, the user can reuse the variable for storing other data types. For example, a = 1; .....a=”abc”;
- Coordinates: EcoSL implements coordinates as a data type. It represents a cell in the spreadsheet.
- String operations: EcoSL allows basic string operations. A user can operate operations on strings the same way as with integer or float.
- Operator overloading: Operators (like +, ==, <=, <,...) have been overloaded for strings and coordinates.
- User-friendly: With no types, EcoSL is very easy to use. Specially for the users who doesn't have to know the details of a language (like data types).
- Scope: EcoSL has dynamic scoping. User can define functions inside functions.

- Control-flow: EcoSL provides users with basic level programming language control of events using conditional statements, iterations, break & continue.
- User defined function: EcoSL allows the user to define their own functions which makes their resulting code more modular, powerful, and reusable.

A function can contain any number of arguments or no arguments. User doesn't have to specify any return type. A user may or may not return any value. Similarly, if a function is returning something, it is not mandatory (like in C) to receive the return value into some value.

2. Language Tutorial

A tutorial walkthrough to familiarize one with the mechanics of using EcoSL for your work

2.1 Simple example

```
{
            func max(a,b){
               if (a>=b)
                   return a;
               else
              return b;
               }
               [1:2] = 2;
               [3:5] = 6;
               max(5,99);
               c = max(5,99);

    println(c);
    print(max([1:2],[3:5]));
}
```

2.2 Running EcoSL interpreter on our sample program: HOWTO

1. Translate: the MySample.eco to a .java program by typing the following in your shell.

   $java <MySample.eco>

2. Compile: takes in to <mySample.java> file and generate the java class file.

   $javac <MySample.java>

3. Execute:

   $java MySample

2.3 Complex example

The example below demonstrates another rich attribute of EcoSL which is dynamic scoping

```
{

a = 1;

func mx(){

        println("In Function Mx picking Global a:",a);

        a=2;

       println("In Function Mx picking new Def of a:",a);

       func mx1(){

                  println("In Function mx1 picking a defined in mx1 a:",a);

                 a=3;

                 println("In Function mx1 picking a defined locally a:",a);

                 }

      mx1();

}

mx();

println("Global a:",a);

}
```

As can be seen from the program above, EcoSL supports nesting of functions. Fun mx() defines another functions func mx1() in it's scope. A run of the program gives the following output:

```
In Function Mx picking Global a:1
In Function Mx picking new def of a:2
In Function mx1 picking a defined in mx1 a:2
In Function mx1 picking a defined locally a:3
Global a:1
```

3. Language Reference Manual

3.1. Introduction

The EcoSL is designed to provide a platform for users to do simple and complex operations using a spreadsheet layout on different kinds of numerical and string data. It is platform independent and stresses on writing minimal code to execute arithmetic, logical or any aggregated operations over single or multiple cells in a spreadsheet. This manual defines the language EcoSL proposed earlier in a related white paper. Regular expression notation has been used to make productions more perspicuous.

3.2. Lexical Conventions

3.2.1 TOKENS

There are classes of tokens: identifiers, keywords, string literals, operators, and other separators. Spaces, tabs and comments separate tokens but are otherwise ignored.

KEYWORDS:

Following identifiers are reserved for use as keywords in EcoSL:

cell	group
if	else
for	forc
forr	while
continue	true
false	return
func	break

IDENTIFIERS:

EcoSL allows identifiers to be declared and used without specifying its type. Identifiers are associated with a type during runtime. Cell identifiers are aliases of the existing cells. These aliases are typically used for repetitive call of same cell in operations. Identifiers can be a sequence of letters and digits; the first character must be an alphabet or an underscore. Identifiers are case sensitive.

3.2.2 Data Types

Derived Data Types:

cell: Cell represents a basic addressable unit in the spreadsheet. Data taken from the input or computed through user program is stored in cells.
group: A group is a combination of cells. All operations valid on cell are valid on group.

Internal Data Types:

Each cell on a sheet is considered to be an identifier in itself. Type checking is done during run time. Each cell can store data of the following data-types:

INT: If compiler encounters with sequence of integers then it is mapped to int data type.
BOOLEAN: If any cell has true or false cell automatically be assigned Boolean data type.
FLOAT: Any alphanumeric value which starts with an integer part, a decimal point, a fraction part, an e, and an optionally signed integer exponent.
STRING: A sequence of characters surrounded by double quotes ‘‘ ’’.

3.2.3 BUILT IN FUNCTIONS:

Most of the EcoSL operations are carried out using following built in functions. Names of functions are self explanatory of their operation. These functions (except conout and displaysheet) are for group operations.

max	min
avg	sum
count	scale
println	print

3.2.4 Constants

There are three types of constants in EcoSL: integer, floating point and strings.

Integer: A number consists of a string of digits with an optional sign ‘-‘. All built-in mathematical operations performed on numbers assume base-10 format.

Floating Point: A decimal representation with an optional '.'.

String: A string is a sequence of characters enclosed by double quotes: “string”. A double quote inside the string is indicated by two consecutive double quotes. The second double quote is ignored in the final token.

3.2.5 Comments

The characters /* start what may be a multi-line comment terminated by */, while the characters // start a single-line comment.

3.3 Declaration

Spreadsheet can be viewed as a collection of cells. A cell represents a basic addressable unit in the spreadsheet. Each cell is addressable by it's co-ordinates relative to an origin. A cell or a collection of cells can be declared as a group.

EcoSL does not mandate the user to declare an identifier before using it. User can use a variable without declaring it. Adding to the flexibility the user does not need to provide a type for any identifier he would use. Type checking for all identifiers is done at the run time.

Declaration in EcoSL looks like this:

id = constants

constants --- integer|float|string

id--- variable|coordinates

coordinates--- [x:y]

x,y – integer constants

3.4. Expressions

3.4.1 Primary Expressions

Primary expression includes identifier, constant and function calls.

Identifier: An identifier itself is a left-value expression. It will be evaluated to some values bounded to this identifier.

Constant A constant is a right-value expression, which will be evaluated to the constant itself.

Function call: A function call consists of a function identifier, followed by a list of arguments enclosed by ( ). The list of arguments contains zero or more arguments separated by a comma ",". Each argument is an expression. Function call is a right-value expression.

3.4.2 Arithmetic operators

Arithmetic operators take primary expressions as operands.

Unary arithmetic operators

Unary operators '+' and '-' can be prefixed to an expression. '+' operator returns the expression itself whereas the '-' operator returns the negative of the primary expression. They are applicable to all the identifiers except groups.

Example: –14 or +5

Multiplicative operators

Binary operators '*', '/', and '%' indicate multiplication, division, and modulo, respectively. They are grouped left to right. They are applicable to all the identifiers except groups. Example: 18 * a or 12 / 3 or 63 % 5 (a is an identifier)

Additive operators

Binary operators '+' and '-' indicate addition and subtraction, respectively. They are grouped left to right. They are applicable to all the identifiers except groups.

Example 19 + 4 or 14 – b (b is an identifier)

Relational operators

Binary relational operators >=, <=, ==, !=, > and < indicate whether the first operand is greater than or equal to, less than or equal to, equal to, not equal to, greater than, or less than the second operand, respectively.

Example. a != 12 or b == 4 or c >= 5 (a, b, c are identifiers)

Assignment operators

Assignment operator in EcoSL is =. It is associative from right to left. Assignment operator requires modifiable lvalue as their left operand. The type of an assignment expression is that of its unqualified left operand. The result is not an lvalue. Its value is the value stored in the left operand after the assignment.

Logical operators

Logical operators take relational expressions and numbers as operands. They have the lowest precedence. In EcoSL '= =' (is equal to), '&&' (and), '!=' (is not equal to) and '||' (or) are logical operators.

3.4.3 Productions in Expressions

expression:

primary
- expression
expression binop expression
lvalue asgnop expression

primary:

co-ordinate
constant

primary (expressionlistopt)

lvalue:

co-ordinate
(lvalue)

binop:

* / %
+ -
< > <= >=
== !=
&&
||

asgnop:

=

3.5. Statements

A statement is a complete instruction to the computer. Statements are basically elements of a program. A sequence of statements will be executed sequentially, unless the flow-control statements indicate otherwise.

3.5.1 Statements in { and }

A group of zero or more statements can be surrounded by { and }, in which case they altogether are treated as a single statement. This is true of all function calls.

3.5.2 Assignments

An assignment is in this form:
id = expression ;
where id is a variable or coordinate and ; is the terminator of this assignment statement.

3.5.3 Conditional statements

A conditional statement is in form:

if (expression) statement
or

if (expression) statement else statement

Conditional statements choose one of a set of statements to execute, based on the evaluation of the expression. The expression is the controlling expression. For both forms of the *if* statement, the first statement is executed if the controlling expression evaluates to nonzero. For the second form, the second statement is executed if the controlling expression evaluates to zero. An *else* clause that follows multiple sequential *else-less if* statements is associated with the most recent *if* statement in the same block.

3.5.4 Iterative statements

Iterative statements are basically loops. There are two kinds of loops, for and while.

For statement

The *for* statement has the following forms:

for (expression ; expression ; expression) statement

The first expression specifies initialization for the loop. The second expression is the controlling expression which is evaluated before each iteration. The third expression often specifies incrimination which is evaluated after each iteration.

forc (co-ordinate$_1$ asgnop co-ordinate$_2$ co-ordinate$_3$) statement

This is specialized for loop with just one expression which is defining the range of cells in a column. Values of co-ordinate$_1$ are operated against values defined from starting cell of *co-ordinate$_2$* to destination cell *co-ordinate$_3$*

forr (co-ordinate asgnop co-ordinate co-ordinate) statement t

This is again specialized loop similar to above one with exception that it runs across a row.

While statement

The while statement is a general iterative statement. It is in this form:

while (expression) statement

The expression is evaluated at the start of each loop. If the expression is evaluated to false, the loop will be skipped. An infinite loop will be introduced if the expression will always evaluate to true and a break statement is not used inside the loop.

Break statement

The break statement will break the inner-most or labeled iterative statement. This statement consists of keyword break then followed by a ‘;’.

Continue statement
The continue statement will end the current iteration of the innermost or labeled iterative statement and proceed to its next iteration. This statement consists of keyword continue then followed by ';'.

3.5.5 Function invocation and return

Function call
Different from other expressions, a function call followed by ';' can be a single statement.

Return statement
The return statement is used inside the function definition body, in order to return from the function at that point. It can be followed by an optional expression, for the return value, and ';'.

3.5.6 Productions in Statements

statement:

expression ;
{ statementlist}
if (expression) statement
if (expression) statement else statement
for (expression ; expression ; expression) statement
forc (co-ordinate asgnop co-ordinate co-ordinate) statement
forr (co-ordinate asgnop co-ordinate co-ordinate) statement
while (expression) statement
break ;
continue ;
return ;
return (expression) ;

statement-list:

statement
statement statement-list

3.6. Input

User should provide input data in a CSV file. Input can be either a single file or can be multiple files. The input file should consist of data on which the user intends to run his program and carry

out operations

3.7. OUTPUT

As an output, user can opt for:

- A graph where the user data are matched and a line graph is drawn on the screen.
- A sheet where the data are displayed in tabular form.
- File where the output data are stored in a CSV file.

3.8. Scope and Namespace

EcoSL has dynamic scope. This language has one namespace.

3.9. ERRORS

All kind of error messages will be displayed in console.

4. Project Plan

4.1 Project Process

All team members in the group followed these processes during the various stages of development of the project.

4.1.1 Planning and Specification

We started with a group of 4 but just before submission of our whitepaper one of the members dropped off. As a result, we had to revise our proposed plan. Work allocated was based more on the ability of each member rather than dividing equally. General guidelines and specifications were planned during group meetings at different stages of the project life cycle. In the beginning, we focused on defining rules for a general language and then ultimately for the EcoSL programming language. As the project progressed, we began designing and specifying the different components that would actualize our goals. As development began, we started facing number of issues related to Java and Antlr. In order to account for various design and code related issues we were in regular touch with Phong (our TA). These meetings were extremely helpful in understanding the scope of the language, efficiency of implementation, setting up incremental goals, and division of labor. We regularly missed our internal deadlines and most of the work is done at the end.

4.1.2 Development and Testing

Each team member was responsible for testing his work first (unit testing). Then we checked each other's modules to verify the functionalities. All work which compiled properly was meant to be checked into CVS which helped in testing other people's work. This made each other aware of what the others were doing in the project and made it easier in our discussions. Each of the developed stages was subjected to unit testing. Initially we planned to do integration and regression testing after every change in code base. Most of the major bugs were identified and reported during the unit testing phases.

4.2 Team Responsibilities

Listed below are the primary responsibilities of each team member.

Team Member	Responsibility

Srinivas	Grammar Design Parser/AST Walker Testing
Somenath	Parser Walker / Code Generation Classes Design and Implementation Testing
Lalit K Kanteti	CVS design and maintenance Grammar Design Implement few Classes designed by Somenath Presentation

Note: Documentation was generally a group effort. The Final Report was written by Lalit

4.3 Programming style (Coding Conventions)

We agreed upon maintaining certain conventions. These conventions were learnt from class or from internet. All members must adhere to the following coding guideline during all phases of the project. We followed java UpperCamelCase for the class names and lowerCamelCase for variables and method names.

- Use descriptive names for identifiers
- Most critical sections of the code should have descriptive comments
- Every logically complete modification or addition to functionality should be version -controlled.
- Version control comments should explicitly specified what the change was
- Every module should include a brief description of its overall functionality
- **Every file should state its name, purpose, and author name in the header**
- Nested blocks of statements should be evenly indented and statements on the same level should be indented by the same amount of spaces.
- Avoid writing blocks all in one line.
- Avoid writing long expressions in single line.
- Comments are required for logical decision and initial declaration of logic variables.

These conventions were followed strictly as much as possible because it will benefit readability of the code for third party reader. Code should be written in a clear layout that will be easy to read.

4.3.1 ANTLR Examples

ANTLR productions always followed the format of:

production: <non-terminal> or <terminal>
| non-terminal or <terminal>
;

If a multiple rule existed, the pipe symbols were always placed at the beginning of the statement. All semi-colons were placed on single lines. The lexer token names are in upper case and the parser rules are in lower case.

4.3.2 JAVA Examples

JAVA codes were written using above standard programming conventions. Initially we thought to adhere to Hungarian or/and Camel notations. But because not every member was very familiar with convention and initial programs weren't adhered hence we had to drop the idea. Following is example

```
{
        for(;;){
                stmt1;
                if(<too long expression>
                                <too long expr>)
                {
                        Stmt2;
                }
        }
}
```

4.4 Project Timeline (Planned)

Event	**Estimated End Date**
Finalize Proposal	Sept 23, 2007
Design for Lexer & Parser	Oct 9, 2007
Working Lexer & Parser	Oct 12, 2007
LRM	Oct 19, 2007
Walker design and Implementation	Nov 9, 2007
Code Generation	Nov 26, 2007
Testing	Dec 7, 2007
Report & Presentation	Dec 15,2007

4.5 Software Development Environment

Everyone had new laptops which have Windows Vista and hence we used all windows based applications. EcoSL was developed using the Microsoft Windows operating system. The primary applications used to develop the project were JAVA SDE 1.5.x and ANTLR 2.7.3. The source code was controlled using CVS, which permitted the integration of the Windows environment with files stored in a UNIX environment. CVS was accessed by team using Eclipse – CVS perspective

4.6 Project Log

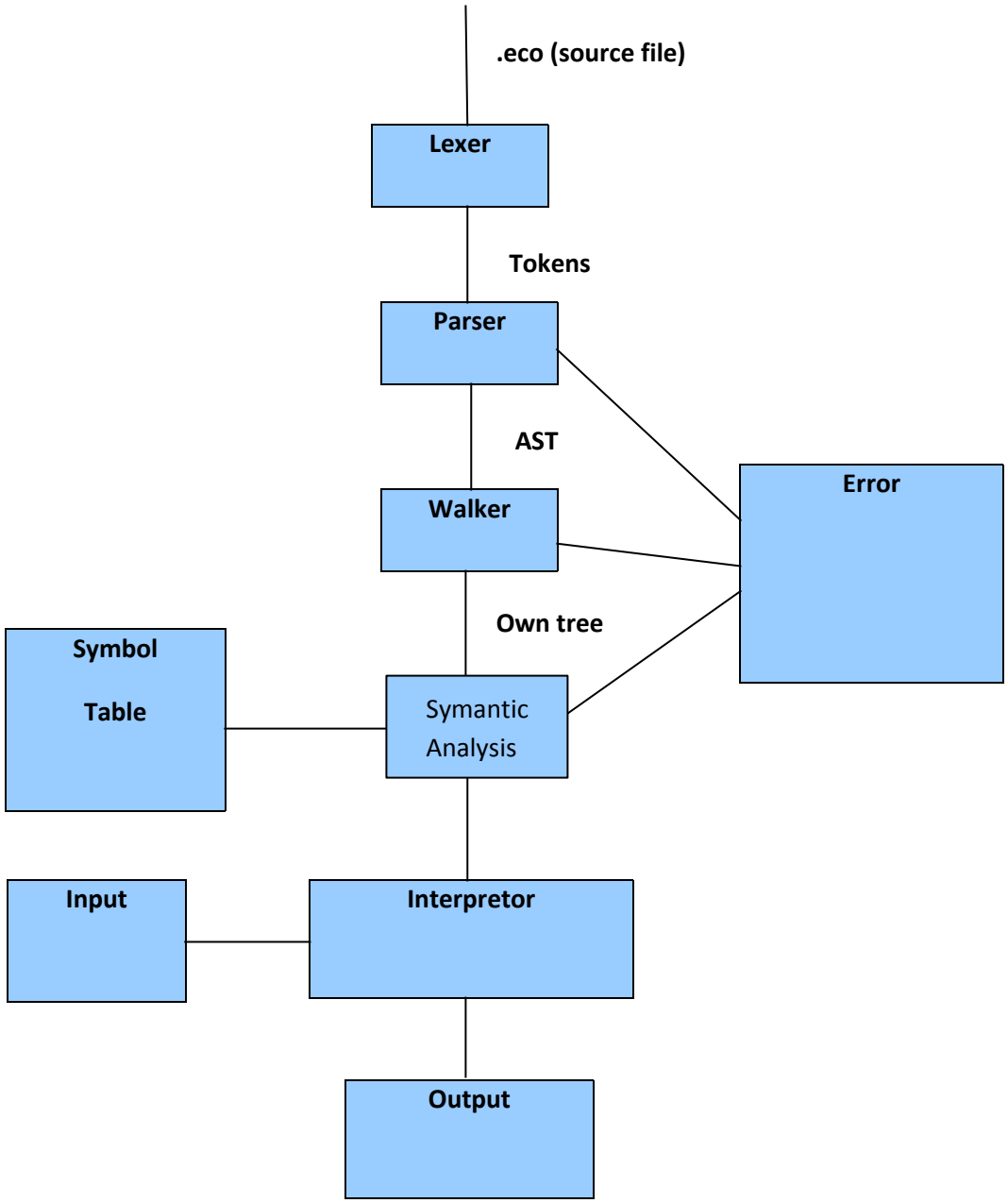
A running log of all key dates was accounted for and is listed below.

Event	**Actual End Date**
Finalize Proposal	Sept 25, 2007
Design for Lexer & Parser	Oct 13, 2007
Working Lexer	Oct 19, 2007
Working Parser	Nov 6, 2007
Walker design and Implementation	Dec 9, 2007
Code Generation	Dec 17, 2007
Testing	Dec 18, 2007
Report & Presentation	Dec 19, 2007

5. Architectural Design

5.1 Compiler Overview

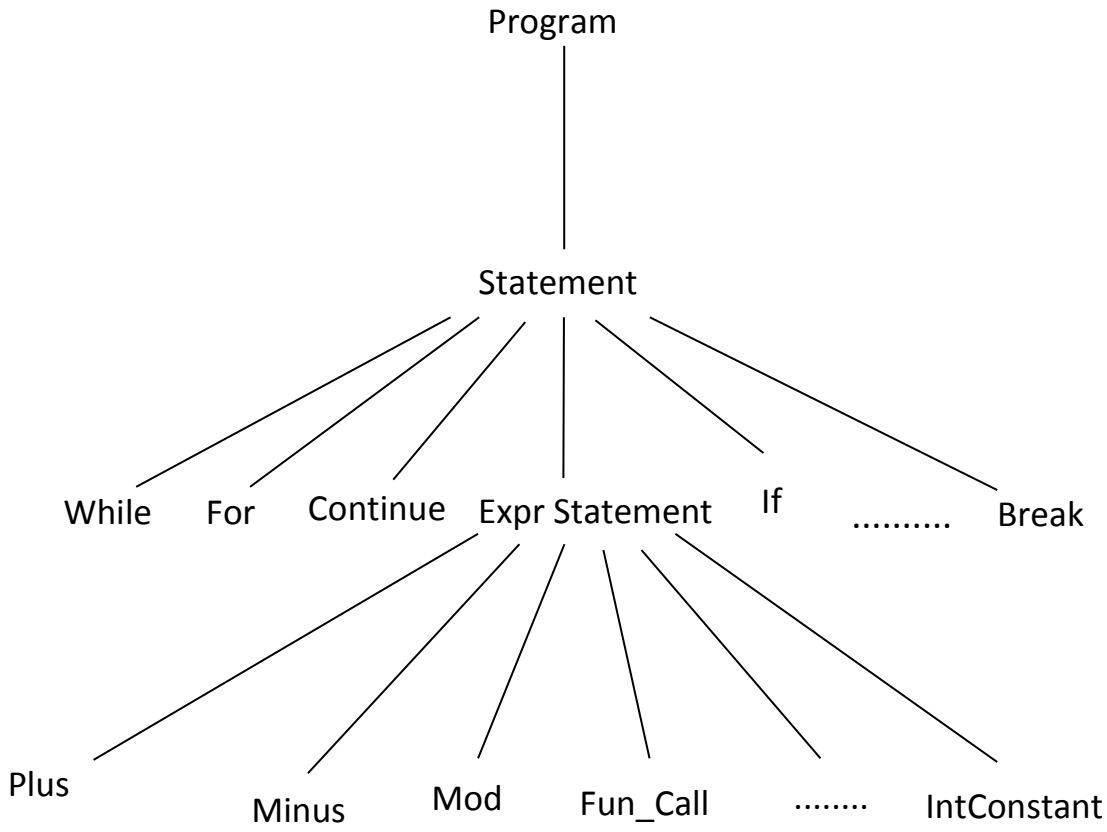
The EcoSL interpreter is developed using ANTLR and JAVA. ANTLR was used primarily for the lexer and parser; the walker was written using a combination of ANTLR and embedded JAVA code. The executer (runtime environment) and the backend (class functions) were written entirely in Java.
The diagram below shows a brief overview of the dataflow in the compiler.



A user inputs a .eco file to the interpretor. The first thing that the translator does is create a **lexer** on that input file. The **lexer** was created using ANTLR and produces the tokens that were defined in the grammar file. This **lexer** is then passed to the *parser* which was also defined in the ANTLR grammar file. The **parser** receives tokens from the **lexer** as is necessary as it parses the file to match the file to the grammar rules that we have defined. If there are syntactical errors, the **parser** reports these errors to the user via standard out. These errors were modified slightly in the grammar so that they would be easier to interpret by the user. If the inputted source file is syntactically correct, the **parser** creates an **Abstract Syntax Tree or AST** from the grammar rules. Now, the **walker** walks through the **AST** generates a simpler tree (own tree). The idea behind is to split the work into two parts. The option is to implement the semantic analysis in the **AST** itself. But that takes a more efforts and it is a little complex too. So, we split it up into two parts. First part generates a new simpler tree from the AST and the second part does the semantic analysis. The semantic analyzer contains the logical classes. The **semantic analyzer** catches all the errors and exceptions and displays relevant error messages at the console. Also the semantic analyzer interacts with the **symbol table** to store and retrieve variables or coordinates. After the semantic analysis, if there are no errors, the **interpretor** executes all the statements one by one. Also the interpretor takes the data for the program from input file and put the results in an output file (Although the input and output file implementation has not been done yet but they are in progress).

The class diagram is shown below:

In EcoSL, everything is a statement. And the statement expands to different types of statements and expressions. We have two interfaces, **Stmt** for statements and Expr for expressions. All the statement classes implement **Stmt** interface and expression classes implement **Expr** interface.



6. Test Plan

The testing for ECOSL was split the testing into three stages – unit testing, integrated testing and regression testing. Each stage helps in weeding out flaws that the previous stage may not have caught.

6.1 Unit Testing

Unit testing is generally seen as a “white box” test class which is biased to looking at and evaluating the code as implemented, rather than evaluating conformance to some set of requirements. For ECOSL, unit testing was performed on the major modules of the lexer, parser, walker, executor, and the back-end. Each of these modules was tested separately with sufficient test cases to satisfy their behavior matched the code as implemented. Because of these test cases only we encountered so many errors that we had to change our design altogether which lead to change of language in some cases.

6.2 Integrated Testing

Once each module has been satisfactorily tested, the next step was to combine these modules into a single working unit and test it. The lexer and parser were in a single unit of code, integrated with the walker and interpretor. At this time, limited features of EcoSL were implemented and the entire unit was tested. Our integration testing lead to lot more bugs which we thought were solved with unit testing. For example we initially started with concept that everything would be inside function which didn’t work well after integration. Many such occurrences led us to change whole concept again and wastage of lot of time.

6.3 Test cases

In a co-operative development environment, the synergistic productivity would be best only if everyone gives bug free code. We had this principle clear when we began the project. So in EcoSL, we reduced our debug time to a good extent. Categorically unit testing was performed as below:

Lexer/parser: The grammar of a language is the foundation stone for any translator implementation for the language. With this in mind we started of by testing the very basic constructs of arithmetic expressions, statements to the more complex functions, blocks of statements. A simple approach was to have actions defined across the various levels of production rules. On giving a sample input if the reductions have been successful the action would get printed.

Eg. On giving an input [1:2] = [2:2] + [3:2];

it would print expression_statement reduced successfully.

Similarly moving along the hierarchy, does a statement block get reduced to a statement successfully? Many small snippets of code like the ones above were used to incrementally test the lexer/parser of EcoSL.

AST Construction: After having good success with the parser the next step was to give some structure to our result parse trees by using ANTLR built in buildAst functionality. Again to test the AST the following snippet of code was added in out Main.java file which helped us to visualize the problems in our AST construction.

```
/*Get the AST from the parser*/
CommonAST parseTree = (CommonAST)parser.getAST();
/*Print the AST in a human-readable format */
System.out.println(parseTree.toStringList());
/*Open a window in which the AST is displayed graphically */
ASTFrame frame = new ASTFrame("AST from the ECoSL parser", parseTree);
frame.setVisible(true);
```

Having a visual representation of our AST construction helped us to understand and also quickly correct our mistakes. We spent some time
In getting the AST for the statement_block, expr_statement, co-ordinate right.

Walker: Having a good AST made the walker construction simple. The visual notation of AST helped us to understand the hierarchy of syntax tree very well. The only time taking task while working on the walker was to create stubs of all the classes which we would use for the semantic phase. Again to see how our tree walker was working we added the following snippet to the Main.java program

```
EcoslWalker walker = new EcoslWalker();
Program prog = walker.program(parseTree);

AST results = walker.getAST();
DumpASTVisitor visitor = new DumpASTVisitor();
visitor.visit(results);
```

Static Semantic Checking:

Summary of the testing Phase

Testing Phase	Breakdown of test labour
Primary grammar debugging and Testing	Srinivas & Somnath
Static Semantic Checking Class	Somnath & Lalit
Sample code snippets for EcoSL	Srinivas, Somnath & Lalit

The list of final test Cases:

Basic Undeclared Identifier testing

```
{
 /*
   * Demonstrates it's an interpreter. Symbol table is working
   */
A    = 10;
[1:1] = 20;
[1:2] = A + [1:1];
Println(A);
C    = C * [1:2];
}
```

Output:

```
30
Undeclared identifier C
```

Control Flow Statements

```
{
 /*
  * Simple while loop execution
  */
  i = 10;
  while (i > 0){
    i = i-1; println(i);
  }


/*
 * Testing certain control statements in iterations
 */
  i = 20;
  while (i >= 0) {
     if ( i % 2 == 0 ) {
         print(i , " ");
     }
     i = i - 1;
  }
  println("     ");

 /*
  * Testing continue in iterations
  */
  i = 30;
  while ( i > 0 ) {
       if ( i % 2 != 0 ) {
            i = i - 1;
            continue;
       }
       print(i, " ");
       i = i - 1;
  }
/*
  *Testing break in iterations
  */
  println(" "," ");
  i = 30;
  while( i > 0) {
       i = i - 1;
       print(i, " ");
```

```
      if ( i == 20)
        break;
  }

}
```

Scoping rules verification

```
{
          a = 1;
          func mx(){
          println("Global a:",a);
          a=2;
          println("1st func a:",a);
          func mx1(){
                    println("2nd func a:",a);
                    a=3;
                    println("2nd func a:",a);

                    }
          mx1();
          }
          mx();
          println("global a:",a);
}
```

7. Lessons Learnt

As with any project, there is always scope for learning something useful for the future. In our case we have learned a lot many lessons than we expected that will be useful for the future.

7.1 Lalit K Kanteti:

- Don't wait for an angel to come and help you.
- Use TA to the fullest. If you are stuck don't feel ashamed to approach your TA or professor. If you don't lean they don't loose anything and you loose your grade.
- Java is easy only if you have enough practice
- Make proper comments as you will not remember why you used a particular method.
- Design properly and get it approved with your TA or Professor so that you don't have problems while development. Bad design eats up lot of time and you will be running in no direction.
- Don't keep others waiting for your status and meetings

7.2 Somenath Das

- Got a clear picture of how a language actually works
- Learned the basics of programming language. Now, when I write a code in any language, I make less errors. I can see a clearer picture of control flow.
- Understand scoping and binding very well.
- Lots of coding sharpened my Java skills.
- Learned antlr and CVS.
- Learned what to do and what not to do in a team project.
- Early planning and weekly deadlines.
- Last but not the least, it was fun to write a programming language using a programming language.

7.3 Srinivas

I found out that effective planning, communication and commitment are key ingredients required in a co-operative devolopment project. To kickstart the project it is very essentialto have the development environment set up as quickly as possible - this includes versioning control system, IDE and support tools. Having the entire team at the

same starting point helped avoiding delays due to any one individual. Working in this project I learned the following: working with ANTLR; This tool which was the basis for our translator is not very well documented.

Yes, one always has the ocean of discussion threads on the W3 (www). Nevertheless,learning the tool the right way was quite time consuming. It was an iterative process of many trail and errors. However after understanding it I have developed a great appreciation towards this tool.

I was not very familiar coding in Java and this project did help me to polish certain areas of my programming.

The other important learning was from the phase wise testing of the code I was writing. It helped to integrate from other team members and on running simple test cases were able to periodically reduce the number of errors. It helped me realise that however good you understand a language, tool unless you test your work you will never realise what you were doing wrong. I had improved on this principle throughout, I guess so!!!!

Lastly, it was very important to hold regular team meetings to brainstorm on certain issues and as always **proved synergistic efforts are fruitful.**

7.4 Future Plans

We are targeting our next release to have following:

- File handling, i.e. users can perform I/O operations on files. He will also have freedom to input and output data in various formats.
- Graphical representation
- Control flow dependency between coordinates
- Multiple spreadsheets and various operations among the different spreadsheets.

8. Appendix

Lexer and Parser code
Filename: NECoSL.g

```
class P extends Parser;

options {
                  k = 2;
                  buildAST=true;
                  exportVocab = NECoSL;
}

tokens
{
      PROGRAM="program";
      STMT_BLOCK="stmt_block";
      STMT="stmt";
      EXPR_STMT="expr_stmt";
      EXPR_LIST="expr_list";
      FORMAL_LIST="formal_list";
      FORMAL_ARGS="formal_args";
      FUNC_CALL="func_call";
      FUNC_DECL="func_decl";
      BREAK="break";
      CONTINUE="continue";
      RETURN="return";
      COORDINATE="coordinate";
      }

imaginaryTokenDefinitions:
    SIGN_PLUS
    SIGN_MINUS
    ;

program        :  LBRACE! (statement)+  RBRACE! {
                        #program = #([PROGRAM, "program"], program);
                        }
               ;

statement      : ( func_decl
                     |   if_statement
                     |   for_statement
                     |   forc_statement
                     |   forr_statement
                     |   while_statement
                     |   break_statement
                     |   continue_statement
                     |   return_statement
                     |   expr_statement SEMI!
                     |   (LBRACE!
                    (statement)+
                    RBRACE!)  {
```

```
                                #statement = #([STMT_BLOCK,
                               "stmt_block"], statement);
                               }
                  )
               ;


func_decl      : "func"! ID LPAREN! (formal_args)* RPAREN! LBRACE!
(statement)* RBRACE! {
                     #func_decl = #([FUNC_DECL, "func_decl"],
func_decl);
                   }
               ;

formal_args    : ID (COMMA! ID)* {
                  #formal_args = #([FORMAL_ARGS, "formal_args"] ,
formal_args);
                  }
               ;




func_call          : ID LPAREN! (expr_list)? RPAREN! {
                           #func_call = #([FUNC_CALL, "func_call"],
func_call);
                     }
                   ;

break_statement   : BREAK^(SEMI!)
                          ;

continue_statement : CONTINUE^(SEMI!)
                          ;

return_statement   : RETURN^ (expr)? SEMI!
                          ;

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

for_statement  : "for"^ LPAREN! expr SEMI! (assign_expr)? SEMI! expr
RPAREN! statement
                      ;

forc_statement : "forc"^ coordinate ASSIGN! coordinate coordinate
statement
                      ;

forr_statement : "forr"^ coordinate ASSIGN! coordinate coordinate
statement
                      ;

while_statement: "while"^ LPAREN! expr RPAREN! statement
                      ;
```

```
expr_statement : expr {
                     #expr_statement = #([EXPR_STMT, "expr_stmt"],
expr_statement);
                     }
               ;

expr    : assign_expr (ASSIGN^ expr)?
               ;

expr_list      : expr (COMMA! expr)* {
                    #expr_list = #([EXPR_LIST, "expr_list"],
expr_list);
                    }
                     ;

assign_expr    : logic_term ( OR^ logic_term)*
                 ;

logic_term     : logic_factor ( AND^ logic_factor)*
                   ;

logic_factor   : (NOT^)? rel_expr
                     ;

rel_expr       : arith_expr ( (GE^ | LE^ | GT^| LT^ | EQ^ | NE^ )
arith_expr)?
                   ;

arith_expr     : arith_term ( (PLUS^ | MINUS^) arith_term)*
                     ;

arith_term     : arith_factor ( (MUL^ | DIV^ | MOD^) arith_factor)*
                     ;

arith_factor   :( p:PLUS^  { #p.setType(SIGN_PLUS);}
               |  m:MINUS^ { #m.setType(SIGN_MINUS);})? rvalue
                     ;

rvalue               : coordinate | REAL_CONSTANT | INT_CONSTANT | STR
| CHAR | ID | TRUE | FALSE
                     | LPAREN! expr RPAREN!
                     | func_call
                     ;

coordinate     : LBRAC! (expr) COLON! (expr) RBRAC!
                 { #coordinate = #([COORDINATE, "coordinate"] ,
coordinate); }
               ;

class L extends Lexer;

options { k = 2;
            testLiterals = false;
          exportVocab=NECoSL;
          }
```

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

ID
options         {
                testLiterals = true;
                }
                  : ( 'a'..'z' | 'A'..'Z' | '_' | '$')
                ( 'a'..'z' | 'A'..'Z' | '_' | '$' | '0'..'9' )*
                  ;

NUM            : (DIGIT)+ ('.' (DIGIT)+ {$setType(REAL_CONSTANT);} |
{$setType(INT_CONSTANT);})
               ;

STR             : '"'! ( 'a'..'z' | 'A'..'Z' | '0'..'9' | '_' | '$' |
'\\' | ':' | '=' | '#' | ' ' | '.' | '\n' | '\t')* '"'!
                ;

AND                     : "&&"
                        ;

LE                      : "<="
                        ;

SEMI                    : ';'
                        ;

OR                      : "||"
                        ;

GT                      : '>'
                        ;

LPAREN                  : '('
                        ;

ASSIGN                  :  '='
                        ;

GE                      : ">="
                        ;

RPAREN                  : ')'
                        ;

EQ                      : "=="
                        ;

LBRACE                  : '{'
                        ;

PLUS                    : '+'
                        ;

NOT                     : '!'
                        ;
```

```
RBRACE                  : '}'
                        ;

MINUS                   : '-'
                        ;

NE                      : "!="
                        ;

LBRAC                   : '['
                        ;

MUL                     : '*'
                        ;

MOD                     : '%'
                        ;

LT                      : '<'
                        ;

RBRAC                   : ']'
                        ;

DIV                     : '/'
                        ;

COLON                   : ':'
                        ;

COMMA                   : ','
                        ;

WS                      : (' ' | '\t')+ {$setType(Token.SKIP);};

NL                      : ('\n' {newline(); }
                        | ('\r' '\n') => '\r' '\n' {newline();}
                        | '\r') {$setType(Token.SKIP);}
                        ;

COMMENT                 : "/*" ( options { generateAmbigWarnings =
false; }                            : { LA(2) != '/' }? '*'
                        | "\r\n"                    { newline(); }
                        | ( '\r' | '\n' )       { newline(); }
                        | ~( '*'| '\r' | '\n' )
                        )*
                              "*/" { $setType(Token.SKIP); }
                        ;

CPPCOMMENT              : "//" ( ~('\n') )*
                             { $setType(Token.SKIP); }
                        ;
```

walker.g (Walker code: Constructs our own code from AST)

```
{
      import java.util.*;
}

class EcoslWalker extends TreeParser;

options{
        buildAST=true;
        importVocab = NECoSL;
        }


program returns [Program prog]
{  prog=null;Stmt s = null; }
      : #(PROGRAM { prog = new Program();}
          (s = statement
            {
                prog.addStmt(s);
          }
       )*
   );


statement returns [Stmt s]
{ s = null; Stmt s1, s2; Expr e1 = null, e2 = null, e3 = null;
java.util.Vector args=null; }
                             : #(FUNC_DECL
                                                    ID

      (args=formal_args)?
                                                    {
                                     s1 = new StmtBlock();
                                    }
                                    (s2=statement {
                                          ((StmtBlock)s1).addStmt(s2);
                                          }
                                       )*
                                                    {
                                                          s = new
FunctionDecl (#ID.getText(), args, s1);
                                                    }
                                                )
                             | #(STMT_BLOCK {
                                     s = new StmtBlock();
                                    }
                                       (s1=statement {
```

```
                                        ((StmtBlock)s).addStmt(s1);
                                        }
                                    )*
                                )
                            | #(EXPR_STMT e1 = expr
                                    {
                                        s = new ExprStmt(e1);
                                    }

                                )
                            | #("if" e1=expr s1=statement
(s2=statement { s = new Else(e1,s1,s2);} | { s = new If(e1,s1);} ))
                            | #("while" e1 = expr s1 = statement
                                { s = new While (e1, s1); }
                                )
                            | #("for" e1=expr e2=expr e3=expr
s1=statement
                                { s = new For(e1,e2,e3,s1); }
                                )
                            | #(BREAK    { s = new Break(); } )
                            | #(CONTINUE { s = new Continue(); } )
                            | #(RETURN (e1=expr)?  { s = new Return
(e1); } )
                            ;

formal_args returns [java.util.Vector formalArgs]
{formalArgs = null; }
                    : #(FORMAL_ARGS { formalArgs = new
java.util.Vector(); } (ID { formalArgs.add (new
Variable(#ID.getText())); })*);


expr_list returns [ Vector el ]
{ el = null; Expr e; }
                        :#(EXPR_LIST {
                                el = new Vector();
                            }
                            (e = expr {
                                    el.add(e);
                                    })*
                        )
                        ;

expr returns [Expr e]
{ e = null; Expr a,b; Vector v = new java.util.Vector();  v = null;}
                        : #(ASSIGN a=expr b=expr {e = new
AssignExpr(a,b); })
                        | #(NOT  a=expr        { e = new Not(a); })
                        | #(OR   a=expr b=expr { e = new Or(a,b); })
                        | #(AND  a=expr b=expr { e = new And(a,b); })
                        | #(GE   a=expr b=expr { e = new Ge(a, b); })
                        | #(LE   a=expr b=expr { e = new Le(a, b); })
                        | #(LT   a=expr b=expr { e = new Lt(a, b); })
                        | #(GT   a=expr b=expr { e = new Gt(a, b); })
                        | #(EQ   a=expr b=expr { e = new Eq(a, b); })
                        | #(NE   a=expr b=expr { e = new Ne(a, b); })
                        | #(PLUS a=expr b=expr { e = new Plus(a,b);})
```

```
                        | #(MINUS a=expr b=expr { e = new
Minus(a,b); })
                        | #(MUL   a=expr b=expr { e = new Mul(a,b); })
                        | #(DIV   a=expr b=expr { e = new Div(a,b); })
                        | #(MOD   a=expr b=expr { e = new Mod(a,b); })
                        | #(COORDINATE  a=expr b=expr { e = new
Coordinate(a, b); } )
                        | INT_CONSTANT {  e = new
IntConstant(#INT_CONSTANT.getText(), Type.Int); }
                        | REAL_CONSTANT {  e = new
FloatConstant(#REAL_CONSTANT.getText(), Type.Float); }
                        | STR  {  e = new
StringConstant(#STR.getText(), Type.Str); }
                        | ID   { e = new Variable(#ID.getText()); }
                        | #(FUNC_CALL ID (v=expr_list)? { e = new
FuncCall(#ID.getText(), v); })
                        ;
```

Program.java :Main node for any program

```
import java.util.Vector;


public class Program {

      Vector<Stmt> stmts = new Vector<Stmt>();
      Vector<FunctionDecl> fndefs = new Vector<FunctionDecl>();

      public Program() {
            // TODO Auto-generated constructor stub
      }

      public void execute(Env env){
            int j = stmts.size();
            for (int i= 0;i < j;++i)
                  ((Stmt)stmts.get(i)).execute(env);
      }
      public void addStmt (Stmt stmt)
      {
            stmts.addElement(stmt);
      }
}
```

EcoSL.java: Main interpreter class file

```
//import java.io.DataInputStream;
import java.io.*;
import antlr.CommonAST;


/* This is the Interpreter */

class EcoSL {
```

```
    public static void main(String[] args) {
        try {
        //  String filename = args[0];
            String filename = "C:\\Users\\som\\Documents\\Java\\Java
Programs\\J2EE\\EcoSL_CVS\\src\\test"; //args[0];
        //  String filename = "C:\\Users\\som\\Documents\\Fall
2007\\Programming Language and
Translators\\Project\\Test_scenarios\\UserDefineSpreadsheetfunctions\\B
asic_coordinates.txt";

            L lexer = new L(new FileInputStream(filename));
            P parser = new P(lexer);
            parser.program(); // It is not required when the walker is
included

        //  Get the AST from the parser
            CommonAST parseTree = (CommonAST)parser.getAST();

        //  Print the AST in a human-readable format
        //  System.out.println(parseTree.toStringList());

        //  Open a window in which the AST is displayed graphically
        //  ASTFrame frame = new ASTFrame("AST from the ECoSL parser",
parseTree);
        //  frame.setVisible(true);

            EcoslWalker walker = new EcoslWalker();
            Program prog = walker.program(parseTree);

          //  AST results = walker.getAST();
          //  DumpASTVisitor visitor = new DumpASTVisitor();
          //  visitor.visit(results);


            Env env = new Env();
            env.funcT.put("print", new PrintFunc());
            env.funcT.put("println", new PrintlnFunc());
            env.funcT.put("avg", new AverageFunc());
            env.funcT.put("sigma", new SumFunction());
            prog.execute(env);

            System.exit(1);
        } catch(Exception e) {
            e.printStackTrace();
        }
    }
}
```

Symbol Table and Environment classes

Env.java

```
import java.util.*;

public class Env {
```

```
	Hashtable<String, Value> cordsys ;
	Hashtable<String, Func> funcT ;

	boolean breakFlag;
	boolean continueFlag;
	boolean returnFlag;
	Value returnValue;

	//    protected Env prev;

	Table presentTable;// = new Table();

	public Env (){
		//presentTable = global;
		presentTable = new Table();
		cordsys = new Hashtable<String, Value>();
		funcT = new Hashtable<String, Func>();
	}

	public Table getPresentTable() {
		return presentTable;
	}

	public void setPresentTable(Table t)	{
		this.presentTable = t;
	}

	// enter new scope
	public void createNewScope() {
		setPresentTable(new Table(this.getPresentTable()));
	}

	public void removeScope() {
		presentTable = presentTable.prev;
	}
	public void putcordinate(String in_c, Value in_v){
		this.cordsys.put(in_c, in_v);
	}

	public Value getcordinate(String c){
		return (Value)this.cordsys.get(c);
	}

	public void putFunc(String name, Func f){
		this.funcT.put(name, f);
	}

	public Func getFunc(String name){
		return this.funcT.get(name);

	}
}
```

Table.java (Symbol table implemented as hash Table with linked list structure)

```
import java.util.*;
public class Table {
	private Hashtable<String,Value> symtab;

	protected Table prev;


	public Table()
	{
		this(null);
	}

	public Table(Table prev){
		try {
			symtab = new Hashtable<String,Value>();
			this.prev = prev;
		} catch (Exception ex) {
			System.out.println(ex.getMessage());
		}
	}


	public void putIdInSymTab(String Str_key_Symboltable, Value sv){
		try {
			symtab.put(Str_key_Symboltable, sv);
		} catch (Exception ex) {
			System.out.println(ex.getMessage());
		}

	}

	public boolean isInSymTab(String Str_key_Symboltable){

		return symtab.containsKey(Str_key_Symboltable)
			|| (prev != null &&
prev.isInSymTab(Str_key_Symboltable));
	}

	public Value getIdFromSymTab(String Str_key_Symboltable){
		try {
			// get the value from the current table
			Value ret = (Value)symtab.get(Str_key_Symboltable);
			// if its not there, ask previous table
			if (ret == null)
				if (prev != null) return
prev.getIdFromSymTab(Str_key_Symboltable);
				else return null;
			else
				return ret;
		} catch (Exception ex) {
			System.out.println(ex.getMessage());
			return null;
		}
```

```
	}

}
```

Control Statements: Return, Break, Continue

Break.java

```
public class Break implements Stmt{

	public Break() {
	}
	public void execute(Env env){
		env.breakFlag = true;
	}
}
```

Return.java

```
public class Return implements Stmt{

	private Expr e;

	public Return(Expr e) {
		this.e = e;
	}
	public void execute(Env env){
		env.returnFlag = true;
		env.returnValue = e.evaluate(env);
	}
}
```

Continue.java

```
public class Continue implements Stmt{

	public Continue() {

	}
	public void execute(Env env){
		env.continueFlag = true;
	}
}
```

Coordinate to represent cells of a spreadsheet

Coordinate.java

```
public class Coordinate implements Expr{
	Expr x;
	Expr y;
	Value v;

	public Coordinate(Expr a, Expr b) {
		x = a;
		y  = b;
```

```
	}

	public Value evaluate(Env env) {
		try {
			Value vx = x.evaluate(env);
			Value vy = y.evaluate(env);

			if( vx.type == Type.Int && vy.type == Type.Int ) {
				String s = vx.val.toString() + ":"
+vy.val.toString();
				v = env.getcordinate(s);
				return v;
			}
			else {
				System.out.println("Error: Incorrect values
inside the coordinate");
				System.exit(1);
			}
			return null;
		}
		catch(Exception e) {
			System.out.println(e.getMessage());
			return null;
		}
	}
}
```

Variable.java

```
public class Variable implements Expr{

	String name;
	Value v;

	public Variable(String str) {
		name = str;
	}
	public Value evaluate(Env env) {
		try {
			v = env.presentTable.getIdFromSymTab(name);
			if ( null == v ) {
				System.out.print("\nError: Variable " + name +
" not defined.");
				System.exit(1);
			}
			return v;
		} catch ( Exception ex ) {
			System.out.print(ex.getMessage());
			return null;
		}
	}
}
```

Type checking through Type.java

```
public class Type {

	public String name = "";

	public Type(String s) {
		name = s;
	}
	public static final Type
		Int = new Type("int"), Float = new Type("float"),
		Bool = new Type("boolean"),   Str = new Type ("str"),
		Crdnt = new Type("coordinate");

	public static boolean numeric(Type p) {
		return p == Type.Int || p == Type.Float;
	}

	public static boolean boolchk(Type p) {
		return p == Type.Bool;
	}

	public static Type max(Type p1, Type p2) {
		if (!numeric(p1) || !numeric(p2))
			return null;
		else if (p1 == Type.Float || p2 == Type.Float)
			return Type.Float;
		else
			return Type.Int;

	}
}
```

Value stored in Symbol table using this class. This class holds value and type of variable constants and coordinates

```
public class Value {

	Object val;
	Type type;

	public Value() {

	}
	public Value(Object str, Type t) {
		val = str;
		type = t;
	}

	public Object getValue(Env env) {
		try{
			return this.val;
		} catch (Exception ex) {
```

```
                System.out.print(ex.getMessage());
                return null;
            }
        }
        public Object getType(Env env) {
            try{
                return this.type;
            } catch (Exception ex) {
                System.out.print(ex.getMessage());
                return null;
            }
        }
        public String toString() {
            return (val.toString());
        }

}
```

IntConstant.java

```
public class IntConstant implements Expr{

        Value v;

        public IntConstant(Object obj, Type t) {
            v= new Value(obj,t);

        }
        public Value evaluate(Env env) {
                return v;
        }
}
```

FloatConstant.java

```
public class FloatConstant implements Expr{

        Value v;

        public FloatConstant(Object obj, Type t) {
            v = new Value (obj, t );
        }

        public Value evaluate(Env env) {
            return v;
        }

}
```

StringConstant.java

```
public class StringConstant implements Expr{
```

```
	Value v;

	public StringConstant(Object obj, Type t) {
		v = new Value (obj, t );
	}

	public Value evaluate(Env env) {
		return v;
	}
}
```

BooleanConstant.java

```
public class BooleanConstant implements Expr{

	Value v;

	public BooleanConstant(Object obj, Type t) {
		v.val = obj;
		v.type = t;
	}
	public Value evaluate(Env env) {
		return v;
	}
}
```

Expr.java

```
public interface Expr {

	public Value evaluate(Env env);

}
```

Expr_list.java

```
import java.util.Vector;


public class Expr_list implements Expr {

	Vector<Expr> exprs = new Vector<Expr>();

	public Expr_list() {
		// TODO Auto-generated constructor stub
	}
	public Value evaluate(Env env) {

		System.out.print("\nInside Expr_list\n");
		Value v = new Value();
		return v;
	}
	public void addExprList(Expr expr)
	{
		exprs.addElement(expr);
	}
```

```
}
```

ExprStmt.java

```
public class ExprStmt implements Stmt{

	Expr x;

	public ExprStmt(Expr y) {
		x=y;
	}

	public void execute(Env env) {
		try {
			Value  k = x.evaluate(env);
			if (k == null) {
				System.out.println("Error: Expression failed");
			}
		} catch(Exception e) {
			System.out.println("Error: Expression failed");
			System.out.println(e.getMessage());
		}
	}

	public Value evaluate(Env env) {

		try {
			Value  k = x.evaluate(env);
			if (k == null) {
				System.out.println("Error: Expression failed");
			}
			return k;
		} catch(Exception e) {
			System.out.println("Error: Expression failed");
			System.out.println(e.getMessage());
			return null;
		}
	}
}
```

AssignExpr.java

```
public class AssignExpr implements Expr{

	Expr right;
	Expr left;
	Value ret;

	public AssignExpr(Expr x1, Expr x2){
		left = x1;
		right = x2;

	}
```

```
	public void execute(Env env) {
		try {
			Value  k = this.evaluate(env);
			if (k == null) {
				System.out.println("Error: Expression failed");
			}
		} catch(Exception e) {
			System.out.println("Error: Expression failed");
			System.out.println(e.getMessage());
		}
	}

	public Value evaluate(Env env) {
		try {
			Value rv = right.evaluate(env);
			Table tb = env.getPresentTable();
			if (left instanceof Variable) {
				Variable v = (Variable)left;
				String idName = v.name;
				ret = new Value(rv.val, rv.type);
			    tb.putIdInSymTab(idName, rv);
			}
			else if (left instanceof Coordinate) {

					Coordinate c = (Coordinate)left;
					if ( c == null) {
					System.out.println("Error: Left hand side
should be variable OR COORDINATE!!");
					System.exit(1);
					}
					else{

						Value vx = c.x.evaluate(env);
						Value vy = c.y.evaluate(env);

						if( vx.type == Type.Int && vy.type
== Type.Int ) {
							ret = new Value(rv.val,
Type.Crdnt);
							String s = vx.val.toString()
+ ":" +vy.val.toString();
							env.putcordinate( s, rv);

						}
					}

			}
			else {
				System.out.println("Error: Incorrect use of
Assignment ");
				System.exit(1);
			}

			return ret;
		}
		catch(Exception e) {
			System.out.println(e.getMessage());
```

```
                return null;
            }
        }
}
```

Stmt.java

```
public interface Stmt {

      public void execute(Env env);
      public String toString();
}
```

StmtBlock.java

```
import java.util.*;

public class StmtBlock implements Stmt{

      Vector<Stmt> stmts = new Vector<Stmt>();

      public StmtBlock() {
            // TODO Auto-generated constructor stub
      }
      public void execute(Env env){
            // create new environment
            for (int i= 0;i < stmts.size();++i)
            {
                  ((Stmt)stmts.get(i)).execute(env);
                  if (env.breakFlag || env.continueFlag ||
env.returnFlag) return;
            }
      }
      public void addStmt(Stmt stmt)
      {
            stmts.addElement(stmt);
      }
}
```

Loops

For.java

```
public class For implements Stmt{

      Expr expr1;
      Expr expr2;
      Expr expr3;
      Stmt stmt;

      public For( Expr e1, Expr e2, Expr e3, Stmt s ) {
            expr1 = e1;
            expr2 = e2;
            expr3 = e3;
            stmt = s;
      }
```

```
	public void execute(Env env){
		Value cond = new Value();
		if ( !(expr1 instanceof AssignExpr)) {
			System.out.println("Error: Inside for statement,
first statement should be assignment!!");
			System.exit(1);
		}
		if ( !(expr3 instanceof AssignExpr)) {
			System.out.println("Error: Inside for statement,
third statement should be assignment!!");
			System.exit(1);
		}

		Value v1 = expr1.evaluate(env);

		try{
			while(true) {
				cond = expr2.evaluate(env);
				if( cond.type != Type.Bool  ){
					System.out.print("\nError: Boolean value
expected in the condition of for \n");
					System.exit(1);
				}
				if( ((Boolean)cond.val).toString() == "true" ){

					stmt.execute(env);
					Value v2 = expr3.evaluate(env);
					if (env.returnFlag) return;
					if (env.breakFlag)
					{
						env.breakFlag = false;
						return;
					}
					if (env.continueFlag)
					{
						env.continueFlag = false;
					}
				}
				else
					break;
			}
		}catch (Exception ex) {
			System.out.print(ex.getMessage());
		}
	}
}
```

While.java

```
public class While implements Stmt{

	Expr expr;
	Stmt stmt;
	public While(Expr e, Stmt s) {
		expr = e;
```

```
            stmt = s;
      }
      public void execute(Env env){
            try{
                  Value cond = new Value();
                  while(true) {
                        cond = expr.evaluate(env);
                        if( cond.type != Type.Bool  ){
                              System.out.print("\nError: Boolean value
expected inside while condition\n");
                              System.exit(1);
                        }
                        if( ((Boolean)cond.val).toString() == "true" ){

                                    stmt.execute(env);
                                    if (env.returnFlag) return;
                                    if (env.breakFlag)
                                    {
                                          env.breakFlag = false;
                                          return;
                                    }
                                    if (env.continueFlag)
                                    {
                                          env.continueFlag = false;
                                    }
                        }
                        else
                              break;
                  }
            }catch (Exception ex) {
                        System.out.print(ex.getMessage());
            }
      }
}
```

Function related classes (for in built functions)

Func.java:

```
import java.util.Vector;


public class Func {

      Vector<Variable> args;
      String name;
      Stmt stmts;

      protected Func(){}

      public Func(String n, Vector<Variable> t, Stmt s) {
            name = n;
            args = t;
            stmts = s;
      }
```

```
	public Value execute (Vector<Expr> args, Env env)
	{
		int a,b;
		if(args == null) {
			b = 0;
		}else {
			b = args.size();
		}

		if(this.args == null) {
			a = 0;
		}else {
			a = this.args.size();
		}
		if (a != b) {
			System.out.println("Error: Incorrect parameters for
the function call " + name);
			System.exit(1);
		}

		Value [] x = new Value[a];
		for (int i=0; i< b; i++) {
			x[i] = args.elementAt(i).evaluate(env);
		}
		// push arg in ST
		env.createNewScope();
		Table tb = env.getPresentTable();

		for (int i = 0; i < a; i++) {
			tb.putIdInSymTab(this.args.elementAt(i).name, x[i]);
		}
		this.stmts.execute(env);
		env.removeScope();

		if (env.returnFlag)
		{
			env.returnFlag = false;
			return env.returnValue;
		}

		return null;
	}
}
```

FuncCall.java

```
import java.util.Vector;

public class FuncCall implements Expr{

	Vector<Expr> args;
	String name;
	Value ret;
	public FuncCall(String n, Vector<Expr> t) {
		name = n;
		args = t;
```

```
	}

	public Value evaluate(Env env){
		try {
			Func f= (Func)env.getFunc(name);
			Value r= f.execute(this.args, env);
			if (r == null){
				return new Value("true", Type.Bool);
			}
			else return r;
		} catch (Exception ex) {
			System.out.println(ex.getMessage());
			return null;
		}

	}

}
```

FunctionDecl.java

```
import java.util.Vector;


public class FunctionDecl implements Stmt{

	Func fn;
	String name;

	public FunctionDecl(String funcname, Vector<Variable> args, Stmt
s) {
		fn = new Func(funcname, args, s);
	}

	public void execute(Env env){
		try{
			env.putFunc(fn.name, fn);
		} catch (Exception ex) {
		System.out.println(ex.getMessage());
		}
	}
}
```

Conditional Statement if

If.java

```
public class If implements Stmt{
	Expr expr;
	Stmt stmt;

	public If(Expr e, Stmt s) {
	expr = e;
	stmt = s;
	}
```

```
	public void execute(Env env){
		try{
			Value v = expr.evaluate(env);
			if( v.type != Type.Bool  ) {
				System.out.print("\nError: Boolean value
expected inside if condition\n");
				System.exit(1);
			}
			else {
				if( ((Boolean)v.val).toString() == "true" ){

					stmt.execute(env);
				}
			}
		}catch (Exception ex) {
			System.out.print(ex.getMessage());
		}
	}
}
```

Else.java

```
public class Else implements Stmt{
	Expr expr;
	Stmt stmt1;
	Stmt stmt2;
	public Else(Expr e, Stmt s1, Stmt s2) {
		expr = e;
		stmt1 = s1;
		stmt2 = s2;
	}
	public void execute(Env env){
		try{
			Value v = expr.evaluate(env);
			if( v.type != Type.Bool  ) {
				System.out.print("\nError: Boolean value
expected inside if else condition\n");
				System.exit(1);
			}
			else {
				if( ((Boolean)v.val).toString() == "true" ){

					stmt1.execute(env);
				}
				else
					stmt2.execute(env);
			}
		}catch (Exception ex) {
			System.out.print(ex.getMessage());
		}
	}
}
```

Arithmetic operators
Plus.java

```
public class Plus implements Expr{

	Expr left;
	Expr right;
	Value ret;

	public Plus(Expr x1, Expr x2){

		left = x1;
		right = x2;
		ret = new Value();
	}

	public Value evaluate(Env env) {

		try{
			Value vr = right.evaluate(env);
			Value vl = left.evaluate(env);

			Type tmp= Type.max(vr.type, vl.type);

			if (tmp == null) {
				if ( (vr.type == Type.Str) && (vl.type ==
Type.Str) ) {
					String a = vl.val.toString() +
vr.val.toString();
					ret.type = Type.Str;
					ret.val = a;

				} else {
					System.out.print("\nError: Type mismatch
in + : Operation undefined for " + vl.type.name  + " and " +
vr.type.name);
					System.exit(1);
				}
			}

			if ( tmp == Type.Int ) {
				ret.type = Type.Int;
				ret.val = Integer.parseInt(vl.val.toString()) +
Integer.parseInt(vr.val.toString());
			}
			if ( tmp == Type.Float ) {
				ret.type = Type.Float;
				ret.val = Float.parseFloat(vl.val.toString()) +
Float.parseFloat(vr.val.toString());
			}

			return ret;

		}catch (Exception ex) {
			System.out.print(ex.getMessage());
			return null;
		}
	}
```

```
}
```

Div.java (for purpose of Division)

```
public class Div implements Expr{

	Expr left;
	Expr right;
	Value ret;

	public Div(Expr x1, Expr x2){

		left = x1;
		right = x2;
		ret = new Value();
	}

	public Value evaluate(Env env) {
		try{
			Value vr = right.evaluate(env);
			Value vl = left.evaluate(env);

			Type tmp= Type.max(vr.type, vl.type);

			if (tmp == null) {
					System.out.print("\nError: Type mismatch
in / : Operation undefined for " + vl.type.name  + " and " +
vr.type.name);
					System.exit(1);
			}

			if ( tmp == Type.Int ) {
				ret.type = Type.Int;
				ret.val = Integer.parseInt(vl.val.toString()) /
Integer.parseInt(vr.val.toString());
			}
			if ( tmp == Type.Float ) {
				ret.type = Type.Float;
				ret.val = Float.parseFloat(vl.val.toString()) /
Float.parseFloat(vr.val.toString());
			}

			return ret;

		}catch (Exception ex) {
			System.out.print(ex.getMessage());
			return null;
		}
	}
}
```

Mul.java (Multiply)

```
public class Mul implements Expr{

	Expr left;
	Expr right;
	Value ret;

	public Mul(Expr x1, Expr x2){

		left = x1;
		right = x2;
		ret = new Value();
	}

	public Value evaluate(Env env) {
		try{
			Value vr = right.evaluate(env);
			Value vl = left.evaluate(env);

			Type tmp= Type.max(vr.type, vl.type);

			if (tmp == null) {
					System.out.print("\nError: Type mismatch
in * : Operation undefined for " + vl.type.name  + " and " +
vr.type.name);
					System.exit(1);
			}

			if ( tmp == Type.Int ) {
				ret.type = Type.Int;
				ret.val = Integer.parseInt(vl.val.toString()) *
Integer.parseInt(vr.val.toString());
			}
			if ( tmp == Type.Float ) {
				ret.type = Type.Float;
				ret.val = Float.parseFloat(vl.val.toString()) *
Float.parseFloat(vr.val.toString());
			}

			return ret;

		}catch (Exception ex) {
			System.out.print(ex.getMessage());
			return null;
		}
	}
}
```

Mod.java (Mod operator)

```
public class Mod implements Expr{

	Expr left;
	Expr right;
	Value ret;
```

```
	public Mod(Expr x1, Expr x2){

		left = x1;
		right = x2;
		ret = new Value();
	}

	public Value evaluate(Env env) {
		try{
			Value vr = right.evaluate(env);
			Value vl = left.evaluate(env);

			Type tmp= Type.max(vr.type, vl.type);

			if (tmp == null) {
					System.out.print("\nError: Type mismatch
in % : Operation undefined for " + vl.type.name  + " and " +
vr.type.name);
					System.exit(1);
			}

			if ( tmp == Type.Int ) {
				ret.type = Type.Int;
				ret.val = Integer.parseInt(vl.val.toString()) %
Integer.parseInt(vr.val.toString());
			}
			if ( tmp == Type.Float ) {
				ret.type = Type.Float;
				ret.val = Float.parseFloat(vl.val.toString()) %
Float.parseFloat(vr.val.toString());
			}

			return ret;

		}catch (Exception ex) {
			System.out.print(ex.getMessage());
			return null;
		}
	}
}
```

Minus.java (For subtraction)

```
public class Minus implements Expr{

	Expr left;
	Expr right;
	Value ret;

	public Minus(Expr x1, Expr x2){

		left = x1;
		right = x2;
		ret = new Value();
	}
```

```
	public Value evaluate(Env env) {
		try{
			Value vr = right.evaluate(env);
			Value vl = left.evaluate(env);

			Type tmp= Type.max(vr.type, vl.type);

			if (tmp == null) {
					System.out.print("\nError: Type mismatch
in - : Operation undefined for " + vl.type.name  + " and " +
vr.type.name);
					System.exit(1);
			}

			if ( tmp == Type.Int ) {
				ret.type = Type.Int;
				ret.val = Integer.parseInt(vl.val.toString()) -
Integer.parseInt(vr.val.toString());
			}
			if ( tmp == Type.Float ) {
				ret.type = Type.Float;
				ret.val = Float.parseFloat(vl.val.toString()) -
Float.parseFloat(vr.val.toString());
			}

			return ret;

		}catch (Exception ex) {
			System.out.print(ex.getMessage());
			return null;
		}
	}
}
```

Relational operators:

Not.java (!)

```
public class Not implements Expr{

	Expr right;

	public Not(Expr x1){

		right = x1;
	}

	public Value evaluate(Env env) {
		Value ret = new Value();
		try {
			Value vr = right.evaluate(env);
			if( vr.type != Type.Bool ) {
```

```
                        System.out.print("\nError: Boolean is expected
with ! \n");
                        System.exit(1);
                  }

                  if ( Boolean.parseBoolean(vr.val.toString()) ==
true )
                        ret.val = Boolean.valueOf("false");
                  else
                        ret.val = Boolean.valueOf("true");

                  ret.type = Type.Bool;
                  return ret;
            } catch (Exception ex) {
                  System.out.print(ex.getMessage());
                  return null;
            }

      }

}
```

Eq.java (==)

```
public class Eq implements Expr{

      Expr left;
      Expr right;
      Value ret;

      public Eq(Expr x1, Expr x2){

            left = x1;
            right = x2;
            ret = new Value();
      }

      public Value evaluate(Env env) {
            try{
                  Value vr = right.evaluate(env);
                  Value vl = left.evaluate(env);
                  if( ( vr.type == Type.Bool ) && ( vl.type ==
Type.Bool ) ) {
                        ret.type = Type.Bool;
                        if ( Boolean.parseBoolean(vr.val.toString()) ==
Boolean.parseBoolean(vl.val.toString()) )
                              ret.val = Boolean.valueOf("true");
                        else
                              ret.val = Boolean.valueOf("false");
                  } else if( ( vr.type == Type.Str ) && ( vl.type ==
Type.Str ) ) {
                        ret.type = Type.Bool;
                        if
( (vr.val.toString()).compareTo(vl.val.toString()) == 0 )
```

```
                              ret.val = Boolean.valueOf("true");
                        else
                              ret.val = Boolean.valueOf("false");
                  } else {
                        Type tmp= Type.max(vr.type, vl.type);
                        if (tmp == null) {
                              System.out.print("\nError: Type mismatch
in / : Operation undefined for " + vl.type.name  + " and " +
vr.type.name);
                              System.exit(1);
                        }
                        ret.type = Type.Bool;
                        if ( tmp == Type.Int ) {
                              if ( Integer.parseInt(vr.val.toString())
== Integer.parseInt(vl.val.toString()) )
                                    ret.val = Boolean.valueOf("true");
                              else
                                    ret.val = Boolean.valueOf("false");
                        }
                        if ( tmp == Type.Float ) {
                              if ( Float.parseFloat(vr.val.toString())
== Float.parseFloat(vl.val.toString()) )
                                    ret.val = Boolean.valueOf("true");
                              else
                                    ret.val = Boolean.valueOf("false");
                        }
                  }
                  return ret;

            }catch (Exception ex) {
                  System.out.print(ex.getMessage());
                  return null;
            }
      }

}
```

Ge.java (greater than equal to)

```
public class Ge implements Expr{

      Expr left;
      Expr right;
      Value ret;

      public Ge(Expr x1, Expr x2){

            left = x1;
            right = x2;
            ret = new Value();
      }

      public Value evaluate(Env env) {
            try{
                  Value vr = right.evaluate(env);
```

```
                  Value vl = left.evaluate(env);
                  ret.type = Type.Bool;

                  Type tmp= Type.max(vr.type, vl.type);
                  if (tmp == null) {

                        if ( (vr.type == Type.Str) && (vl.type ==
Type.Str) ) {

                              String a = vl.val.toString();
                              String b = vr.val.toString();

                              if ( a.length() >= b.length() ) {
                                    ret.val = Boolean.valueOf("true");
                              } else {
                                    ret.val = Boolean.valueOf("false");
                              }

                        } else {
                              System.out.print("\nError: Type mismatch
in >= : Operation undefined for " + vl.type.name  + " and " +
vr.type.name);
                              System.exit(1);
                        }
                  }

                  if ( tmp == Type.Int ) {
                        if ( Integer.parseInt(vl.val.toString()) >=
Integer.parseInt(vr.val.toString()) )
                              ret.val = Boolean.valueOf("true");
                        else
                              ret.val = Boolean.valueOf("false");
                  }
                  if ( tmp == Type.Float ) {
                        if ( Float.parseFloat(vl.val.toString()) >=
Float.parseFloat(vr.val.toString()) )
                              ret.val = Boolean.valueOf("true");
                        else
                              ret.val = Boolean.valueOf("false");
                  }

                  return ret;

            }catch (Exception ex) {
                  System.out.print(ex.getMessage());
                  return null;
            }
      }
}
```

Gt.java (greater than)

```
public class Gt implements Expr{
```

```
	Expr left;
	Expr right;
	Value ret;

	public Gt(Expr x1, Expr x2){
		left = x1;
		right = x2;
		ret = new Value();
	}

	public Value evaluate(Env env) {
		try{
			Value vr = right.evaluate(env);
			Value vl = left.evaluate(env);
			ret.type = Type.Bool;

			Type tmp= Type.max(vr.type, vl.type);
			if (tmp == null) {

				if ( (vr.type == Type.Str) && (vl.type ==
Type.Str) ) {

					String a = vl.val.toString();
					String b = vr.val.toString();

					if ( a.length() > b.length() ) {
						ret.val = Boolean.valueOf("true");
					} else {
						ret.val = Boolean.valueOf("false");
					}

				} else {
					System.out.print("\nError: Type mismatch
in > : Operation undefined for " + vl.type.name  + " and " +
vr.type.name);
					System.exit(1);
				}
			}

			if ( tmp == Type.Int ) {
				if ( Integer.parseInt(vl.val.toString()) >
Integer.parseInt(vr.val.toString()) )
					ret.val = Boolean.valueOf("true");
				else
					ret.val = Boolean.valueOf("false");
			}
			if ( tmp == Type.Float ) {
				if ( Float.parseFloat(vl.val.toString()) >
Float.parseFloat(vr.val.toString()) )
					ret.val = Boolean.valueOf("true");
				else
					ret.val = Boolean.valueOf("false");
			}

			return ret;

		}catch (Exception ex) {
```

```
                System.out.print(ex.getMessage());
                return null;
            }
      }
}
```

Less Than
Lt.java

```
public class Lt implements Expr{

      Expr left;
      Expr right;
      Value ret;

      public Lt(Expr x1, Expr x2){

            left = x1;
            right = x2;
            ret = new Value();
      }

      public Value evaluate(Env env) {
            try{
                  Value vr = right.evaluate(env);
                  Value vl = left.evaluate(env);
                  ret.type = Type.Bool;

                  Type tmp= Type.max(vr.type, vl.type);
                  if (tmp == null) {

                        if ( (vr.type == Type.Str) && (vl.type ==
Type.Str) ) {

                              String a = vl.val.toString();
                              String b = vr.val.toString();

                              if ( a.length() < b.length() ) {
                                    ret.val = Boolean.valueOf("true");
                              } else {
                                    ret.val = Boolean.valueOf("false");
                              }

                        } else {
                              System.out.print("\nError: Type mismatch
in < : Operation undefined for " + vl.type.name  + " and " +
vr.type.name);
                              System.exit(1);
                        }
                  }
```

```
                if ( tmp == Type.Int ) {
                        if ( Integer.parseInt(vl.val.toString()) <
Integer.parseInt(vr.val.toString()) )
                                ret.val = Boolean.valueOf("true");
                        else
                                ret.val = Boolean.valueOf("false");
                }
                if ( tmp == Type.Float ) {
                        if ( Float.parseFloat(vl.val.toString()) <
Float.parseFloat(vr.val.toString()) )
                                ret.val = Boolean.valueOf("true");
                        else
                                ret.val = Boolean.valueOf("false");
                }

                return ret;

            }catch (Exception ex) {
                System.out.print(ex.getMessage());
                return null;
            }
      }
}
```

Le.java

```
public class Le implements Expr{

      Expr left;
      Expr right;
      Value ret;

      public Le(Expr x1, Expr x2){
            left = x1;
            right = x2;
            ret = new Value();
      }

      public Value evaluate(Env env) {
            try{
                Value vr = right.evaluate(env);
                Value vl = left.evaluate(env);
                ret.type = Type.Bool;

                Type tmp= Type.max(vr.type, vl.type);
                if (tmp == null) {

                        if ( (vr.type == Type.Str) && (vl.type ==
Type.Str) ) {

                                String a = vl.val.toString();
                                String b = vr.val.toString();
```

```
                                if ( a.length() <= b.length() ) {
                                      ret.val = Boolean.valueOf("true");
                                } else {
                                      ret.val = Boolean.valueOf("false");
                                }

                          } else {
                                System.out.print("\nError: Type mismatch
in <= : Operation undefined for " + vl.type.name  + " and " +
vr.type.name);
                                System.exit(1);
                          }
                    }

                    if ( tmp == Type.Int ) {
                          if ( Integer.parseInt(vl.val.toString()) <=
Integer.parseInt(vr.val.toString()) )
                                ret.val = Boolean.valueOf("true");
                          else
                                ret.val = Boolean.valueOf("false");
                    }
                    if ( tmp == Type.Float ) {
                          if ( Float.parseFloat(vl.val.toString()) <=
Float.parseFloat(vr.val.toString()) )
                                ret.val = Boolean.valueOf("true");
                          else
                                ret.val = Boolean.valueOf("false");
                    }

                    return ret;

              }catch (Exception ex) {
                    System.out.print(ex.getMessage());
                    return null;
              }
        }
}
```

Ne.java : NotEqual To

```
public class Ne implements Expr{
      Expr left;
      Expr right;
      Value ret;

      public Ne(Expr x1, Expr x2){

            left = x1;
            right = x2;
            ret = new Value();
      }

      public Value evaluate(Env env) {
            try{
```

```
                  Value vr = right.evaluate(env);
                  Value vl = left.evaluate(env);
                  if( ( vr.type == Type.Bool ) && ( vl.type ==
Type.Bool ) ) {
                        ret.type = Type.Bool;
                        if ( Boolean.parseBoolean(vr.val.toString()) !=
Boolean.parseBoolean(vl.val.toString()) )
                              ret.val = Boolean.valueOf("true");
                        else
                              ret.val = Boolean.valueOf("false");
                  } else if( ( vr.type == Type.Str ) && ( vl.type ==
Type.Str ) ) {
                        ret.type = Type.Bool;
                        if
( (vr.val.toString()).compareTo(vl.val.toString()) == 0 )
                              ret.val = Boolean.valueOf("false");
                        else
                              ret.val = Boolean.valueOf("true");
                  } else {
                        Type tmp= Type.max(vr.type, vl.type);
                        if (tmp == null) {
                              System.out.print("\nError: Type mismatch
in != : Operation undefined for " + vl.type.name  + " and " +
vr.type.name);
                              System.exit(1);
                        }
                        ret.type = Type.Bool;
                        if ( tmp == Type.Int ) {
                              if
( Integer.parseInt(vr.val.toString()) !=
Integer.parseInt(vl.val.toString()) )
                                    ret.val = Boolean.valueOf("true");
                              else
                                    ret.val = Boolean.valueOf("false");
                        }
                        if ( tmp == Type.Float ) {
                              if
( Float.parseFloat(vr.val.toString()) !=
Float.parseFloat(vl.val.toString()) )
                                    ret.val = Boolean.valueOf("true");
                              else
                                    ret.val = Boolean.valueOf("false");
                        }

                  }
                  return ret;

            }catch (Exception ex) {
                  System.out.print(ex.getMessage());
                  return null;
            }
      }
}
```

Logical Operators: AND, OR

And.java

```
public class And implements Expr{

	Expr left;
	Expr right;
	Value ret;

	public And(Expr x1, Expr x2){

		left = x1;
		right = x2;
		ret = new Value();
	}

	public Value evaluate(Env env) {

		try{
			Value vr = right.evaluate(env);
			Value vl = left.evaluate(env);

			if( ( vr.type != Type.Bool ) || ( vl.type !=
Type.Bool ) ) {
				System.out.print("\nType mismatch in && :
Operation undefined for " + vl.type.name  + " and " + vr.type.name);
				System.exit(1);
			}

			if( ( vr.type == Type.Bool ) && ( vl.type ==
Type.Bool ) ) {
				ret.type = Type.Bool;
				if ( ( Boolean.parseBoolean(vr.val.toString())
== true ) && ( Boolean.parseBoolean(vl.val.toString()) == true ) )
					ret.val = Boolean.valueOf("true");
				else
					ret.val = Boolean.valueOf("false");
			}
			return ret;

		}catch (Exception ex) {
			System.out.print(ex.getMessage());
			return null;
		}
	}
}
```

Or.java

```
public class Or implements Expr{

	Expr left;
	Expr right;
	Value ret;

	public Or(Expr x1, Expr x2){
```

```
            left = x1;
            right = x2;
            ret = new Value();
      }

      public Value evaluate(Env env) {
            try{
                  Value vr = right.evaluate(env);
                  Value vl = left.evaluate(env);

                  if( ( vr.type != Type.Bool ) || ( vl.type !=
Type.Bool ) ) {
                        System.out.print("\nError: Type mismatch in
|| : Operation undefined for " + vl.type.name  + " and " +
vr.type.name);
                        System.exit(1);
                  }

                  if( ( vr.type == Type.Bool ) && ( vl.type ==
Type.Bool ) ) {
                        ret.type = Type.Bool;
                        if ( ( Boolean.parseBoolean(vr.val.toString())
== true ) || ( Boolean.parseBoolean(vl.val.toString()) == true ) )
                              ret.val = Boolean.valueOf("true");
                        else
                              ret.val = Boolean.valueOf("false");
                  }
                  return ret;

            }catch (Exception ex) {
                  System.out.print(ex.getMessage());
                  return null;
            }
      }
}
```

Built In functions:

PrintFunc.java : prints arguments

```
import java.util.*;
public class PrintFunc extends Func{
      public PrintFunc()
      {
      }

      public Value execute(Vector<Expr> args, Env env)
      {
            if (args != null){
                  for(ListIterator<Expr> iter =
args.listIterator();iter.hasNext();)
                  {
                  System.out.print (iter.next().evaluate(env));
                  }
```

```
            }
            return null;
      }
}
```

PrintlnFunc.java : prints arguments and gives new line

```
import java.util.ListIterator;
import java.util.Vector;
public class PrintlnFunc extends Func{
      public PrintlnFunc()
      {
      }

      public Value execute(Vector<Expr> args, Env env)
      {
            for(ListIterator<Expr> iter =
args.listIterator();iter.hasNext();)
            {
                  System.out.print(iter.next().evaluate(env));
            }
            System.out.print("\n");
            return null;
      }
}
```

SumFunction.java
Computes summation of all arguments of function

```
import java.util.*;

public class SumFunction extends Func{

      public SumFunction()
      {
      }

      public Value execute(Vector<Expr> args, Env env)
      {
            int n = args.size();
            if ( n == 0 ) {
                  System.out.print("\nError: No arguments for Sum
function");
                  System.exit(1);
            }

            Value ret = ((Expr)(args.elementAt(0))).evaluate(env);
            if (n == 1) {
                  return ret;
            }

            for(int i=1; i < n ; i++)     {
```

```
                Value vr = ((Expr)(args.elementAt(i))).evaluate(env);

                Type tmp= Type.max(vr.type, ret.type);

                if (tmp == null) {

                        System.out.print("\nError: Type mismatch
in Sum function : Operation undefined for " + ret.type.name  + " and "
+ vr.type.name);
                        System.exit(1);
                }

                if ( tmp == Type.Int ) {
                      ret.type = Type.Int;
                      ret.val = Integer.parseInt(ret.val.toString())
+ Integer.parseInt(vr.val.toString());
                }
                if ( tmp == Type.Float ) {
                      ret.type = Type.Float;
                      ret.val = Float.parseFloat(ret.val.toString())
+ Float.parseFloat(vr.val.toString());
                }
            }

            return ret;
      }
}
```

Average function: Computes average over given argument list

```
import java.util.*;

public class AverageFunc extends Func{
      public AverageFunc()
      {
      }

      public Value execute(Vector<Expr> args, Env env)
      {
            int n = args.size();
            if ( n == 0 ) {
                  System.out.print("\nError: No arguments for avg
function");
                  System.exit(1);
            }

            Value ret = ((Expr)(args.elementAt(0))).evaluate(env);
            if (n == 1) {
                  return ret;
            }

            for(int i=1; i < n ; i++)     {
                  Value vr = ((Expr)(args.elementAt(i))).evaluate(env);

                  Type tmp= Type.max(vr.type, ret.type);
```

```
                  if (tmp == null) {

                              System.out.print("\nError: Type mismatch
in avg function : Operation undefined for " + ret.type.name  + " and "
+ vr.type.name);
                              System.exit(1);
                  }

                  if ( tmp == Type.Int ) {
                        ret.type = Type.Int;
                        ret.val = Integer.parseInt(ret.val.toString())
+ Integer.parseInt(vr.val.toString());
                  }
                  if ( tmp == Type.Float ) {
                        ret.type = Type.Float;
                        ret.val = Float.parseFloat(ret.val.toString())
+ Float.parseFloat(vr.val.toString());
                  }
            }

            ret.type = Type.Float;
            ret.val = Float.parseFloat(ret.val.toString())/n;

            return ret;
      }
}
```