R – A Scripting language for a call routing engine

Language Reference Manual:

COMS W4115: Programming Languages and Translators

Rajiv S Kumar

rajiv.kumar@verizonbusiness.com

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, formfeeds 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:

break	int
char	return
continue	string

else	while
float	target
goto	
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. The bool, int and float data types have semantics very similar to the corresponding primitive types in the Java language. The string type represents an array of characters.

Program Structure

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

R {
declarations

statements

user defined functions
}

Execution begins at the top of the R script at the variable declarations and follows down to the statement block.

The variable declaration block will have the list of variables used in the script. R requires variables to be defined before they are used. All the variables declared in the declarations section will have global scope, which means they are accessible to all the functions. However, the variables defined inside user defined functions will have local scope.

The statement block will have a series of statements. 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. If no target is specified, the R-Engine will drop the call.

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 <Optional array brackets and size> Identifier;

Examples:

```
int i;
float x;
string s;
int[10] a;
```

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:

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

Labeled Statement:

A labeled statement begins with an identifier, followed by a colon and a statement. The labels are used by the goto statement. Labeled statements and goto will be implemented if time permits.

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. The R-Engine will provide 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
* /	Multiplication, Division	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

statement
…
statement
}

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

If time permits, for loop construct will be added.

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. The goto statement will be added if time permits.

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. They have the same syntactical structure as inbuilt functions. User defined functions should be defined after the statement block. Also, the block should begin with the attribute `[UserFunctions]`.

Sample R program:

```
/**
* This is a sample R program
*/

R {

	//Declaration block
	int a;
	float b;
	string x;
	bool m;

	//Statement block
	a = 3+4*5;
	b = (a/10) * a + (a - 1)/(4-2);
	b= -5.0;
	bar();
	a = foo(1, 2);
	//If a function call begins with e.
	//it is an R-Engine function
	x = e.CallProperties("AccountNumber");

	if (a > b) {
		e.print("a > b\n");
		m = true;
	} else {
	    e.print("a <= b\n");
```

```
	}
	target = a;

	//User defined functions
	[UserFunctions]

	string bar() {
		int n;
		float y;
		n = 5;
		n = foo (1, 3);
		return "Hello bar";
	}

	int foo (int z, float zz) {
		string x;
		return 0;
	}

}
```

ANTLR Grammar:

The following is a “Draft” version of the Grammar for the R Scripting language.

```
/*************************************************************************
                              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'
            |     'r'
            |     't'
            |     'b'
            |     'f'
            |     '"'
            |     '\''
            |     '\\'
            )
      ;

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 :    '/'  ;
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; DECLS; FUNCTIONDEFS; FNCALL; RFNCALL; ARGS;}

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

decls : (decl)*  { #decls = #([DECLS, "DECLS"], #decls); } ;

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

decl :  (basicType) (LBRACK! NUM RBRACK!)* ID SEMI! ;

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^) 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^ decls (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;
```