R – A Scripting language for a call routing engine

Final Report: COMS W4115: Programming Languages and Translators

Rajiv S Kumar

rk2268@columbia.edu

Introduction:

Most call centers today use **ACD** (Automatic call distribution) technology to send incoming customer phone calls to Agents. Agents in a call center typically login to a Queue and wait for calls. The telephony switch used in the call center has ACD capabilities to keep incoming calls in queue and play messages to the caller until an agent becomes available. When agents are available, the call is sent to the agent who has been available for the longest time.

Most ACDs have limited scripting capabilities to customize call routing behavior. However, most ACDs do have the concept of a **"Route Point"**, where a call can be parked to let a third party application determine where to send the call. This third party application, called a **"Routing Engine"**, is typically custom coded in a high level language such as C++ or Java. Very few general purpose routing engines exist in the market and they tend to be proprietary and extremely expensive.

R is a scripting language for a general purpose call routing engine called "**R-Engine"**. **R** has the basic features used in most call centers to let system administrators quickly customize their call routing behavior. **R** is a strongly typed, interpreted language that uses the classical syntax of languages such as Java and C.

The **R** Script would be triggered by a call landing at a Route Point. The ultimate purpose of every **R** Script is to specify a "target" where R-Engine should send the call. The R-Engine has a library of inbuilt functions to provide access to things like Date/ Time, Call properties and Call Center Statistics. The **R** scripts work on the data returned by these inbuilt functions and specify a target. The following examples should make the concept clear.

Example Scripts:

Date time example:

```
/**
* This is a sample R program that sends calls to agents during office hours (Mon-Fri 9 AM to 5 PM)
* and sends calls to voice mail during after hours.
*/

R {

	//Get hour in 24 hr format
	int hour = e.getDateTime(11);
	e.print("Hour now is " + hour);

	//Get day of week with Sunday = 1
```

```
        int dayOfWeek = e.getDateTime(7);
        e.print("Day of week is " + dayOfWeek);

        target = "5601"; //voicemail number

        if (dayOfWeek >= 2 && dayOfWeek <= 6
                && hour >= 9 && hour < 17) {
                        target = "8000"; //Office hours number.
        }

        e.print(target);

}
```

A caller language preference example:

```
/**
* This is a sample R program that sends calls to queue 78001 for english speaking
* callers and sends calls to 79001 for spanish speaking callers.
*/

R {

        e.print(e.callProperty("callerLang"));

        if ( e.callProperty("callerLang") == "English") {
                target = "78001";
        } else if ( e.callProperty("callerLang") == "Spanish") {
                target = "79001";
        }

        //Call is now transferred to the number specified by "target".

}
```

Features:

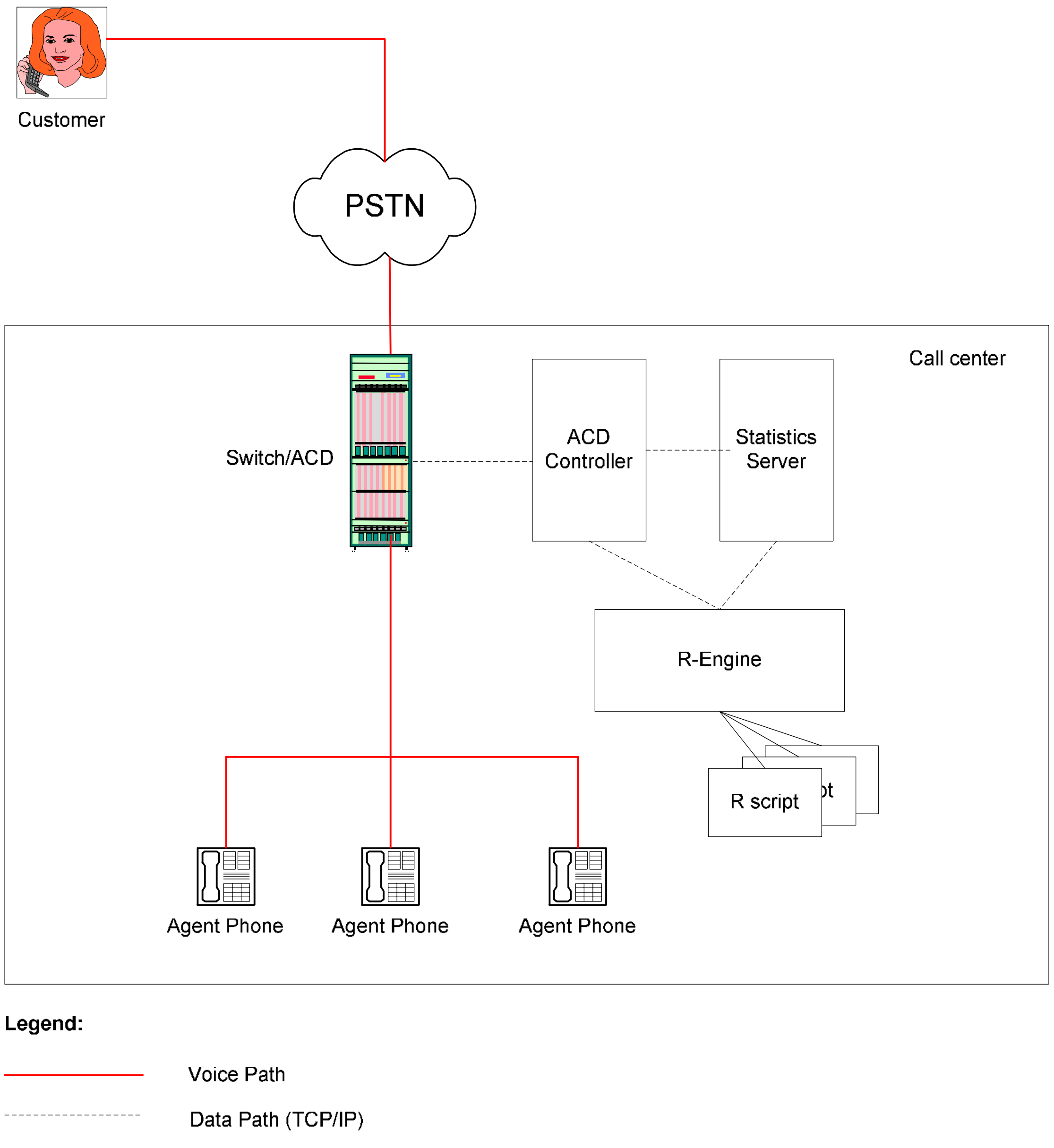
R has the basic features of a scripting language such as the ability to declare variables, arrays, loops, conditional statements, user-defined functions and the ability to call R-Engine inbuilt functions. The real power of such a scripting language comes from the ability to harness the R-Engine inbuilt functions to build sophisticated call routing logic that is tailor made to suit the call center’s needs.

Implementation:

R is implemented as an interpreted language. The ANTLR generated lexer, parser and tree walker are part of the R-Engine itself, so that the scripts can be directly fed to the R-Engine for execution. The R-Engine has been developed in Java.

Real Life Scenario:

Since the main purpose of this project is to develop a scripting language, a few inbuilt functions have been developed for demonstration purposes. The R scripts part of the demo only print the target to the console after executing the script. When used in real life, the R-Engine would be part of a conglomerate of software components that control call routing behavior as shown below:



The typical sequence of operations would be as follows:

1. Customer calls a toll free number.
2. The PSTN (public switch telephony network) sends the call to the switch at the call center.
3. The switch (ACD) parks the call at a route point and tells the ACD Controller that there is a new call at the route point. Prior to this, the switch or an IVR might ask the caller for the caller information such as language preference and account number and store all this information in the ACD controller.
4. ACD controller asks the R-Engine to execute the script corresponding to the route-point. It also passes all of the call properties such as callers account number, language preference etc as inputs to the R-Engine.
5. R-Engine executes the script corresponding to the route point and selects a target. The R-Engine tells the ACD controller to transfer the call to the selected target. *We are only interested in this step for our project.*
6. ACD controller informs the switch to transfer the call to the selected target.
7. ACD/Switch transfers the call to the selected target.

Language Tutorial

R uses the classical syntax of modern programming languages such as Java. For someone familiar with these languages, very little training is necessary. This choice of syntax was deliberate because most people who build and maintain such systems would have dabbled with programming in their past and there is no need to make them learn a completely new syntax. However, R is a simple scripting language that doesn't have advanced concepts such as object oriented programming or polymorphism. Every R Script has the following structure:

R {

declarations | statements

}

The R scripting language can be quickly learnt by looking at a few samples.

A simple example:

R Scripts are text files with a .r extension. In the test scripts folder, if you open tutorial1.r with a text editor such as notepad, you should see the following:

```
R {
     string y = "8000";
     string z = "8001";
     int i = 10;
     if (i == 10 ) {
           target = y;
     }
     else {
           target = z;
     }
     e.print(target);
     //This script will send the call to extension 8000
}
```

To run the above sample, use the following command line command:

```
java -classpath antlr.jar;REngine.jar REngine tutorial1.r
```

This would print the following lines on the console:

```
8000
Routing call to 8000
```

Notes:

The above sample selects one of two different targets based on the value of the variable i and prints the value of the target. It's easy to note that the syntax to declare variables, string literals, if then else construct, semicolon at the end of each statement are all like Java.

R has a predefined string variable named target which should be set by the script before the end of the script. Also, inbuilt R functions are called with the **e.** prefix. For e.g. in the above sample, the inbuilt print function is called as e.print().

More Example Scripts:

A Date time example:

Let us go back to the date time example mentioned before. In the test scripts folder, if you open testDateTime.r, you would see the following:

```
/**
* This is a sample R program that sends calls to agents during office hours (Mon-Fri 9 AM to 5 PM)
* and sends calls to voice mail during after hours.
*/

R {

    //Get hour in 24 hr format
    int hour = e.getDateTime(11);
    e.print("Hour now is " + hour);

    //Get day of week with Sunday = 1
    int dayOfWeek = e.getDateTime(7);
    e.print("Day of week is " + dayOfWeek);

    target = "5601"; //voicemail number

    if (dayOfWeek >= 2 && dayOfWeek <= 6
        && hour >= 9 && hour < 17) {
            target = "8000"; //Office hours number.
    }

    e.print(target);

}
```

To run the script, use the following command line command:

```
java -classpath antlr.jar;REngine.jar REngine testDateTime.r
```

This prints something like the following to the console:

```
Hour now is 16
Day of week is 3
8000
Routing call to 8000
```

Notes:

The above sample uses the R-Engine inbuilt getDateTime function to get the current hour and current day of the week and decides where to send the call. The getDateTime function internally uses the get() method of java.util.Calendar. The input argument for the getDateTime function is the same as the Constant Field values of the java.util.Calendar class.

A caller language preference example:

CallerLang.r in the test folder demonstrates how to select different targets based on the caller's language preference. Call properties such as the caller's language preference can be input as key-value pairs in the second command line argument to the REngine in the format shown below:

key1=value1;key2=value2;key3=value3

The REngine makes the key-value pairs passed as a command line argument available to the script with the e.callProperty method.

```
/**
* This is a sample R program that sends calls to queue 78001 for english speaking callers and sends
calls to 79001 for spanish speaking callers.
*/

R {

     e.print(e.callProperty("callerLang"));

     if ( e.callProperty("callerLang") == "English") {
          target = "78001";
     } else if ( e.callProperty("callerLang") == "Spanish") {
          target = "79001";
     }

     //Call is now transferred to the number specified by "target".

}
```

To run the script, with English as the language preference, use the following command:

```
java -classpath antlr.jar;REngine.jar REngine CallerLang.r
callerLang=English
```

This would print:

```
78001
Routing call to 78001
```

To run the script, with Spanish as the language preference, use the following command:

```
java -classpath antlr.jar;REngine.jar REngine CallerLang.r
callerLang=Spanish
```

This would print:

```
79001
Routing call to 79001
```

To run the script, with Dutch as the language preference, use the following command:

```
java -classpath antlr.jar;REngine.jar REngine CallerLang.r
callerLang=Dutch
```

This would print:

```
Target not set. Call will be dropped
```

User Functions:

Users can define their own functions as shown in the following example (tutorial2.r):

```
/*
This is a sample R program to demonstrate user functions
*/

R {


    int y = foo(10);
    e.print("After foo() y = " +  y);
    bar();
    target = "9000";
    e.print("Back in main target = " + target);

    [UserFunctions]

    int foo(int j) {

        e.print("Inside foo j = " + j);
        return j*10;
    }

    bool bar() {
         e.print("Inside bar");
         return true;
    }
}
```

To run this sample, use the command:

```
java -classpath antlr.jar;REngine.jar REngine tutorial2.r
```

This would print:

```
Inside foo j = 10
After foo() y = 100
Inside bar
Back in main target = 9000
Routing call to 9000
```

Notes:

Unlike inbuilt functions, user defined functions can be called without the e. prefix. The following rules apply to user defined functions.

- User defined functions should be contained within the R { } block.
- R has no void data type; all user functions should return a value.
- User functions cannot be nested within other user functions.
- The block of code containing user defined functions should start with the `[UserFunctions]` attribute.
- The `[UserFunctions]` attribute should start after the last statement of the R { } block. The following is an error:

  ```
  R {

  [UserFunctions]

  int foo() {
  return 0;
  }

  //error. After [UserFunctions], only function definitions //are allowed
  int x;
  x = 5;

  }
  ```

User functions have access to the variables defined in the containing R { } block. User functions can also define their own local variables. The next section explains the scoping rules for variables.

Scope Rules:

Variables defined inside user functions have local scope. Variables defined inside the R { } block have global scope (visible to all user functions). Unlike in Java and C++ where each left brace begins a new scope, R has a single scope for the whole function. Hence redefining a variable with the same name is an error as shown in the following example:

```
R {

    int x = 100;
    if (true) {
        int x = 50; //error. x redefined.
    }
}
```

The following example illustrates how the variables defined in the R { } block are accessible to user functions.

```
R {
    int x = 100;
    foo();
    e.print("Back in main x = " +  x); //Prints 100

    [UserFunctions]

    int foo() {
        e.print("Inside foo x = " + x); //Prints 100 - the outer x.

        int x = 50; //Ok. x is local to foo.
        e.print("Local x = "+ x);  //Prints 50 - the local x.
        return 0;
    }

}
```

This example prints:

```
Inside foo x = 100
Local x = 50
Back in main x = 100
Target not set. Call will be dropped
```

In R, function arguments are passed by value. If the value of a variable should be available outside a user function, the variable can be made global (defined in the R { } block).

Type conversion:

R data types can be converted to other types using the three inbuilt type conversion functions defined in R:

- `toI – Converts to int.`
- `toS – Converts to string.`
- `toF – Converts to float.`

A type conversion example (typeConvert.r) is shown below:

```
R {

//Type conversion functions example

     string s = "345.67";

     float f = e.toF(s);
     e.print("float f = " + f);

     int i = e.toI(s);
     e.print("int i = " +  i);

     s = e.toS(f + 10.5);
     e.print("f + 10.5 = " + s);

     s = e.toS(i - 5);
     e.print("i - 5 = " + s);
}
```

The above example prints:

```
float f = 345.67
int i = 345
f + 10.5 = 356.17
i - 5 = 340
Target not set. Call will be dropped
```

Arrays:

R supports multi-dimensional arrays of the basic data types as shown in the following example (testArrays.r).

```
R {
     //Test arrays

     string y[3];
```

```
    y[0] = "he";
    y[1] = "haw";
    y[2] = "ha";

    e.print(y[0]);
    e.print(y[1]);
    e.print(y[2]);

    int x = 5;
    int a[x+5]; //Dynamic allocation of arrays

    a[9] = 3;
    e.print(a[9]);

    int i[2][3];
    i[0][0] = 0;
    i[0][1] = 1;
    i[0][2] = 2;
    i[1][0] = 3;
    i[1][1] = 4;
    i[1][2] = 5;

    e.print(i[0][0]);
    e.print(i[0][1]);
    e.print(i[0][2]);
    e.print(i[1][0]);
    e.print(i[1][1]);
    e.print(i[1][2]);

    int j[2][3][2];

    j[0][2][1] = 100;
    e.print(j[0][2][1]);

}
```

The above example prints:

```
he
haw
ha
3
0
1
2
3
4
5
100
Target not set. Call will be dropped
```

Loops

It is possible to write loops in R code using the while statement. The semantics of while, break, continue and return statements are similar to those in Java.

```
R {

    //This sample tests while loops, break,
    //continue and return statements.

    int i;
    while (i < 20) {i = i + 1;}

    e.print("i = " + i);

    i = testBreak(10);
    e.print("testBreak returned i = " + i);

    int g;
    testReturn();
    e.print("after testReturn g = " + g);

    testContinue();
    e.print("after testContinue g = " + g);

    [UserFunctions]

    int testBreak(int j) {

        while (j < 20) {
            if (j == 15) {
                e.print("testBreak breaking");
                break;
            }
            j = j + 1;
            e.print("testBreak j = " +  j);
        }

        e.print("testBreak before return");
        return j;

    }


    int testReturn() {

        g = 0;
        while (g < 5) {
            if (g == 3) {
                e.print("testReturn returning");
                return 0;
            }
```

```
                    g = g + 1;
                    e.print("testReturn g = " +  g);
            }

            //We should

            e.print("testReturn return");
            return 0;
    }


    int testContinue() {
            g = 6;
            while (g < 11) {

                    if (g == 8) {
                            e.print("g is 8");
                            g = g + 1;
                            continue;
                            e.print("shouldn't print this");
                    }
                    else {
                            e.print("while loop g is " + g);
                            g = g + 1;
                    }

            }
            e.print("Out of while loop");
            return 0;
    }

}
```

Language Manual

Lexical Conventions:

The R Script programs would be stored in files with a .r extension. The following describes the lexical conventions of the language.

White Space

Spaces, tabs, newlines, form feeds and comments are considered white space. White space is used for separating tokens, but ignored.

Comments

Multiline comments would start with /* and end with */. Single line comments would begin with //. Nesting comments is not allowed. Also, comments cannot occur within string literals.

Identifiers

Identifiers or names refer to variables and function names. An identifier is a sequence of letters and digits which begin with a letter. Identifiers are case sensitive.

Keywords

The following identifiers are reserved for use as keywords may not be used otherwise:

bool	int
break	return
continue	string
else	while
float	target
if	

Constants

R has integer constants, floating point constants and string literals. Integer constants are a sequence of digits with an optional – sign in the beginning. Floating point constants are also an optionally signed sequence of digits, followed by a dot and followed by another sequence of digits. String literals are a sequence of characters enclosed in double quotes. String literals can also contain the following escape sequences:

newline	\n
horizontal tab	\t
carriage return	\r
backslash	\\
single quote	\’
double quote	\”

Types

R has four data types: bool, int, float and string. These types are internally implemented using java data types are shown in the table below:

R Data type	**Corresponding Java Type**
bool	boolean
int	long
float	double
string	java.lang.String

Program Structure

The general structure of an R script would be as follows:

```
R {
declarations | statements

user defined functions
}
```

R requires variables to be defined before they are used. All the variables declared in the declarations section have global scope, which means they are accessible to all the user functions. However, the variables defined inside user defined functions have local scope.

Statements as defined below can be assignment expressions, function calls, compound statements, selection statements etc. In the statement block, the script can call inbuilt functions or user defined functions. The ultimate goal of the statement block is to set the value of an inbuilt variable called “target”. After executing the last statement, the R-Engine will transfer the phone call to the specified target.

User defined functions will let the user break down a large script into reusable pieces of code. User defined functions are explained in detail below.

Declarations

R requires variables to be defined before they are used. All the variables declared in the declarations section will have global scope. However, the variables defined inside user defined functions will have local scope. A declaration can be for a basic type or an array. Declarations will have the form:

Type Identifier [Optional array brackets and size] [Optional assignment expression];

Examples:

```
int i;
float x;
string s = “Hello”;
int a[10][10];
```

R will automatically initialize bool variables to false, int and float variables to zero. String variables are initialized to an empty string.

Statements

Statements can be one of:

- Expression Statement
- Compound Statement
- Selection Statement
- Iteration Statement
- Jump Statement

Expression Statement:

Most expression statements are assignments or function calls. An expression can also be empty. Arithmetic operations are supported on the int and float types, but not for strings and arrays. However, strings can be concatenated using the + operator and lexicographically compared using the comparison operators. The R-Engine provides inbuilt functions for manipulating strings.

Operator Precedence:

The following table shows the precedence and associativity of all operators. Operators grouped in the same row all have the same precedence. Each row has higher precedence than the next. These operators have the same meaning as in C.

Operator	**Description**	**Associativity**
() []	Function call, Array index	Left to right
! -	Negation, Unary –	Right to left
* / %	Mult, Div, Modulus	Left to right
+ -	Addition, Subtraction	Left to right
< <= > >=	Relational Operators	Left to right
== !=	Equality Operators	Left to right
&&	Logical And	Left to right
||	Logical Or	Left to right
=	Assignment	Right to left

Compound Statement:

Compound statements are a sequence of statements in the form:

```
{
        declarations | statements
}
```

Compound statements are useful as the body of if, else and while statements. The body of a function is also a compound statement.

Selection Statement:

R provides the “if” “else” statements for choosing one of several flows of control.

Iteration Statement

R provides the while loop statement in the form:

```
while (expression) statement
```

Jump Statement

R provides the following statements for transfer control unconditionally.

- break;
- continue;
- return;

The meanings of these statements are similar to those of most high level languages like C.

Functions

Functions can be user defined or inbuilt functions.

All functions will have the form:

```
returnType functionName (formalParameters) {
        declarations | statements
}
```

The arguments to a function are passed by value. If the modified value of a variable is to be available outside the function, the variable can be made global by declaring it in the beginning of the R script.

R has no void type. All functions should return a value. If the functions do not provide a return value, the return values will be assumed to be zero for integer and float types and an empty string for string types.

Inbuilt Functions:

The R-Engine will provide inbuilt functions to access call properties, date/time, string manipulation routines etc. All inbuilt R-Engine functions should be called with the prefix “e.”. For example:

```
string x;
x = e.callProperties(“AccountNumber”);
```

User Defined Functions:

R Scripts can have functions defined by the user. User defined functions should be defined after the statement block. Also, the block should begin with the attribute `[UserFunctions].`

Project Plan

Process

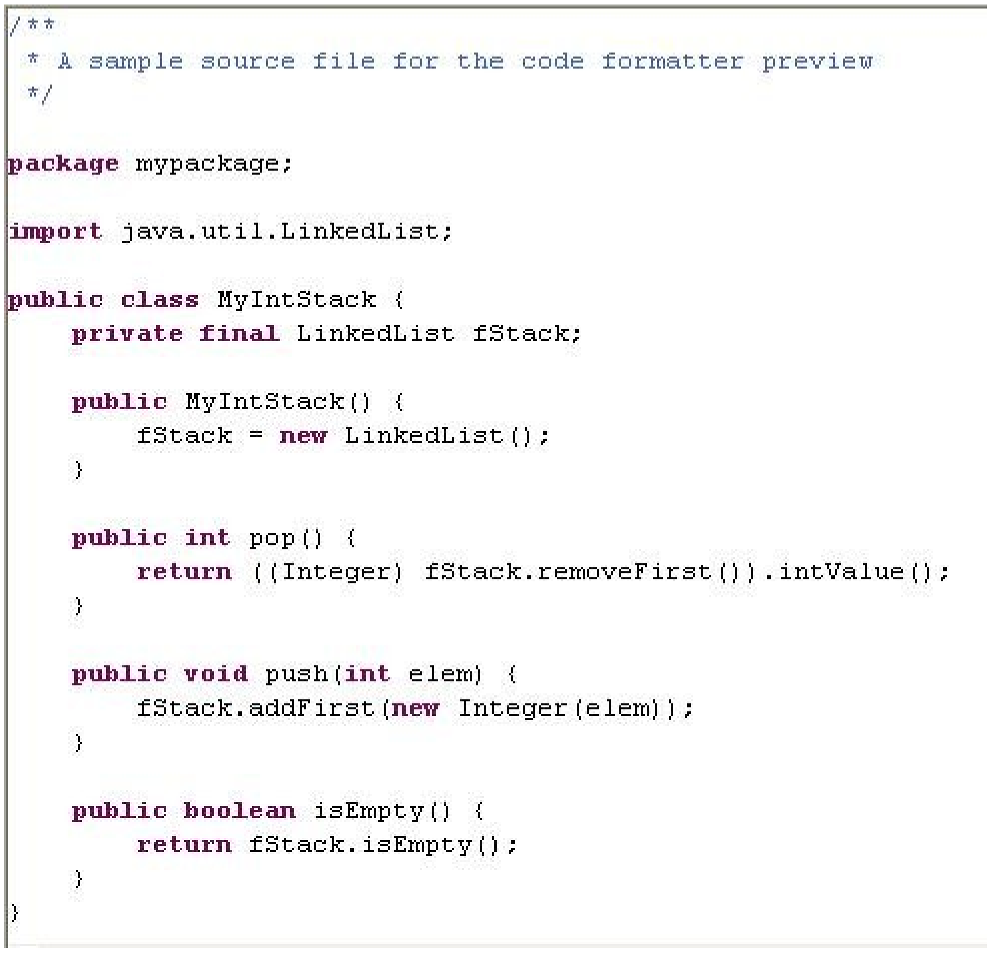
As a CVN student working alone, there was no need for me to plan who does what. The timelines for the deliverables were dictated by the course deadlines. For development and testing I used an iterative process of implementing new features and regression testing the whole project with automated scripts. Every week, I had a TODO list of features I wanted to implement, such as implement arrays, implement user functions and so on. After implementing each new feature, I created test scripts to test the new feature and added them to the automated testing suite. After every milestone, I updated the code in my version control system (Visual source safe).

Programming style guide

R code was written in Java using the Java coding conventions defined on the java.sun.com website:

http://java.sun.com/docs/codeconv/html/CodeConvTOC.doc.html

The code was formatted using the source code formatting feature of Eclipse. The template used for formatting is shown below:



```
/**
 * A sample source file for the code formatter preview
 */

package mypackage;

import java.util.LinkedList;

public class MyIntStack {
    private final LinkedList fStack;

    public MyIntStack() {
        fStack = new LinkedList();
    }

    public int pop() {
        return ((Integer) fStack.removeFirst()).intValue();
    }

    public void push(int elem) {
        fStack.addFirst(new Integer(elem));
    }

    public boolean isEmpty() {
        return fStack.isEmpty();
    }
}
```

Project timeline

The project timeline was largely dictated by the class deadlines. The timeline set for the class was as follows:

09/05	Semester started
09/26	Language White Paper due
10/19	Language Reference Manual due
12/19	Final Reports due

Roles and responsibilities

As a CVN student who was supposed to work alone on this project, the author performed all the roles himself.

Software development environment

The whole project was developed as a Java project using the Eclipse IDE on a Windows XP laptop. Full details of the software development environment are as follows:

- JDK version 1.5.0_06
- Eclipse SDK 3.1.2
- ANTLR version 2.7.6
- Windows XP Professional, Service Pack 2.

Visual Source safe on the author's corporate LAN was used for version control.

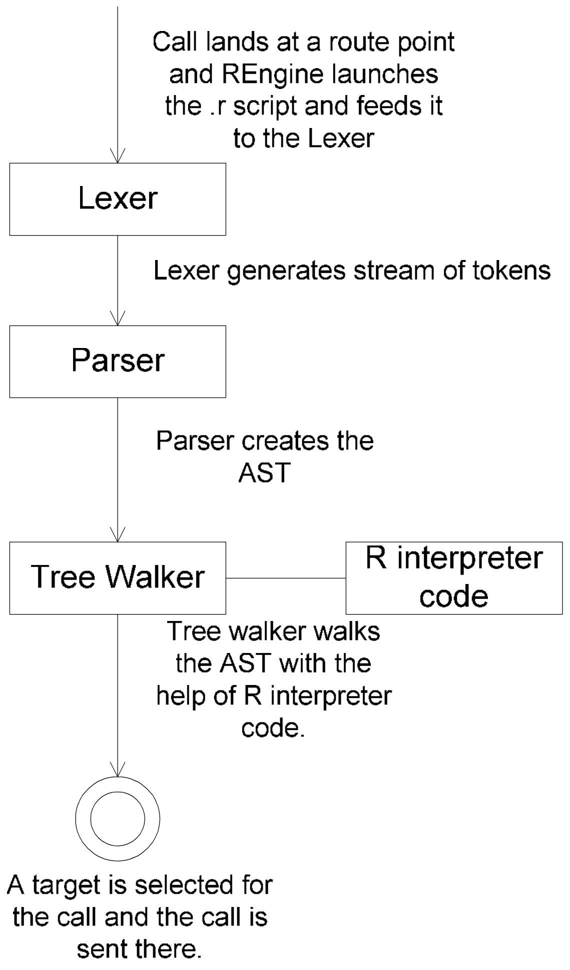
Project log

The actual dates for the project are as follows:

09/12 - 09/19	Worked on White Paper
09/19	Submitted White Paper
09/20	Received feedback on White Paper
09/26 - 10/17	Worked on Language Reference Manual
10/18	Submitted Language Reference Manual
10/19 - 10/31	Worked on Scanner and Parser
10/31	Got Scanner and Parser to generate AST as expected
11/01 – 11/07	Tree walker. Got expressions to work
11/07- 11/14	Got inbuilt functions and user functions to work
11/14 – 11/21	Got loops and arrays to work
11/21 – 11/28	Testing, bug fixing and adding code for REngine.
11/28 – 12/06	Break for homework and exam.
12/09 - 12/13	Refactoring code to improve error handling and code cleanup.
12/13 – 12/18	Worked on Final Report
12/19	Project complete.

Architectural Design

The following diagram shows the basic steps of the R Scripting language.


Call lands at a route point and REngine launches the .r script and feeds it to the Lexer
Lexer
Lexer generates stream of tokens
Parser
Parser creates the AST
Tree Walker
R interpreter code
Tree walker walks the AST with the help of R interpreter code.
A target is selected for the call and the call is sent there.

The lexer, parser, tree walker and the R interpreter code are explained below.

The Lexer:

The Lexer classes for the project are in RLexer.java and RLexerTokenTypes.java. They were automatically generated by ANTLR using the r.g grammar file (r.g is shown in the appendix). For someone familiar with the Dragon book sample on the class web site, the grammar should look familiar. The sample has been extended to include things like single and multi line comments, string literals, escape sequences and a few more operators. The Lexer uses a look ahead of k = 2.

The Parser:

RParser.java has the parser generated by ANTLR using the r.g grammar file (r.g is in shown in the appendix). The parser uses look ahead of k = 2 and automatically builds the AST. For someone familiar with the Dragon book sample on the class web site, the parser grammar should look familiar. The dragon book sample has been extended to include strings and string literals, user function declarations, user function calls, inbuilt function calls, ability to interleave declarations and statements and the ability to assign values to variables in the declaration statements.

The Tree Walker:

RWalker.java has the tree walker generated by ANTLR using the r.g grammar file (r.g is shown in the appendix). The tree walking is done in two steps. The first step (prewalktree) is to find user defined functions and store references to their AST body in the user function table. The second step (called walktree) is the actual execution of the AST. It starts at the root node called "R" and walks down until there are no more executable statements. After the tree walking is complete, the REngine checks the value of the target and sends the call to that number. The tree walker depends on the rest of the R interpreter code to maintain symbol tables, create data types, handle errors, copy function arguments and so on.

R interpreter code:

The R interpreter code is the rest of the project code that was 'not' generated by ANTLR. The responsibilities of the R interpreter code include:

- Starting the execution of the script
- Creating and managing data types
- Providing the body of the code for operations on data types such as + and -.
- Maintaining context
- Maintaining symbol tables
- Maintaining table of user functions
- Executing internal functions
- Executing user functions
- Handling errors
- Setting call properties at the beginning of the script
- Transferring calls after the execution of the script

The different classes of the R Interpreter code and their responsibilities are explained in detail below.

REngine

REngine (REngine.java in the appendix) is the starting point of the project. The script to execute and the call properties are passed as command line argument to the REngine. It stores the call properties in a hash table and makes those available to the script (through the callProperties function) during execution. It loads the lexer, parser and tree walker to execute the input script. At the end of the script, REngine checks the value of the target and sends the call to the target. REngine is also responsible for handling exception thrown during the scanning, parsing or tree walking process.

RInterpreter

RInterpreter (RInterpreter.java in the appendix) is the loaded by the tree walker to execute the AST. It holds the main context and has a reference to the current context object. It also maintains the user function table and has the code to find and execute user defined functions.

RContext

RContext (RContext.java in the appendix) represents a function context. A context is where the symbol table lives. RContext has the functions to add variables into the Symbol table, assign variable values, lookup symbol table values and maintain some state information required for loops. The reference to the main context within which the code in the R { } body executes is created at script startup and is always present in the RInterpreter. The function contexts for user functions are created dynamically upon function invocation. Each time a user function is called, we enter a new context with its own symbol table. When the function call returns, the previous context becomes the current context. This supports calling user functions within user functions as shown in testUserFuncs.r.

RSymbolTable

RSymbolTable (RSymbolTable.java in the appendix) represents the symbol table. Internally it uses a hash table to map names and variables. It has a pointer to the parent symbol table (of the main context).

RFunctions

RFunctions (RFunctions.java in the appendix) has all the inbuilt functions of the R scripting language. This includes functions like e.print(), type conversion functions, string manipulation routines, call properties functions and call statistics functions. The tree walker calls the static function “callRFunction” with the name of the function to execute and a vector of arguments. callRFunction will find the function and execute it with the list of arguments.

RUserFunction

RUserFunc (RUserFunc.java in the appendix) represents a user defined function. It stores the list of arguments, return value, the AST and context for the function body. It creates the function context when the user function is invoked. Objects of this type are stored in the user function table during the first step of the tree walking process.

RFactory

RFactory (RFactory.java in the appendix) creates the R data types based on the corresponding types in the script. Any time during the tree walking process if we need to create a variable, we call the create method of RFactory.

RType

RType (RType.java in the appendix) is the base class from which all the R data types derive. It defines dummy methods for all the operators that are possible on data types such as arithmetic operators, logical operators and unary operators. All the operators in this base class throw a “not implemented” error. It is up to the derived classes to provide meaningful implementations of these operators.

RBool

RBool (RBool.java in the appendix) represents the bool data type in r scripts. It derives from RType and provides operators such as logical and, logical or and comparison operators to compare bool types. Internally it stores the value as a java boolean primitive.

RInt

RInt (RInt.java in the appendix) represents the int data type in r scripts. It derives from RType and provides arithmetic operators and comparison operators for numeric types. Internally it stores the value as a java long primitive.

RDouble

RDouble (RDouble.java in the appendix) represents the float data type in r scripts. It derives from RType and provides arithmetic operators and comparison operators for numeric types. Internally it stores the value as a java double primitive.

RString

RString (RString.java in the appendix) represents the string data type in r scripts. It derives from RType and provides comparison operators and the plus operator for string types. Internally it stores the value as a java.lang.String object.

RArray

RArray (RArray.java in the appendix) represents a multi dimensional array of arbitrary dimensions. Internally it stores the array items as a single dimensional array of RType. The dimensions of the array are set at declaration and cannot be modified later. The algorithm in the access() function computes the actual index within the single dimension that should be used to access the array elements.

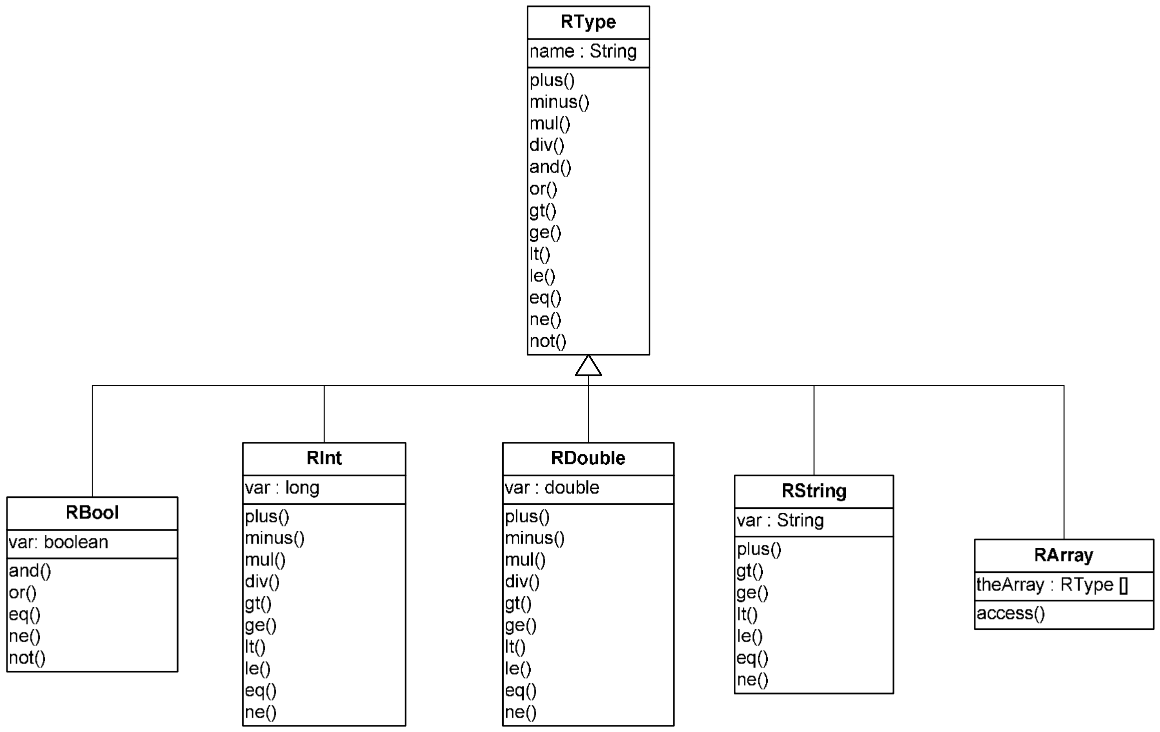
RException

RException (RException.java in the appendix) represents an exception that can get thrown during the tree walking process. The REngine is responsible for catching these exceptions and showing the message to the user.

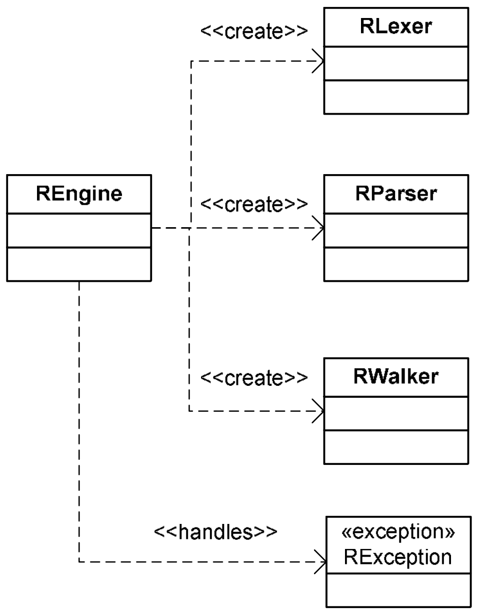
Class Diagrams

The following UML class diagrams show the relationships between the different classes in the R interpreter code.

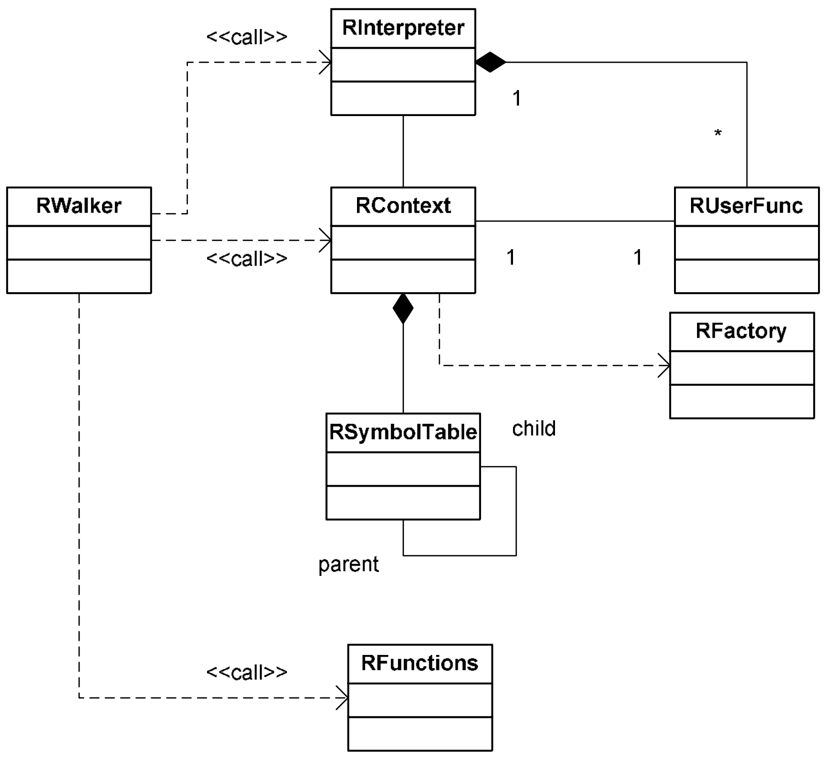
Class Diagram - Data Types:



Class Diagram – REngine



Class Diagram – Tree Walker



Test Plan

The language tutorial in chapter 2 of this report has several working examples, each of which test some specific feature of R. The zip file containing this final report has more than 40 scripts which test the scripting language in detail. The test scripts are also listed in the appendix. Each of the scripts whose name is of the form error*.r contains R code that is supposed to generate errors. Each of the error*.r files test some error handling feature of R. The following table has a list of other scripts in the test folder and their purpose:

Script Name	**Purpose**
testExpr.r	Tests expressions
testArrays.r	Tests Arrays
scope.r	Tests scope rules
loops.r	Tests while, break, continue and return statements
typeConvert.r	Tests type conversion functions
testRecursion.r	Tests recursion
StringCompares.r	Tests string comparison operators
inbuiltFuncs.r	Tests R-Engine inbuilt functions
testuserFuncs.r	Tests user defined functions
testDateTime.r	Demonstrates how to route calls based on time of day.
CallerLang.r	Demonstrates how to route calls based on the callers language preference
serviceLevel.r	Demonstrates how to route calls based on the service level statistic.
estimatedWaitTime.r	Demonstrates how to route calls based on the estimated wait time statistic.

Automated testing

A batch file (runAll.bat) was used to run the test scripts one after the other. The output of each script is added is added to a file and compared against an expected result. Each time a change is made to the R compiler code, the automated test script was run to make sure nothing broke. The automated test batch file which runs 44 scripts one after the other and collects the results is shown below:

```
java -classpath antlr.jar;REngine.jar REngine testExpr.r >1.tmp
java -classpath antlr.jar;REngine.jar REngine testArrays.r >2.tmp
java -classpath antlr.jar;REngine.jar REngine scope.r >3.tmp
java -classpath antlr.jar;REngine.jar REngine loops.r >4.tmp
java -classpath antlr.jar;REngine.jar REngine typeConvert.r >5.tmp
java -classpath antlr.jar;REngine.jar REngine testRecursion.r >6.tmp
java -classpath antlr.jar;REngine.jar REngine StringCompares.r >7.tmp
java -classpath antlr.jar;REngine.jar REngine inbuiltFuncs.r >8.tmp
java -classpath antlr.jar;REngine.jar REngine testuserFuncs.r >9.tmp
java -classpath antlr.jar;REngine.jar REngine testDateTime.r >10.tmp
java -classpath antlr.jar;REngine.jar REngine CallerLang.r callerLang=English >11.tmp
java -classpath antlr.jar;REngine.jar REngine CallerLang.r callerLang=Spanish >12.tmp
java -classpath antlr.jar;REngine.jar REngine serviceLevel.r >13.tmp
java -classpath antlr.jar;REngine.jar REngine estimatedWaitTime.r >14.tmp
FOR /L %%v IN (1,1,30) DO java -classpath antlr.jar;REngine.jar REngine error%%v.r
>e%%v.tmp
copy /Y 1.tmp testResult.txt
FOR /L %%v IN (2,1,14) DO copy testResult.txt+%%v.tmp
FOR /L %%v IN (1,1,30) DO copy testResult.txt+e%%v.tmp
del *.tmp
fc testResult.txt ExpectedResults.txt
```

The automated script will show some differences between the test result and the expected result because the functions for date and time will generate different results each time they are run. Apart from these minor differences, everything else should match the expected results.

Lessons Learned

This was a very useful course that gave me a good understanding of the basics of language and compiler design. The examples on the class web site are extremely useful as a starting point for most students. I should have started looking at these examples much earlier instead of wasting my time trying to figure out things that are already available. ANTLR makes generating the lexer and parser easy and there are a lot of grammar samples out there on ANTLR web site for almost every programming language. However, all those samples only generate the lexer and parser. The real heavy lifting in the project is the tree walker where you have to write most of the code yourself.

There was a period of a few days where my AST was coming out very well but I just couldn't decide how to proceed further with the tree walker. You have to decide whether you want to write some of the tree walker code yourself or learn ANTLR syntax to generate similar tree walker code. Making your code work with ANTLR generated tree walker code can sometimes be a challenge. I wanted the R-Engine to execute the script by itself and not generate Java or C files which again need to be compiled by another compiler. I started out with the assumption that writing an interpreter will be much easier than writing a compiler that generates assembly code or IR. However, I soon realized that my tree walker had to do plenty of state management. For e.g. if the script has a break or continue statement, I had to maintain state to find out whether I am really within a while statement. Also jump statements like break and continue get complicated because your looping is implemented within the context of several loops that the ANTLR tree walker itself generates. Getting this portion of the project to work correctly was perhaps the most complicated part of the project.

All said and done, I did manage to make everything work like I had initially envisioned. Writing a compiler is a problem that takes focus and concentration and it is probably easier to work alone than working with team members who may not all share your vision for your language. I didn't have to spend any time selling my ideas about the language I wanted to create. On the down side, working with a team might have given me an opportunity to interact with team members who might have had much better ideas about how to do things than I did.

Appendix

Code listing:

ANTLR Grammar (r.g file):

```
/************************************************************
                 The Scanner
*************************************************************/

class RLexer extends Lexer;
options {
k = 2;
charVocabulary = '\3'..'\377';
}

WHITESPACE : (' ' | '\t' | '\n' { newline(); } | '\r' )+
                      { $setType(Token.SKIP); } ;

SL_COMMENT :
     "//"
     (~'\n')* '\n'
     { _ttype = Token.SKIP; newline(); }
     ;

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

protected DIGITS : ('0'..'9')+ ;

STRING_LITERAL
     :     '"'! (ESCAPECHAR|~'"')* '"'!
     ;

protected
ESCAPECHAR
     : '\\'
           (
                 'n' {$setText('\n');}
                 |'r' {$setText('\r');}
                 |'t' {$setText('\t');}
                 |'b' {$setText('\b');}
                 |'f' {$setText('\f');}
                 |'"' {$setText('\"');}
                 |'\'' {$setText('\'');}
                 |'\\' {$setText('\\');}
           )
     ;

NUM : DIGITS ('.' DIGITS { $setType(REAL); } )? ;
```

```
AND :    "&&" ;   LE :     "<=" ;   SEMI :   ';'  ;
OR  :    "||" ;   GT :     '>'  ;   LPAREN : '('  ;
ASSIGN : '='  ;   GE :     ">=" ;   RPAREN : ')'  ;
EQ :     "==" ;   LBRACE : '{'  ;   PLUS :   '+'  ;
NOT :    '!'  ;   RBRACE : '}'  ;   MINUS :  '-'  ;
NE :     "!=" ;   LBRACK : '['  ;   MUL :    '*'  ;
LT :     '<'  ;   RBRACK : ']'  ;   DIV :    '/'  ;
MOD :    '%'  ;   COMMA :  ','  ;   DOT : '.'  ;

ID : ('_' | 'a'..'z' | 'A'..'Z') ('_' | 'a'..'z' | 'A'..'Z' | '0'..'9' )* ;


/************************************************************
                    The Parser
*************************************************************/

class RParser extends Parser;
options {  k=2; buildAST = true; }
tokens  { NEGATE; DECL; FUNCTIONDEFS; FNCALL; RFNCALL; ARGS;}

program : "R"^ LBRACE! (decl | stmt)* (functionDefs)?  RBRACE! ;

basicType : "int" | "string" | "bool" | "float";

decl :  (basicType) ID (LBRACK! boolExpr RBRACK!)* (ASSIGN boolExpr)? SEMI!
            { #decl = #([DECL, "DECL"], #decl); }
            ;

stmt : loc ASSIGN^ boolExpr SEMI!
     | "if"^ LPAREN! boolExpr RPAREN! stmt (options {greedy=true;}: "else"! stmt)?
     | "while"^ LPAREN! boolExpr RPAREN! stmt
     | "break" SEMI!
     | "continue" SEMI!
     | "return"^ boolExpr SEMI!
     | compoundStatement
     | functionCall SEMI!
     | SEMI
     ;

loc      : ID^ (LBRACK! boolExpr RBRACK!)* ;
boolExpr : join (OR^ join)* ;
join     : equality (AND^ equality)* ;
equality : rel ((EQ^ | NE^) rel)* ;
rel      : expr ((LT^ | LE^ | GT^ | GE^) expr)* ;
expr     : term ((PLUS^ | MINUS^) term)* ;
term     : unary ((MUL^ | DIV^ | MOD^) unary)* ;
unary    : MINUS^ unary { #unary.setType(NEGATE); }
                  | NOT^ unary
                  | factor
                  ;
factor   : LPAREN! boolExpr RPAREN!
                  | loc
                  | NUM
                  | REAL
                  | STRING_LITERAL
                  | "true"
                  | "false"
                  | (ID LPAREN) => functionCall
                  | ("e" DOT) => functionCall
                  ;

functionCall : ID LPAREN! argList RPAREN!
                  { #functionCall = #([FNCALL, "FNCALL"], #functionCall); }
                  | "e"! DOT! ID LPAREN! argList RPAREN!
                  { #functionCall = #([RFNCALL, "RFNCALL"], #functionCall); };
```

```
argList : (boolExpr (COMMA! boolExpr)*)?;

compoundStatement : LBRACE^ (decl | stmt)* RBRACE! ;

functionDefs: (LBRACK! "UserFunctions"! RBRACK!) (functionDef)*
      { #functionDefs = #([FUNCTIONDEFS, "FUNCTIONDEFS"], #functionDefs); } ;

functionDef: (basicType) ID^ LPAREN! args RPAREN! compoundStatement ;

args : ( arg (COMMA! arg)* )?
      { #args = #([ARGS, "ARGS"], #args); } ;

arg  : (basicType) ID;


/*********************************************************
     Tree Walker
*********************************************************/

class RWalker extends TreeParser;

{
  RInterpreter rip = RInterpreter.getInstance();
}

program throws RException
{ RType r = null;}
  : #("R"
      (r=walktree)*
    )
  ;

type returns [int t]
{ t = RType.RTypeUnknown; }
  : ( "bool"  { t = RType.RTypeBool;  }
    | "string"  { t = RType.RTypeString;  }
    | "int"   { t = RType.RTypeInt;   }
    | "float" { t = RType.RTypeDouble; } )
  ;

prewalktree throws RException
{
      AST t = _t.getFirstChild();
      boolean bFuncsFound = false;
      while (t != null) {
              if (t.getText().equals("FUNCTIONDEFS")) {
                  _t = t;
                  bFuncsFound = true;
                      break;
              }
              t = t.getNextSibling();
      }
      if (!bFuncsFound) return;
}
: #(FUNCTIONDEFS (userfunc)* )
;

userfunc throws RException
{
java.util.Vector v = new java.util.Vector();
int retType = RType.RTypeUnknown;
}
:#(fname:ID

      retType=type
      doArgs[v]
```

```
        funcBody:.

        {       RUserFunc userFunc = new RUserFunc(fname.getText(), retType, v, #funcBody);
                rip.addUserFunction(userFunc);
        }

)
;

doArgs[java.util.Vector v]  throws RException
: #(ARGS (arg[v])*)
;

arg [java.util.Vector v]  throws RException
{int i = RType.RTypeUnknown; RType r = null;}
:(i=type ID
                { r = RFactory.create(#ID.getText(), i);
                  v.add(r);
                }
);

walktree returns [RType r] throws RException
        {
                RType a = null;
                RType b = null;
                r = null;

                RContext context = rip.getCurrentContext();
                if (context == null) {
                        throw new RException("Current context should not be null");
                }

                if (context.getCurrentState() == RContext.HitBreak) {
                        _retTree = null;
                        return null;
                }

                if (context.getCurrentState() == RContext.HitContinue) {
                        context.setCurrentState(RContext.OkToProceed);
                        _retTree = null;
                        return null;
                }

                if (context.getCurrentState() == RContext.HitReturn) {
                        _retTree = null;
                        return null;
                }
        }

  : #(DECL { int i = RType.RTypeUnknown; }
        (i=type ID
                {
                        AST dims = _t;
                        if (dims == null) {
                                r = context.put(#ID.getText(), i);
                        }
                        else if (dims.getType() == ASSIGN) {
                                r = context.put(#ID.getText(), i);
                                a = walktree(_t.getNextSibling());
                                context.assign(r, a);
                        }
                        else {
                                java.util.Vector arrayDims = new java.util.Vector();
                                while(dims != null) {
                                        r = walktree(dims);
                                        arrayDims.add(r);
                                dims = dims.getNextSibling();
                            }
                            context.put(#ID.getText(), i, arrayDims);
                        }
                }
    )
 )
| #(ASSIGN a=walktree b=walktree)
                        { r = context.assign( a, b ); }
| #(OR a=walktree
                {
                        if (!(a instanceof RBool)) {
                            throw new RException("left side of || should be a bool expression");
                        }
                        if (((RBool)a).var == false) {
                            //we evaluate RHS of || only
                            //if LHS evaluates to false
                            b=walktree(_t);
                            _t = _retTree;
                            r = a.or(b);
                        } else {
                            r = a; //return true
                        }
                }
    )
| #(AND  a=walktree
                {
                        if (!(a instanceof RBool)) {
                            throw new RException("left side of && should be a bool expression");
                        }
                        if (((RBool)a).var == true) {
                            //we evaluate RHS of && only
                            //if LHS evaluates to true
                            b=walktree(_t);
                            _t = _retTree;
                            r = a.and(b);
                        } else {
                            r = a; //return false
                        }
                }
    )
| #(EQ     a=walktree b=walktree { r = a.eq(b); } )
| #(NE     a=walktree b=walktree { r = a.ne(b); } )
| #(LT     a=walktree b=walktree { r = a.lt(b); } )
| #(LE     a=walktree b=walktree { r = a.le(b); } )
| #(GT     a=walktree b=walktree { r = a.gt(b); } )
| #(GE     a=walktree b=walktree { r = a.ge(b); } )
| #(PLUS   a=walktree b=walktree { r = a.plus(b); } )
| #(MINUS  a=walktree b=walktree { r = a.minus(b); } )
| #(MUL    a=walktree b=walktree { r = a.mul(b); } )
| #(DIV    a=walktree b=walktree { r = a.div(b); } )
| #(MOD    a=walktree b=walktree { r = a.modulus(b); } )
| #(NOT    a=walktree        { r = a.not(); } )
| #(NEGATE a=walktree        { r = a.uminus(); } )
| NUM             { r = context.getInt(#NUM.getText()); }
| REAL            { r = context.getDouble(#REAL.getText()); }
| STRING_LITERAL
                        { r = context.getString(#STRING_LITERAL.getText()); }
| "true"          { r = new RBool( true ); }
| "false"         { r = new RBool( false ); }
| #(LBRACE (r=walktree)*)
| #("if" a=walktree thenAST:. (elseAST:.)?)
    {
        if ( !( a instanceof RBool ) ) {
            throw new RException( "if: requires a boolean expression" );
        }
        if ( ((RBool)a).var )
            r = walktree( #thenAST );
        else if ( elseAST != null )
            r = walktree( #elseAST );
    }
| #("while"
    {
            context.whileBegin();
            AST whileBody = (AST)_t.getNextSibling();
            while (context.shouldProceed()) {
                    a = walktree(_t);
                    if (!(a instanceof RBool)) {
                            throw new RException("while: requires a boolean expression");
                    }
                    if (((RBool) a).var == true) {
                            r = walktree(whileBody);
                    } else {
                            break;
                    }
        }
        if (context.getCurrentState() != RContext.HitReturn) {
            context.whileEnd();
        }
    }
 )
| "break"
    {
            if (context.getLoopNest() == 0) {
                    throw new RException("break without enclosing while statement");
            }
            context.hitBreak();
    }
| "continue"
    {
            if (context.getLoopNest() == 0) {
                    throw new RException("continue without enclosing while statement");
            }
            context.hitContinue();
    }
| #("return" a=walktree)
    {
            if (a != null) {
                    context.assignReturnValue(a);
            }
            else {
                    throw new RException("Function must return a value");
            }
            context.hitReturn();
    }
| #(RFNCALL
    {
            //Call internal function
            String fName = _t.getText();
            java.util.Vector funcArgs = new java.util.Vector();
            AST tArgs = _t.getNextSibling();
            while(tArgs != null) {
                    a = walktree(tArgs);
                    funcArgs.add(a);
                    tArgs = tArgs.getNextSibling();
            }
            r = RFunctions.callRFunction(fName, funcArgs);
    }
 )
| #(FNCALL
    {
            //Call user function
            String fName = _t.getText();
            java.util.Vector vArgs = new java.util.Vector();
            AST tArgs = _t.getNextSibling();
            while(tArgs != null) {
                    a = walktree(tArgs);
                    vArgs.add(a);
                    tArgs = tArgs.getNextSibling();
            }
            r = rip.executeUserFunction(this, fName, vArgs);
        }
   )
  | #(ID

            {
                    AST dims = _t;
                    if (dims ==  null) {
                            r = context.get(#ID.getText());
                    } else {
                            RType d;
                            java.util.Vector arrayDims = new java.util.Vector();
                            while(dims != null) {
                                    d = walktree(dims);
                                    arrayDims.add(d);
                                    dims = dims.getNextSibling();
                            }
                            r = context.get(#ID.getText(), arrayDims);
                    }

            }

    )
  ;
```

RArray.java

```
import java.util.Vector;

/**
 * Represents a multi dimensional array of arbitrary dimensions. Internally it
 * stores the array items as a single dimensional array of RType. The dimensions
 * of the array are set at declaration and cannot be modified later. The
 * algorithm in the access() function computes the actual index within the
 * single dimension that should be used to access the array elements.
 *
 * @author Rajiv S Kumar rk2268@columbia.edu
 *
 */
public class RArray extends RType {
        private int dataType;

        private RType[] theArray;

        private int[] dims;

        private int[] mult;

        public RArray(String token, int t, Vector aDims) throws RException {
                super(token);
                this.dataType = t;
                // Copy aDims to array dims
        if (aDims == null || aDims.size() == 0) {
            throw new RException("Cannot create null array " + name);
        }
        int nTotal = 1;
        dims = new int[aDims.size()];

        for (int a = 0; a < dims.length; a++) {
            dims[a] = (int) ((RInt) aDims.get(a)).var;
            if (dims[a] == 0) {
                throw new RException("dimension " + a + " cannot be zero for "
                        + name);
            }
            nTotal *= dims[a];
        }

        // System.out.println("Total elements = " + nTotal);

        // Create multiplicative factors for each dimension.
        // Example for a 4 dimen array if dims = { 4, 3, 7, 2 } then mult will
        // have
        // mult = {0, 42, 14, 2} this is the product of 3*7*2, 7*2, 2
        mult = new int[dims.length];
        mult[0] = 0;
        for (int m = 1; m < mult.length; m++) {
            mult[m] = 1;
            for (int n = m; n < dims.length; n++) {
                mult[m] *= dims[n];
            }
        }
        /*
       * for (int i = 0; i < mult.length; i++) { System.out.println("mult[" +
       * i + "] = " + mult[i]); }
       */

        // Create a single dimensional array large enough to hold the multi
        // dimensional array.
        theArray = new RType[nTotal];

        // We have created an array of references. Now
        // create the actual elements of the array
        for (int i = 0; i < theArray.length; i++) {
            theArray[i] = RFactory.create("[]", dataType);
        }
        // We have flattened out the multi dimensional array into a single
        // dimension.
    }

    public RType access(Vector v) throws RException {
        // Vector v holds the dimensions of the desired element.
        if (v == null || v.size() == 0) {
            throw new RException("Array access requires index");
        }

        int sz = v.size();

        if (sz != dims.length) {
            throw new RException("Cannot access " + dims.length
                    + " dimensional array with index of " + sz + " dimensions");
        }

        // Get last element in vector. assumes sz > 0;

        int arrIndx = (int) ((RInt) v.get(sz - 1)).var;
```

```
		if (arrIndx > dims[sz - 1] - 1) {
			// array bounds exception
			throw new RException(
					"Array access out of bounds for last dimension");
		}

		if (sz == 1) {
			// Single dimensional array. No computation necessary. Just return
			// b[arrIndx];
			return theArray[arrIndx];
		}

		for (int i = 0; i < sz - 1; i++) {
			int n = (int) ((RInt) v.get(i)).var;
			if (n > (dims[i] - 1)) {
				throw new RException(
						"Array access out of bounds for dimension " + i);
			}
			arrIndx += (n * mult[i + 1]);
		}
		// System.out.println("Returning element " + arrIndx);
		return theArray[arrIndx];
	}
}
```

RBool.java

```
/**
 * RBool represents the bool data type.
 *
 * @author Rajiv S Kumar rk2268@columbia.edu
 *
 * RBool is derived the Mx sample on the class web site.
 */
public class RBool extends RType {
	boolean var;

	public RBool() {
		var = false;
	}

	RBool(boolean var) {
		this.var = var;
	}

	public int dataType() {
		return RType.RTypeBool;
	}

	public String displayType() {
		return "bool";
	}

	public RType copy() {
		return new RBool(var);
	}

	public RType and(RType b) throws RException {
		if (b instanceof RBool)
			return new RBool(var && ((RBool) b).var);
		throw new RException("bool && " + b.displayType() + " not supported");
	}
	public RType or(RType b) throws RException {
		if (b instanceof RBool)
			return new RBool(var || ((RBool) b).var);
		throw new RException("bool || " + b.displayType() + " not supported");
	}

	public RType not() {
		return new RBool(!var);

	}

	public RType eq(RType b) throws RException {
		// not exclusive or
		if (b instanceof RBool)
			return new RBool((var && ((RBool) b).var)
					|| (!var && !((RBool) b).var));
		throw new RException("bool == " + b.displayType() + " not supported");
	}

	public RType ne(RType b) throws RException {
		// exclusive or
		if (b instanceof RBool)
			return new RBool((var && !((RBool) b).var)
					|| (!var && ((RBool) b).var));
		throw new RException("bool != " + b.displayType() + " not supported");
	}

	public String toString() {
		return Boolean.toString(var);
	}
}
```

RContext.java

```
import java.util.Vector;

/**
 * RContext represents a function context. Each context knows about the
 * associated function. RContext has the functions to add variables into the
 * symbol table, assign variable values, lookup symbol table values and maintain
 * some state information for loops. The main context within which the code in
 * the R { } body executes is always present. The function contexts for user
 * functions are created dynamically upon function invocation.
 *
 * @author Rajiv S Kumar rk2268@columbia.edu
 *
 */
public class RContext {

	private RSymbolTable rst = null;

	private RUserFunc associatedFunc = null;

	public RContext(RSymbolTable parent, RUserFunc assocFunc) {
		rst = new RSymbolTable(parent);
		associatedFunc = assocFunc;
	}

	public RSymbolTable getSymbolTable() {
		return rst;
	}
```

```java
public RType put(String token, int t) throws RException {
	RType var = RFactory.create(token, t);
	if (var != null) {
		rst.put(var);
	}
	return var;
}

public RType put(String token, int t, java.util.Vector vDims)
		throws RException {
	RType var = new RArray(token, t, vDims);
	if (var != null) {
		rst.put(var);
	}
	return var;
}

public RType get(String token) throws RException {
	return rst.get(token);
}

public RType get(String token, Vector vDims) throws RException {
	RType temp = rst.get(token);
	if (temp == null) {
		return null;
	}
	if (temp instanceof RArray) {
		RArray rArray = (RArray) temp;
		return rArray.access(vDims);
	} else {
		throw new RException("Trying to access non array type " + token
				+ " as an array");
	}
}

public void assignReturnValue(RType b) throws RException {
	if (associatedFunc != null) {
		RType ret = associatedFunc.getReturnValue();
		assign(ret, b);
	}
}

public RType assign(RType a, RType b) throws RException {

	if (a instanceof RBool) {
		if (b instanceof RBool) {
			((RBool) a).var = ((RBool) b).var;
		} else {
			throw new RException("Cannot assign non bool type to " + a.name);
		}
	} else if (a instanceof RInt) {
		if (b instanceof RInt) {
			((RInt) a).var = ((RInt) b).var;
		} else if (b instanceof RDouble) {
			System.out.println("Trying to assign float to int \"" + a.name
					+ "\" possible loss of data");
			((RInt) a).var = (int) ((RDouble) b).var;
		} else {
			throw new RException("Cannot assign non numeric type to "
					+ a.name);
		}
	} else if (a instanceof RDouble) {
		if (b instanceof RInt) {
			((RDouble) a).var = ((RInt) b).var;
		} else if (b instanceof RDouble) {
```

```
                        ((RDouble) a).var = ((RDouble) b).var;
                } else {
                        throw new RException("Cannot assign non numeric type to "
                                        + a.name);
                }
        }
        if (a instanceof RString) {
                if (b instanceof RString) {
                        ((RString) a).var = ((RString) b).var;
                } else {
                        throw new RException("Cannot assign non string type to "
                                        + a.name);
                }
        }
        return a;
}

public RInt getInt(String token) {
        return new RInt(Integer.parseInt(token));
}

public RDouble getDouble(String token) {
        return new RDouble(Double.parseDouble(token));
}

public RString getString(String token) {
        return new RString(token);
}

private int loopNest = 0;

public static final int OkToProceed = 0;

public static final int HitBreak = 1;

public static final int HitContinue = 2;

public static final int HitReturn = 3;

private int currentState = OkToProceed;

public void hitBreak() {
        // System.out.println("hitBreak()");
        currentState = HitBreak;
}

public void hitContinue() {
        // System.out.println("hitContinue()");
        currentState = HitContinue;
}

public void hitReturn() {
        // System.out.println("hitReturn()");
        currentState = HitReturn;
}

public void whileBegin() {
        // System.out.println("whileBegin()");
        loopNest++;
}

public void whileEnd() {
        // System.out.println("whileEnd()");
        currentState = OkToProceed;
        loopNest--;
```

```
	}

	public int getCurrentState() {
		return currentState;
	}

	public int getLoopNest() {
		return loopNest;
	}

	public void setCurrentState(int s) {
		currentState = s;
	}

	public boolean shouldProceed() {
		return currentState == OkToProceed;
	}

	public void print() {
		rst.printSymbolTable();
	}

}
```

RDouble.java

```
/**
 * RDouble represents the float data type.
 *
 * @author Rajiv S Kumar - rk2268@columbia.edu
 *
 * RDouble is derived from of the Mx sample on the class web site.
 */

public class RDouble extends RType {
	double var; // Internally we use a double to store a float.

	public RDouble() {

	}

	public RDouble(double x) {
		var = x;
	}

	public int dataType() {
		return RType.RTypeDouble;
	}

	public String displayType() {
		return "float";
	}

	public RType copy() {
		return new RDouble(var);
	}

	public static double doubleValue(RType b) throws RException {
		if (b instanceof RDouble)
			return ((RDouble) b).var;
		if (b instanceof RInt)
			return (double) ((RInt) b).var;
		b.error("cannot convert " + b + " to float");
		return 0;
	}

	public RType uminus() {
		return new RDouble(-var);
	}

	public RType plus(RType b) throws RException {
		return new RDouble(var + doubleValue(b));
	}

	public RType minus(RType b) throws RException {
		return new RDouble(var - doubleValue(b));
	}

	public RType mul(RType b) throws RException {
		return new RDouble(var * doubleValue(b));
	}

	public RType div(RType b) throws RException {
		if (b instanceof RDouble)
			return new RDouble(var / doubleValue(b));
		return new RDouble(var / RDouble.doubleValue(b));
	}

	public RType modulus(RType b) throws RException {
		return new RDouble(var % doubleValue(b));
	}

	public RType gt(RType b) throws RException {
		return new RBool(var > doubleValue(b));
	}

	public RType ge(RType b) throws RException {
		return new RBool(var >= doubleValue(b));
	}

	public RType lt(RType b) throws RException {
		return new RBool(var < doubleValue(b));
	}

	public RType le(RType b) throws RException {
		return new RBool(var <= doubleValue(b));
	}

	public RType eq(RType b) throws RException {
		return new RBool(var == doubleValue(b));
	}

	public RType ne(RType b) throws RException {
		return new RBool(var != doubleValue(b));
	}

	public String toString() {
		return Double.toString(var);
	}

}
```

REngine.java

```
import java.io.DataInputStream;
import java.io.FileInputStream;

import antlr.debug.misc.ASTFrame;

/**
 * REngine is the starting point of the project. It loads the lexer, parser and
 * tree walker to execute the input script. The script to execute is passed as a
 * command line argument along with the call properties.
 *
 * @author Rajiv S Kumar - rk2268@columbia.edu
 */

public class REngine {

	public static void main(String[] args) {
		boolean debug = false;
		try {

			if (args.length == 0) {
				System.out
						.println("Usage java REngine <scriptName> "
								+ "[Call properties in the format
key1=value1;key2=value2;key3=value3...]");
				System.out
						.println("Example:\njava REngine 8001.r
callerId=9149090965;acctNumber=4567890;");
				return;
			}

			if (args.length > 1) {
				setCallProperties(args[1]);
			}

			FileInputStream filename = new FileInputStream(args[0]);
			DataInputStream input = new DataInputStream(filename);
			RLexer lexer = new RLexer(new DataInputStream(input));
			RParser parser = new RParser(lexer);
			parser.program();
			antlr.CommonAST ast = (antlr.CommonAST) parser.getAST();
			if (debug) {
				ASTFrame frame = new ASTFrame("AST", ast);
				frame.setVisible(true);
			}
			RWalker walker = new RWalker();

			// prewalktree goes through the AST once
			// to find and store user function pointers.
			walker.prewalktree(ast);

			// The actual execution of the AST
			walker.program(ast);

			// After the script completes check the target variable.
			// This tells the engine where to send the call.
			RString target = (RString) walker.rip.getMainContext()
					.get("target");
			if (target == null || target.toString().length() == 0) {
				System.out.println("Target not set. Call will be dropped");
			} else {
				// If the script sets the target, we would send the call there.
				// As of now we only print the target to console.
				// In real life, the code to send the call to the target would
				// be added here.
				System.out.println("Routing call to " + target.toString());
			}

		} catch (Exception e) {
			System.out.println(e.getMessage());
			System.out.println("Exception processing script: " + args[0]);
			System.out.println("Stopping execution of script");
			if (debug) {
				e.printStackTrace();
			}
		}
	}

	private static void setCallProperties(String s) {
		String[] kvpairs = s.split(";");
		if (kvpairs.length > 0) {
			for (int i = 0; i < kvpairs.length; i++) {
				String[] kv = kvpairs[i].split("=");
				if (kv.length == 2) {
					RFunctions.setCallProperty(kv[0], kv[1]);
				} else {
					System.out
							.println("Key value pair should be in the format key=value");
				}
			}
		}
	}

}
```

RException.java

```
public class RException extends Exception {
	RException(String msg) {
		super(msg);
	}
}
```

RFactory.java

```
/**
 * RFactory creates the R data types based on the corresponding types in the
 * script.
 *
 * @author Rajiv S Kumar rk2268@columbia.edu
 *
 */
public class RFactory {

	public static RType create(String token, int t) throws RException {
		RType var = null;
		switch (t) {
		case RType.RTypeBool:
			var = new RBool();
			break;
		case RType.RTypeInt:
			var = new RInt();
			break;
		case RType.RTypeDouble:
			var = new RDouble();
```

```
                break;
            case RType.RTypeString:
                var = new RString();
                break;
            default:
                throw new RException("Unknown type " + t + " for token " + token);
            }
            if (var != null) {
                var.name = token;
            }
            return var;
        }

}
```

RFunctions.java

```
import java.util.Calendar;
import java.util.GregorianCalendar;
import java.util.Properties;
import java.util.Vector;

/**
 * RFunctions class has all the inbuilt functions of the R scripting language.
 * This includes functions like e.print(), type conversion functions, string
 * manipulation routines, call properties functions and call statistics
 * functions.
 *
 * @author Rajiv S Kumar - rk2268@columbia.edu
 */

public class RFunctions {

        public static RType callRFunction(String funcName, Vector vArgs)
                throws RException {

            RType ret = new RType();
            if (funcName == null || funcName.length() == 0) {
                throw new RException("Function name cannot be empty");
            }

            if (funcName.compareTo("print") == 0) {
                if (vArgs.size() == 1) {
                    RType arg = (RType) vArgs.get(0);
                    print(arg.toString());
                } else {
                    throw new RException("print takes one argument");
                }
            } else if (funcName.compareTo("foo") == 0) {
                if (vArgs.size() == 1) {
                    RType arg = (RType) vArgs.get(0);
                    if (arg instanceof RInt) {
                        ret = new RInt(foo(((RInt) arg).var));
                    } else if (arg instanceof RDouble) {
                        ret = new RInt(foo((int) ((RDouble) arg).var));
                    } else {
                        throw new RException("foo argument should be numeric type");
                    }
                } else {
                    throw new RException("foo takes one argument");
                }
            } else if (funcName.compareTo("toS") == 0) {
                if (vArgs.size() != 1) {
                        throw new RException("toS takes one argument");
                }
                ret = toS((RType) vArgs.get(0));
        } else if (funcName.compareTo("toI") == 0) {
                if (vArgs.size() != 1) {
                        throw new RException("toI takes one argument");
                }
                ret = toI((RType) vArgs.get(0));
        } else if (funcName.compareTo("toF") == 0) {
                if (vArgs.size() != 1) {
                        throw new RException("toF takes one argument");
                }
                ret = toF((RType) vArgs.get(0));
        } else if (funcName.compareTo("getDateTime") == 0) {
                if (vArgs.size() != 1) {
                        throw new RException("getDateTime takes one argument");
                }
                if (((RType) vArgs.get(0)).dataType() != RType.RTypeInt) {
                        throw new RException("getDateTime argument should be an int");
                }
                ret = getDateTime((RInt) vArgs.get(0));
        } else if (funcName.compareTo("callProperty") == 0) {
                if (vArgs.size() != 1) {
                        throw new RException("callProperty takes one argument");
                }
                if (((RType) vArgs.get(0)).dataType() != RType.RTypeString) {
                        throw new RException("callProperty argument should be a string");
                }
                ret = callProperty((RString) vArgs.get(0));
        } else if (funcName.compareTo("len") == 0) {
                if (vArgs.size() != 1) {
                        throw new RException("len takes one argument");
                }
                if (((RType) vArgs.get(0)).dataType() != RType.RTypeString) {
                        throw new RException("len argument should be a string");
                }
                ret = len((RString) vArgs.get(0));
        } else if (funcName.compareTo("uCase") == 0) {
                if (vArgs.size() != 1) {
                        throw new RException("uCase takes one argument");
                }
                if (((RType) vArgs.get(0)).dataType() != RType.RTypeString) {
                        throw new RException("uCase argument should be a string");
                }
                ret = uCase((RString) vArgs.get(0));
        } else if (funcName.compareTo("lCase") == 0) {
                if (vArgs.size() != 1) {
                        throw new RException("lCase takes one argument");
                }
                if (((RType) vArgs.get(0)).dataType() != RType.RTypeString) {
                        throw new RException("lCase argument should be a string");
                }
                ret = lCase((RString) vArgs.get(0));
        } else if (funcName.compareTo("mid") == 0) {
                if (vArgs.size() != 3) {
                        throw new RException("mid takes 3 arguments");
                }
                if (((RType) vArgs.get(0)).dataType() != RType.RTypeString) {
                        throw new RException("mid argument 1 should be a string");
                }
                if (((RType) vArgs.get(1)).dataType() != RType.RTypeInt) {
                        throw new RException("mid argument 2 should be an int");
                }
                if (((RType) vArgs.get(2)).dataType() != RType.RTypeInt) {
                        throw new RException("mid argument 3 should be an int");
```

```
			}
			ret = mid((RString) vArgs.get(0), (RInt) vArgs.get(1), (RInt) vArgs
					.get(2));
		} else if (funcName.compareTo("isIn") == 0) {
			if (vArgs.size() != 2) {
				throw new RException("isIn takes 2 arguments");
			}
			if (((RType) vArgs.get(0)).dataType() != RType.RTypeString) {
				throw new RException("isIn argument 1 should be a string");
			}
			if (((RType) vArgs.get(1)).dataType() != RType.RTypeString) {
				throw new RException("mid argument 2 should be a string");
			}

			ret = isIn((RString) vArgs.get(0), (RString) vArgs.get(1));
		} else if (funcName.compareTo("serviceLevel") == 0) {
			if (vArgs.size() != 1) {
				throw new RException("serviceLevel takes one argument");
			}
			if (((RType) vArgs.get(0)).dataType() != RType.RTypeString) {
				throw new RException("serviceLevel argument should be a string");
			}
			ret = serviceLevel((RString) vArgs.get(0));
		} else if (funcName.compareTo("estimatedWaitTime") == 0) {
			if (vArgs.size() != 1) {
				throw new RException("estimatedWaitTime takes one argument");
			}
			if (((RType) vArgs.get(0)).dataType() != RType.RTypeString) {
				throw new RException(
						"estimatedWaitTime argument should be a string");
			}
			ret = estimatedWaitTime((RString) vArgs.get(0));
		}

		return ret;
	}

	// foo is used only for testing.
	public static long foo(long i) {
		return i + 100;
	}

	/**
	* Prints input to console.
	*
	* @param str
	*/

	public static void print(String str) {
		System.out.println(str);
	}

	/**
	* Converts input to a string
	*
	* @param in
	* @return
	*/
	public static RString toS(RType in) {
		return new RString(in.toString());
	}

	/**
	* Converts input to an int
```

```
*
* @param in
* @return
*/
public static RInt toI(RType in) throws RException {
	RInt ret = null;
	try {
		if (in instanceof RInt) {
			ret = new RInt(((RInt) in).var);
		} else if (in instanceof RDouble) {
			ret = new RInt((long) ((RDouble) in).var);
		} else if (in instanceof RString) {
			ret = new RInt((long) Double.parseDouble(in.toString()));
		} else {
			throw new RException("Error converting " + in.toString()
					+ " to  int");
		}

	} catch (Exception ex) {
		print(ex.getMessage());
		throw new RException("Error converting " + in.toString()
				+ " to  int");
	}
	return ret;
}

/**
* Converts input to a float
*
* @param in
* @return
*/
public static RDouble toF(RType in) throws RException {
	RDouble ret = null;
	try {
		if (in instanceof RInt) {
			ret = new RDouble(((RInt) in).var);
		} else if (in instanceof RDouble) {
			ret = new RDouble(((RDouble) in).var);
		} else if (in instanceof RString) {
			ret = new RDouble(Double.parseDouble(in.toString()));
		} else {
			throw new RException("Error converting " + in.toString()
					+ " to  int");
		}
	} catch (Exception ex) {
		print(ex.getMessage());
		throw new RException("Error converting " + in.toString()
				+ " to  float");
	}
	return ret;
}

/**
* Gets the date and time using java.util.GregorianCalendar. The input field
* is the same as the input for the get() method of java.util.Calendar.
* Please check java.sun.com for documentation of this function.
*
* @param field
* @return
*/
public static RInt getDateTime(RInt field) throws RException {
	RInt ret = null;
	Calendar cal = new GregorianCalendar();
```

```
        try {
                ret = new RInt(cal.get((int) field.var));
        } catch (Exception ex) {
                throw new RException("getDateTime Error fetching requested value");
        }
        return ret;
}

/**
* Stores the call properties passed as command line inputs to REngine for
* the script.
*/
private static Properties callProperties = new Properties();

/**
* Lookup call property based on key. The key is the input.
*
* @param strKey
* @return
*/
public static RString callProperty(RString strKey) {
        return new RString(callProperties.getProperty(strKey.toString(), ""));
}

public static void setCallProperty(String key, String value) {
        callProperties.put(key, value);
}

/**
* Returns the length of a string
*
* @param strIn
* @return
*/
public static RInt len(RString strIn) {
        return new RInt(strIn.var.length());
}

/**
* Returns the sub string of a given string.
*
* @param strIn
* @param nStart
* @param nEnd
* @return
*/
public static RString mid(RString strIn, RInt nStart, RInt nEnd) {
        return new RString(strIn.var
                        .substring((int) nStart.var, (int) nEnd.var));
}

/**
* Finds the occurance of one string within another.
*
* @param strIn
* @param strSearch
* @return
*/
public static RInt isIn(RString strIn, RString strSearch) {
        return new RInt(strIn.var.indexOf(strSearch.var));
}
```

```
	/**
	* Converts a string to uppoer case.
	*
	* @param strIn
	* @return
	*/
	public static RString uCase(RString strIn) {
		return new RString(strIn.var.toUpperCase());
	}

	/**
	* Converts a string to lower case.
	*
	* @param strIn
	* @return
	*/
	public static RString lCase(RString strIn) {
		return new RString(strIn.var.toLowerCase());
	}

	/**
	* Dummy function to imitate serviceLevel for different queues. In real
	* life, these values would come from a statistics server. Please see
	* chapter on Real life scenario in the final report.
	*
	* @param strQueue
	* @return
	*/
	public static RDouble serviceLevel(RString strQueue) {
		if (strQueue.var.compareTo("7800") == 0) {
			return new RDouble(0.8);
		} else if (strQueue.var.compareTo("7801") == 0) {
			return new RDouble(0.9);
		} else if (strQueue.var.compareTo("7802") == 0) {
			return new RDouble(0.75);
		}
		return new RDouble(0.0);
	}

	/**
	* Dummy function to imitate estimated wait times for different queues. In
	* real life, these values would come from a statistics server. Please see
	* chapter on Real life scenario in the final report.
	*
	* @param strQueue
	* @return
	*/

	public static RInt estimatedWaitTime(RString strQueue) {
		if (strQueue.var.compareTo("7800") == 0) {
			return new RInt(25);
		} else if (strQueue.var.compareTo("7801") == 0) {
			return new RInt(50);
		} else if (strQueue.var.compareTo("7802") == 0) {
			return new RInt(60);
		}
		return new RInt(0);
	}

}
```

RInt.java

```
/**
 * Represents the int data type.
 *
 * @author Rajiv S Kumar - rk2268@columbia.edu
 *
 * RInt is derived from the Mx sample on the class web site.
 */

class RInt extends RType {
	long var; // Internally we use a long to store an int.

	public RInt() {
		var = 0;
	}

	public RInt(long x) {
		var = x;
	}

	public int dataType() {
		return RType.RTypeInt;
	}

	public String displayType() {
		return "int";
	}

	public RType copy() {
		return new RInt(var);
	}

	public static long intValue(RType b) throws RException {
		if (b instanceof RDouble)
			return (long) ((RDouble) b).var;
		if (b instanceof RInt)
			return ((RInt) b).var;

		b.error("cannot convert " + b + " to int");
		return 0;
	}

	public RType uminus() {
		return new RInt(-var);
	}

	public RType plus(RType b) throws RException {
		if (b instanceof RInt)
			return new RInt(var + intValue(b));
		return new RDouble(var + RDouble.doubleValue(b));
	}

	public RType minus(RType b) throws RException {
		if (b instanceof RInt)
			return new RInt(var - intValue(b));
		return new RDouble(var - RDouble.doubleValue(b));
	}

	public RType mul(RType b) throws RException {
		if (b instanceof RInt)
			return new RInt(var * intValue(b));
		return new RDouble(var * RDouble.doubleValue(b));
	}
	public RType div(RType b) throws RException {
		if (b instanceof RInt)
			return new RInt(var / intValue(b));
		return new RDouble(var / RDouble.doubleValue(b));
	}

	public RType modulus(RType b) throws RException {
		if (b instanceof RInt)
			return new RInt(var % intValue(b));
		return new RDouble(var % RDouble.doubleValue(b));
	}

	public RType gt(RType b) throws RException {
		if (b instanceof RInt)
			return new RBool(var > intValue(b));
		return b.lt(this);
	}

	public RType ge(RType b) throws RException {
		if (b instanceof RInt)
			return new RBool(var >= intValue(b));
		return b.le(this);
	}

	public RType lt(RType b) throws RException {
		if (b instanceof RInt)
			return new RBool(var < intValue(b));
		return b.gt(this);
	}

	public RType le(RType b) throws RException {
		if (b instanceof RInt)
			return new RBool(var <= intValue(b));
		return b.ge(this);
	}

	public RType eq(RType b) throws RException {
		if (b instanceof RInt)
			return new RBool(var == intValue(b));
		return b.eq(this);
	}

	public RType ne(RType b) throws RException {
		if (b instanceof RInt)
			return new RBool(var != intValue(b));
		return b.ne(this);
	}

	public String toString() {
		return Long.toString(var);
	}

}
```

RInterpreter.java

```
import java.util.Hashtable;
import java.util.Vector;

import antlr.collections.AST;

/**
```

```
 * RInterpreter is the loaded by the tree walker to execute the AST. It holds
 * the main context and has a pointer to the current context object. It also has
 * code to execute user defined functions.
 *
 * @author Rajiv S Kumar rk2268@columbia.edu
 *
 */
public class RInterpreter {

	private RContext mainContext;

	private RContext currentContext;

	private Hashtable userFuncTable = new Hashtable();

	private RInterpreter() {
		mainContext = new RContext(null, null);
		currentContext = mainContext;

		// We always have a predefined variable named target.
		try {
			mainContext.put(new String("target"), RType.RTypeString);
		} catch (RException e) {
			System.out.println("Unable to set target in symbol table");
		}
	}

	private static RInterpreter onlyInstance = null;

	public static RInterpreter getInstance() {
		if (onlyInstance == null) {
			onlyInstance = new RInterpreter();
		}
		return onlyInstance;
	}

	/*
	 * public static void fatalError(String msg) throws RException { throw new
	 * RException(msg); }
	 */

	public RContext getMainContext() {
		return mainContext;
	}

	public RContext getCurrentContext() {
		return currentContext;
	}

	public void addUserFunction(RUserFunc uFunc) throws RException {

		// System.out.println("Adding User function:");
		// uFunc.printDebugInfo();
		RUserFunc ret = (RUserFunc) userFuncTable.get(uFunc.getName());
		if (ret == null) {
			userFuncTable.put(uFunc.getName(), uFunc);
		} else {
			throw new RException("User function " + uFunc.getName()
					+ " redifined");
		}
	}

	public RUserFunc getUserFunction(String token) {
		return (RUserFunc) userFuncTable.get(token);
```

```
}

public RType executeUserFunction(RWalker rWalker, String name, Vector vArgs)
		throws RException {

	RUserFunc ruf = getUserFunction(name);
	if (ruf == null) {
		throw new RException("Attempt to call undefined function " + name);
	}

	// Check vargs size against args
	Vector funcArgs = ruf.getArgs();
	if (funcArgs.size() != vArgs.size()) {
		throw new RException("Error: Attempt to pass " + vArgs.size()
				+ " arguments to function " + name + " which takes "
				+ funcArgs.size() + " arguments");
	}

	// Check vargs types against funcArgs
	for (int i = 0; i < funcArgs.size(); i++) {
		RType funcArg = (RType) funcArgs.get(i);
		RType argPassed = (RType) vArgs.get(i);
		if (!checkArgCompatiblity(funcArg, argPassed)) {
			throw new RException("Function call " + name
					+ ": cannot convert " + " arg " + (i + 1) + " from "
					+ argPassed.displayType() + " to "
					+ funcArg.displayType());
		}
	}

	// Save previous context
	RContext tempContext = currentContext;
	RContext funcContext = ruf.createContext();
	currentContext = funcContext;
	// funcContext.reset();

	// Copy values for args in symbol table.

	for (int i = 0; i < funcArgs.size(); i++) {
		RType funcArg = (RType) funcArgs.get(i);
		RType argPassed = (RType) vArgs.get(i);
		funcContext.assign(funcArg, argPassed);
	}

	// Start executing AST.
	AST funcBody = ruf.getFuncBody();

	try {
		rWalker.walktree(funcBody);
	} catch (Exception ex) {
		ex.printStackTrace();
		throw new RException("Error executing function " + name
				+ " Message = " + ex.getMessage());
	}

	// After executing the function, set context back to what it was before.
	currentContext = tempContext;
	return ruf.getReturnValue();

}

private boolean checkArgCompatiblity(RType funcArg, RType argPassed) {
	int f = funcArg.dataType();
	int a = argPassed.dataType();
	if (f == a) {
			return true;
		}
		if (f == RType.RTypeDouble && a == RType.RTypeInt) {
			return true;
		}

		return false;
	}

}
```

RString.java

```
/**
 * Represents the string data type.
 *
 * @author Rajiv S Kumar - rk2268@columbia.edu
 *
 * RString is derived from the Mx sample on the class web site.
 */

public class RString extends RType {
	String var;

	public RString() {
		var = "";
	}

	public RString(String str) {
		this.var = str;
	}

	public int dataType() {
		return RType.RTypeString;
	}

	public String displayType() {
		return "string";
	}

	public RType copy() {
		return new RString(var);
	}

	public RType plus(RType b) {
		return new RString(var + b.toString());
	}

	public RType gt(RType b) throws RException {
		if (b instanceof RString)
			return new RBool(var.compareTo(((RString) b).var) > 0);
		return b.lt(this);
	}

	public RType ge(RType b) throws RException {
		if (b instanceof RString)
			return new RBool(var.compareTo(((RString) b).var) >= 0);
		return b.le(this);
	}

	public RType lt(RType b) throws RException {
		if (b instanceof RString)
			return new RBool(var.compareTo(((RString) b).var) < 0);
		return b.gt(this);
	}

	public RType le(RType b) throws RException {
		if (b instanceof RString)
			return new RBool(var.compareTo(((RString) b).var) <= 0);
		return b.ge(this);
	}

	public RType eq(RType b) throws RException {
		if (b instanceof RString)
			return new RBool(var.compareTo(((RString) b).var) == 0);
		return b.eq(this);
	}

	public RType ne(RType b) throws RException {
		if (b instanceof RString)
			return new RBool(var.compareTo(((RString) b).var) != 0);
		return b.ne(this);
	}

	public String toString() {
		return var;
	}

}
```

RSymbolTable.java

```
import java.util.Enumeration;
import java.util.Hashtable;

/**
 * Represents the symbol table. Internally uses a hash table to map names and
 * variables. It has a pointer to the parent symbol table (of the main context).
 *
 * @author Rajiv S Kumar rk2268@columbia.edu
 *
 */
public class RSymbolTable {
	private Hashtable table;

	protected RSymbolTable parent;

	public RSymbolTable(RSymbolTable st) {
		table = new Hashtable();
		parent = st;
	}

	public void put(RType r) throws RException {
		RType temp = (RType) table.get(r.name);
		if (temp != null) {
			throw new RException(r.name + " redifned");
		}
		table.put(r.name, r);
	}

	public RType get(String token) throws RException {

		RType t = (RType) table.get(token);
		if (t == null) {
			if (parent != null) {
				t = (RType) parent.get(token);
			}
		}
		if (t == null) {
			throw new RException("Undefined symbol " + token);
		}
		return t;
	}

	public void printSymbolTable() {
		for (Enumeration e = table.keys(); e.hasMoreElements();) {
			String key = (String) e.nextElement();
			RType val = (RType) table.get(key);
			if (val != null) {
				System.out.println(key + " = " + val.toString());
			}
		}
		if (parent != null) {
			System.out.println("Parent Symbol table:");
			parent.printSymbolTable();
		}
	}

}
```

RType.java

```
/**
 * Represents an abstract data type.
 *
 * @author Rajiv S Kumar - rk2268@columbia.edu
 *
 * RType is derived from the Mx sample on the class web site.
 */

public class RType {
	public static final int RTypeUnknown = 0;

	public static final int RTypeBool = 1;

	public static final int RTypeInt = 2;

	public static final int RTypeDouble = 3;

	public static final int RTypeString = 4;

	String name; // used in hash table

	public RType() {
		name = null;
	}

	public RType(String name) {
		this.name = name;
	}

	public int dataType() {
		return RTypeUnknown;
	}

	public String displayType() {
		return "unknown";
	}
```

```
public RType copy() {
	return new RType();
}

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

public RType error(String msg) throws RException {
	throw new RException(msg);
}

public RType assign(RType b) throws RException {
	throw new RException(displayType() + " does not implement = ");
}

public RType uminus() throws RException {
	throw new RException(displayType() + " does not implement - ");
}

public RType plus(RType b) throws RException {
	throw new RException(displayType() + " does not implement + ");
}

public RType minus(RType b) throws RException {
	throw new RException(displayType() + " does not implement - ");
}

public RType mul(RType b) throws RException {
	throw new RException(displayType() + " does not implement * ");
}

public RType div(RType b) throws RException {
	throw new RException(displayType() + " does not implement / ");
}

public RType modulus(RType b) throws RException {
	throw new RException(displayType() + " does not implement % ");
}

public RType gt(RType b) throws RException {
	throw new RException(displayType() + " does not implement > ");
}

public RType ge(RType b) throws RException {
	throw new RException(displayType() + " does not implement >= ");
}

public RType lt(RType b) throws RException {
	throw new RException(displayType() + " does not implement < ");
}

public RType le(RType b) throws RException {
	throw new RException(displayType() + " does not implement <= ");
}

public RType eq(RType b) throws RException {
	throw new RException(displayType() + " does not implement == ");
}

public RType ne(RType b) throws RException {
	throw new RException(displayType() + " does not implement != ");
}

public RType and(RType b) throws RException {
		throw new RException(displayType() + " does not implement && ");
	}

	public RType or(RType b) throws RException {
		throw new RException(displayType() + " does not implement || ");
	}

	public RType not() throws RException {
		throw new RException(displayType() + " does not implement ! ");
	}
}
```

RUserFunc.java

```
import java.util.Vector;

import antlr.collections.AST;

/**
 * RUserFunc represents a user function. It stores the list of arguments and the
 * AST for the function body. It also creates the function context when the user
 * function is invoked.
 *
 * @author Rajiv S Kumar - rk2268@columbia.edu
 *
 */
public class RUserFunc {

	private RContext funcContext;

	private AST funcBody;

	private String name;

	private int retType;

	private Vector args;

	RType returnValue = null;

	public RUserFunc(String funcName, int returnType, Vector arguments,
			AST funcBody) {
		RInterpreter rip = RInterpreter.getInstance();
		this.name = funcName;
		this.retType = returnType;
		this.args = arguments;
		this.funcBody = funcBody;
	}

	public String getName() {
		return name;
	}

	public int getRetType() {
		return retType;
	}

	public Vector getArgs() {
		return args;
	}

	RContext createContext() throws RException {
		funcContext = new RContext(RInterpreter.getInstance().getMainContext()
				.getSymbolTable(), this);
		// Create a variable to hold return type. I'll use VB convention
		// and make return var name same as function name
		returnValue = funcContext.put(this.name, this.retType);

		// Set arguments as local variables in symbol table.
		if (args != null) {
			for (int i = 0; i < args.size(); i++) {
				RType r = (RType) args.get(i);
				funcContext.getSymbolTable().put(r);
			}
		}
		return funcContext;
	}


	RType getReturnValue() {
		return returnValue;
	}

	AST getFuncBody() {
		return funcBody;
	}

	public void printDebugInfo() {
		System.out.println("Func name: " + name);
		System.out.println("Func ret type: " + retType);
		System.out.println("Func args: Len = " + args.size());
		for (int i = 0; i < args.size(); i++) {
			RType r = (RType) args.get(i);
			System.out.println("\t" + r.name + "=" + r.toString());
		}
		// ASTFrame frame = new ASTFrame("Function " + this.name, funcBody);
		// frame.setVisible(true);
	}


}
```

Test Scripts

testExpr.r

```
R {

//This sample tests all kinds of expressions in R

int n = 10;

int y = n + 5;
e.print(y);

y = n - 5;
e.print(y);

y = n * 5;
e.print(y);

y = n / 2;
```

```
e.print(y);

float f =  (n + n) * 1.0/(n - 1);
e.print(f);

e.print("n = " + n);

//Test == operator on ints
if (n == 10) e.print("n == 10");

if (n != 20) e.print("n != 20 is true");

//Test > operator on ints
if (n > 1) e.print("n > 1 is true");
else e.print("n > 1 is false");

//Test < operator on ints
if (n < 100) e.print("n < 100 is true");
else e.print("n < 100 is false");

//Test >= operator on ints
if (n >= 5) e.print("n >= 5 is true");
else e.print("n >= 5 is false");

//Test >= operator on ints
if (n >= 10) e.print("n >= 10 is true");
else e.print("n >= 10 is false");

//Test <= operator on ints
if (n <= 50) e.print("n <= 50 is true");
else e.print("n <= 50 is false");

//Test <= operator on ints
if (n <= 10) e.print("n <= 10 is true");
else e.print("n <= 10 is false");

//Test evaluated expressions on both RHS and LHS
if ((n + 3) > (n + 2)) e.print("n+3 > n+2");

e.print("-----------------------\n");

f = 10.0;
e.print("f == " + f);

//Test == operator on floats
if (f == 10.0) e.print("f == 10.0 is true");

if (f == 10) e.print("f == 10 is true");

if (f != 20) e.print("f != 20 is true");

//Test > operator on floats
if (f > 1) e.print("f > 1 is true");
else e.print("f > 1 is false");

if (f > 1.0) e.print("f > 1.0 is true");
else e.print("f > 1.0 is false");

//Test < operator on floats
if (f < 100) e.print("f < 100 is true");
else e.print("f < 100 is false");

if (f < 100.0) e.print("f < 100.0 is true");
else e.print("f < 100.0 is false");
```

```
//Test >= operator on floats
if (f >= 5) e.print("f >= 5 is true");
else e.print("f >= 5 is false");

if (f >= 5.0) e.print("f >= 5.0 is true");
else e.print("f >= 5.0 is false");

//Test >= operator on floats
if (f >= 10.0) e.print("f >= 10.0 is true");
else e.print("f >= 10.0 is false");

//Test <= operator on floats
if (f <= 50.0) e.print("f <= 50.0 is true");
else e.print("f <= 50.0 is false");

//Test <= operator on floats
if (f <= 10) e.print("f <= 10 is true");
else e.print("f <= 10 is false");

if (f <= 10.0) e.print("f <= 10.0 is true");
else e.print("f <= 10.0 is false");

//Test evaluated expressions on both RHS and LHS
if ((f + 3) > (f + 2)) e.print("f+3 > f+2");

e.print("-----------------------\n");

//Test operators on bool expressions.
bool b;
bool c;
b = true;
c = false;

e.print("b == " + b);
e.print("c == " + c);

if (b == c) e.print("b == c is true");
else e.print("b == c is false");

if (b != c) e.print("b != c is true");
else e.print("b != c is false");

if (!b)  e.print("!b is true");
else e.print("!b is false");

if (!c)  e.print("!c is true");
else e.print("!c is false");

e.print("-----------------------\n");

//Test operators on strings.

string s1 = "ABC";
string s2 = "DEF";

e.print("s1 = " + s1);
e.print("s2 = " + s2);

if (s1 == "ABC")  e.print("s1 == \"ABC\" is true");
else e.print("s1 == \"ABC\" is false");

if (s1 != "ABC")  e.print("s1 != \"ABC\" is true");
else e.print("s1 != \"ABC\" is false");

if (s1 == s2)  e.print("s1 == s2 is true");
```

```
else e.print("s1 == s2 is false");

if (s1 != s2)  e.print("s1 != s2 is true");
else e.print("s1 != s2 is false");

string s3 = s1 + s2;
e.print("s3 = "+ s3);

}
```

testArrays.r

```
R {
	//Test arrays

	string y[3];

	y[0] = "he";
	y[1] = "haw";
	y[2] = "ha";

	e.print(y[0]);
	e.print(y[1]);
	e.print(y[2]);

	int x = 5;
	int a[x+5]; //Dynamic allocation of arrays

	a[9] = 3;
	e.print(a[9]);

	int i[2][3];
	i[0][0] = 0;
	i[0][1] = 1;
	i[0][2] = 2;
	i[1][0] = 3;
	i[1][1] = 4;
	i[1][2] = 5;

	e.print(i[0][0]);
	e.print(i[0][1]);
	e.print(i[0][2]);
	e.print(i[1][0]);
	e.print(i[1][1]);
	e.print(i[1][2]);

	int j[2][3][2];

	j[0][2][1] = 100;
	e.print(j[0][2][1]);

}
```

scope.r

```
R {
	int x = 100;
	foo();
	e.print("Back in main x = " +  x); //Prints 100

	[UserFunctions]
	int foo() {
		e.print("Inside foo x = " + x); //Prints 100 - the outer x.

		int x = 50; //Ok. x is local to foo.
		e.print("Local x = "+ x);  //Prints 50 - the local variable x.
		return 0;
	}

}
```

loops.r

```
R {

	//This sample tests while loops, break,
	//continue and return statements.

	int i;
	while (i < 20) {i = i + 1;}

	e.print("i = " + i);

	i = testBreak(10);
	e.print("testBreak returned i = " + i);

	int g;
	testReturn();
	e.print("after testReturn g = " + g);

	testContinue();
	e.print("after testContinue g = " + g);

	[UserFunctions]

	int testBreak(int j) {

		while (j < 20) {
			if (j == 15) {
				e.print("testBreak breaking");
				break;
			}
			j = j + 1;
			e.print("testBreak j = " +  j);
		}

		e.print("testBreak before return");
		return j;

	}


	int testReturn() {

		g = 0;
		while (g < 5) {
			if (g == 3) {
				e.print("testReturn returning");
				return 0;
			}
			g = g + 1;
			e.print("testReturn g = " +  g);
		}
		//We should

		e.print("testReturn return");
		return 0;
	}


	int testContinue() {
		g = 6;
		while (g < 11) {

			if (g == 8) {
				e.print("g is 8");
				g = g + 1;
				continue;
				e.print("shouldn't print this");
			}
			else {
				e.print("while loop g is " + g);
				g = g + 1;
			}

		}
		e.print("Out of while loop");
		return 0;
	}

}
```

typeConvert.r

```
R {

//Type conversion functions example


	string s = "345.67";

	float f = e.toF(s);
	e.print("float f = " + f);

	int i = e.toI(s);
	e.print("int i = " +  i);

	s = e.toS(f + 10.5);
	e.print("f + 10.5 = " + s);

	s = e.toS(i - 5);
	e.print("i - 5 = " + s);
}
```

testRecursion.r

```
R {
	//Try recursive function

	e.print(factorial(6));

	[UserFunctions]

	int factorial(int n) {
		int l = n;
```

```
            if (l == 0) return 1;
            else return l*factorial(l-1);
        }
}
```

StringCompares.r

```
R {

//The following example tests string comparision operators.

string a;
string b;
string c;

a = "abc";
b = "abc";

if (a == b) {
        e.print ("a==b");
} else if (a > b) {
        e.print("a > b");
} else if (a < b) {
        e.print("a < b");
}

if (a <= b) {
        e.print("a <= b");
}

if (a >= b) {
        e.print("a >= b");
}

}
```

inbuiltFuncs.r

```
R {

        string x[2];
        x[0] = "Hello";
        x[1] = "world";
        e.print(e.lCase(x[0]));
        e.print(e.uCase(x[1]));

        e.print(e.isIn(x[0], "ell"));
        e.print(e.mid(x[0], 1, 3));

        string y = x[0];
        e.print(y);
        int n = 100;
        e.print(n);

        int c = (n * 40)/2;
        e.print(c);
        target = e.toS(c);
        foo(3*4/2);

        int day;
        day = e.getDateTime(5);
        e.print("Day of month now is " + day);
```

```
	int month;
	month = e.getDateTime(2);
	e.print("Month now is " + (month + 1));

	int year;
	year = e.getDateTime(1);
	e.print("Year now is " + (year));

	float sl1; float sl2; float sl3;

	sl1 = e.serviceLevel("7800");
	sl2 = e.serviceLevel("7801");
	sl3 = e.serviceLevel("7802");

	target = "7800";
	if (sl2 > sl1) target = "7801";
	if (sl3 > sl2) target = "7802";

	e.print(target);

	[UserFunctions]
	bool foo(int n) {
		e.print("inside foo " + n  + " target = " + target);
		return true;
	}

}
```

testUserFuncs.r

```
R {

	//Declaration block
	foo2();
	target = "9724430400";

	[UserFunctions]

	int foo(int j) {

		while (j < 100) {
		if (j == 20) { e.print("foo breaking"); break;}
		j = j + 1;
		}

		e.print("foo before return");
		return j;

	}

	int bar() {

		g = 0;
		while (g < 100) {
		if (g == 20) { e.print("bar returning"); return "x";}
		g = g + 1;
		}

		e.print("bar return");
		return 0;
	}

	int foo2() {
        int n;
        n = 100;
        e.print("foo2 calling bar2");
        bar2();
        e.print("foo2 after calling bar2");
        e.print(n);

    }

    int bar2() {
        int bar2x;
        bar2x = 1000;
        e.print("inside bar2. calling foo");
        foo(10);
        e.print(bar2x);
    }

}
```

testDateTime.r

```
/**
* This is a sample R program that sends calls to agents during office hours (Mon-Fri 9 AM to 5 PM) and sends calls
to voice mail during after hours.
*/


R {

    //Get hour in 24 hr format
    int hour = e.getDateTime(11);
    e.print("Hour now is " + hour);

    //Get day of week with Sunday = 1
    int dayOfWeek = e.getDateTime(7);
    e.print("Day of week is " + dayOfWeek);

    target = "5601"; //voicemail number

    if (dayOfWeek >= 2 && dayOfWeek <= 6
        && hour >= 9 && hour < 17) {
            target = "8000"; //Office hours number.
    }

    e.print(target);

}
```

CallerLang.r

```
/**
* This is a sample R program that sends calls to different queues
* based on the caller’s language preference.
*/

R {

    /*
    e.print(e.callProperty("callerId"));
```

```
	e.print(e.callProperty("custAccount"));
	e.print(e.callProperty("callerLang"));
	*/


	if ( e.callProperty("callerLang") == "English") {
		target = "78001";
	} else if ( e.callProperty("callerLang") == "Spanish") {
		target = "79001";
	}


	e.print(target);

}
```

serviceLevel.r

```
R {
	/*
	* We have three queues. Select the
	* queue with the best service level.
	* Service level is the % of calls answered
	* within a particular threshold (such as 15 seconds).
	*
	*
	* The serviceLevel function has been hard coded to
	* return the following:
	* serviceLevel("7800") = 0.8
	* serviceLevel("7801") = 0.9
	* serviceLevel("7802") = 0.75
	*/

	float sl1; float sl2; float sl3;

	sl1 = e.serviceLevel("7800");
	sl2 = e.serviceLevel("7801");
	sl3 = e.serviceLevel("7802");

	target = "7800";
	if (sl2 > sl1) target = "7801";
	if (sl3 > sl2) target = "7802";
}
```

estimatedWaitTime.r

```
R {
	/*
	* We have three queues. Select the
	* queue with the lowest estimated time.
	*
	* Estimated wait time is the time the caller
	* is expected to wait in queue listening to music.
	*
	*
	* The estimatedWaitTime function has been hard coded to
	* return the following:
	*
```

```
      * estimatedWaitTime("7800") = 25
      * estimatedWaitTime("7801") = 50
      * estimatedWaitTime("7802") = 60
      */

       int ew1; int ew2; int ew3;

       ew1 = e.estimatedWaitTime("7800");
       ew2 = e.estimatedWaitTime("7801");
       ew3 = e.estimatedWaitTime("7802");

       target = "7800";
       if (ew2 < ew1) target = "7801";
       if (ew3 < ew2) target = "7802";
}
```

error1.r

```
R {


[UserFunctions]
int foo() {
return 0;
}

int x = 0; //Syntax error. No statements allowed after [UserFunctions]

}
```

error2.r

```
R {

y = 0; //undefined variable y

}
```

error3.r

```
R {

int x;
bool b;
b = x; //Cannot assign non bool type to b

}
```

error4.r

```
R {

int x;
x = "Hello"; //Cannot assign non numeric type to int

}
```

error5.r

```
R {

float x;
x = "Hello"; //Cannot assign non numeric type to float

}
```

error6.r

```
R {

string x;
x = 10; //Cannot assign non string type to x

}
```

error7.r

```
R {

string x;
string y;
x = x * y; //string does not implement *


}
```

error8.r

```
R {

bool b;
bool c;
b = b + c; //error bool does not implement +


}
```

error9.r

```
R {

int x;
bool b;
if (x || b) e.print("Hi"); //error. left side of || must be a bool expression

}
```

error10.r

```
R {
```

```
int x;
bool b;
if (b || x) e.print("Hi"); //error. bool || int not supported

}
```

error11.r

```
R {

int x;
bool b;
if (x && b) e.print("Hi"); //error. left side of && must be a bool expression

}
```

error12.r

```
R {

int x;
bool b = true;
if (b && x) e.print("Hi"); //error. bool && int not supported

}
```

error13.r

```
R {

int x;

if (x) e.print("Hi"); //error. if requires a boolean expression

}
```

error14.r

```
R {

break; //error. break without enclosing while

}
```

error15.r

```
R {

continue; //error. continue without enclosing while

}
```

error16.r

```
R {

foo();

[UserFunctions]

int foo() {
	return ; //syntax error. Unexpected token ; After return a return value must be present.
}

}
```

error17.r

```
R {

foo();

[UserFunctions]

int foo() {
	return 0;
}

//error. foo redefined
int foo() {
	return 0;
}

}
```

error18.r

```
R {

foo(); //error. undefined function foo

}
```

error19.r

```
R {

//error.
//Passing one argument to function foo which takes 2 args

foo(1);

[UserFunctions]

int foo(int x, int y) {
	return 0;
}
}
```

error20.r

```
R {

//error.Second argument to foo should have been an int.

foo(1, true);

[UserFunctions]

int foo(int x, int y) {
        return 0;
}

}
```

error21.r

```
R {

int x[0]; //error. Array dimension cannot be zero

}
```

error22.r

```
R {

int x[5][0]; //error. Second dimension cannot be zero

}
```

error23.r

```
R {

int x[5][2];

int y = x[4][2]; //Array access out of bounds

}
```

error24.r

```
R {

int x[5][2];

int y = x[5][1]; //Array access out of bounds

}
```

error25.r

```
R {

int x[5][2];

int y = x[5]; //Cannot access 2 dimensional array with single dimension

}
```

error26.r

```
R {

int y;
int x = y[0]; //error. y is not an array type

}
```

error27.r

```
R {

e.print(); //error. Print takes one argument

}
```

error28.r

```
R {

e.print("one", "two"); //error. Print takes one argument

}
```

error29.r

```
R {

int x;
int x; //x redefined

}
```

error30.r

```
R {
int x = 1;
//error. while requires a boolean expression
while (x) {
x = x + 1;
}

}
```