Random Data Generator Language (RDGL)

COMS W4115 Programming Languages and Translators
Professor Stephen A. Edwards
Summer 2007(CVN)

"Computer Science is as much about computers as astronomy is about telescopes."

- Edsger Dijkstra

Navid Azimi (na2258)
nazimi@microsoft.com

Contents

Introduction

Random Data Generator Language (RDGL) is a language which facilitates the generation of random data in a very flexible and powerful way. It is most analogous to regular expressions except instead of matching on input strings; RDGL generates output strings given an input expression.

Purpose

There are a number of different areas in which well-formed (or explicitly malformed), randomly generated data is useful. In software testing, for example, randomly generated data can help with security tests (fuzzing), stress tests, input validation, and even help increase overall test coverage by encouraging different data sets during each automated run.

Goals

The goal of this language is to create a simple and intuitive syntax which can facilitate complex expressions to generate data sets which are tailored yet random.

Portability

RDGL is translated to Java code and as a result is capable of running on any operating system or environment supported by Sun's JVM.

Execution

To simplify the execution and use of RDGL, a command shell has been developed which allows RDGL expressions to be executed directly from the command-prompt. The RDGL Shell is provided as a standalone application. However, it is still possible to write source (*.rdgl) plain text files and have them compiled into executable Java code as well.

Language Tutorial

The following is a brief language tutorial for a typical novice programmer. It assumes that the user has some experience with programming concepts (i.e. compilation) and computer languages (i.e. ANTLR, Java).

Before You Start

Before you can get started, please ensure you have all the project prerequisites installed and configured correctly. This includes JDK 5 or later (for compilation), JRE 5 or later (for execution), and ANTLR2 (to generate the parser and walker). Also, please ensure that you have the latest copy of the RDGL project. It is currently hosted on Google Code (here).

Note: the grammar files provided have been written using ANTLR2 and are incompatible with ANTLR3.

Getting Started

The simplest way to get started with RDGL is to use the interactive command shell. Each data type is signified by a single character token. The five most basic RDGL commands to learn include:

```
#     number
@     character
&     alphanumeric
$     symbol
%     wildcard
```

Single Token

To start, the command-shell should greet you with an `RDGL>` prompt. As an example, type `#` and hit return. The output should be similar to:

```
RDGL> #
>> 8

RDGL>
```

If you repeat this action, it is likely that a different number (between 0 and 9, inclusive) will be outputted. The same logic applies for the other data types as well where `@` outputs a single character, `&` outputs a single alphanumeric character, et al.

Predefined Length

To generate an alphanumeric string with a predefined length, type: `&[4]` where 4 indicates the length of the string to generate. Again, this syntax is consistent across all data types where `$[40]` would generate a random string of length 40 containing only symbols (no letters or numbers).

Varying Length

To generate a random word (letters only) of length 1, 3, or 5, one `@[1,3,5]`. For slightly more convoluted examples, see:

```
@[1-7, 10-15]     // generates a number with length between 1 and 7 or
                  // 10 and 15 (inclusive)

@[0,4-6,8]        // generates a number with length 0, 4, 5, 6, or 8.
```

There is no grammar imposed limit on the number of parameters that can be passed.

Numbers

It is often times necessary to generate a number between a minimum and maximum value. This is achieved in RDGL by providing the range of numbers in curly braces. The following examples should be self-explanatory:

```
#{1-12}           // generates a random number between 1 to 12
#{2007-2039}      // generates a random number between 2007 and 2039
```

To pick from a set of unordered numbers, please see enumerations.

Additional Features

There are a number of additional features not covered in the language tutorial. It is recommended that serious users read the language manual for additional information.

Language Manual

Overview

RDGL is comprised of approximately five data types which are used to generate a data type. The data types include: *number* (#), *character* (@), *alphanumeric* (&), *symbol* ($) and *wildcard* (%). As the name implies, alphanumeric is a combination of characters and numbers while wildcard is a combination of alphanumeric and symbols. These data types can then be constrained or modified accordingly (length, range, etc).

In addition to these data types, it is also possible to generate data based on a given set of values. These are called *enumerations*. There are two types of enumerations: normal (?) and nullable (*). The difference between is more closely discussed in the enumerations section.

Data Types

As discussed in the overview, there are seven data types (two of which are enumerations):

```
@  characters (a to z | A to Z)
#  numbers (0 to 9)
&  alphanumeric (characters | numbers)
$  symbols ~(characters | numbers)
%  wildcard (alphanumeric and symbols)
?  enumeration
*  nullable enumeration
```

Each of the data types has been described in more detail in their own section.

Number

This is any number between 0 and 9 inclusive. You can specify the range using, for example, the following expression: `#{25-32,34-43}` will pick a random number that lies between 25 and 32, and 34 and 43. You can only specify the length of the number to be generated using the square brackets. For more information on the length modifiers, see the Modifiers section below.

Character

This is any string between a-Z (irrespective of case). You can specify the length of the string to generate by using, for example, the following expression: `@[2] @[3]`. This will generate a string of five characters. For more information on the length modifiers, see the Modifiers section below.

Alphanumeric

This is any string which contains numbers and letters. You can specify the length of the string to generate by using, for example, the following expression: `&[2-4]`. This will generate a string between the length of two and four characters. For more information on the length modifiers, see the Modifiers section below.

Symbol

This is any string which contains a symbol (no numbers and no letters). You can specify the length of the string to generate by using, for example, the following expression: `$[1,4]`. This will generate a string of

either length one or four. For more information on the length modifiers, see the Modifiers section below.

WildCard

This is any string (alphanumeric + symbols). You can specify the length of the string to generate by using, for example, the following expression: `%[2,4,6-10]`. This will generate a string of either length two, four, six to ten. For more information on the length modifiers, see the Modifiers section below.

Enumerations

The purpose of enumerations is to randomly select a single element from a given list. There are two types of enumerations specified in the RDGL grammar: normal and nullable. The difference is relatively minor. Nullable enumerations, by default, also include the null (or more aptly, empty string) case. The normal enumerations are guaranteed to return one of the given elements. Enumerations must include one or more elements. The following example returns one of the four elements in the list:

```
?{true, false, 0, 1}
```

It is important to note that enumerations need not be homogenous in data-type. They can include any arbitrary set of **letters** or **numbers** as elements. Enumerations are most useful when attempting to randomize a statically known set of values (e.g. `?{June, April, August}`). Expressions are not evaluated inside of enumerations. It is also currently not possible enumeration over a comma as there is no escape character defined. This may change in later revisions of the language specification.

The most practical example of enumerations is for drop downs (e.g. credit card type):

```
*{ Visa, MasterCard, Discover, AmEx }
```

This example could potentially return five elements: the four elements defined and empty string.

Modifiers

The term modifier is used loosely here. The point is that each of the aforementioned data types can be modified by either length or, in the case of numbers, constrained to a range or set of numbers.

Length

To select the length of the string to generate, the square brackets [] are used. Some examples include:

```
$[3]              // length of 3
@[2,5]            // length of 2 or 5
&[3-6]            // length 3, 4, 5, or 6
%[1,2,10-12]      // length of 1, 2, 10, 11 or 12
```

It is possible to specify an inverse range such as `@[14-4]`, however, this will be correctly interpreted as 4-14 automatically.

Range

To select a predefined range of values, the curly braces {} are used. Some examples include:

```
#{2-10}           // pick a number between 2 and 10
#{5-6}            // pick a number between 5 and 6
```

It is possible to specify an inverse range such as `#{14-4}`, however, this will be correctly interpreted as 4-14 automatically. It is important to note that the range selection only works with numbers.

Loops

It is often necessary to repeat a specific expression or set of expressions (statements) multiple times. For this requirement, the concept of looping has been introduced into the RDGL grammar. The basic usage is as follows:

```
(N: statements)    // repeat statements N times
```

It is important to note that there is no way to prematurely break or terminate the loop. Therefore, if misused, could potentially create long running times (as is possible with any other programming language). To terminate execution, it is necessary to halt and kill the host process.

Tabs, Whitespace and Newlines

The RDGL grammar recognizes and consciously ignores all tabs, whitespace and newlines. This is contrary to the original language reference manual. However, during the implementation phase, it was decided that its inclusion was insignificant while offering the end-user some additional and much welcomed flexibility.

Project Plan

Process

As I was the only developer on the project, there was no need for a formalized project timeline and process. The lifecycle of RDGL was motivated primarily around test-driven development and language reference manual outlining the feature requirements.

The generally strategy around development was to ensure that I had a full end-to-end flow (from lexer all the way up to the command shell) working for the most basic of grammars as soon as possible. From there, I was able to start stubbing out tests (roughly 35 unit tests) and then slowly implementing additional features for the failing tests. This initial batch of unit tests also helped materialize progress. This motivation was extremely helpful when chasing down bugs deep inside the grammar/walker.

Programming Style

For the most part, the programming style used throughout the project adheres to the standard Java design guidelines and best practices. All methods have been commented in JavaDoc notation.

Tools

The project was developed using Eclipse (including a number of essential plugins) and JDK 1.5. The most important plugins included JUnit (for testing), Coverclipse (for code coverage results) and Subsclipse (for integrated SVN source control). The code was hosted (and versioned) through Google Code under the

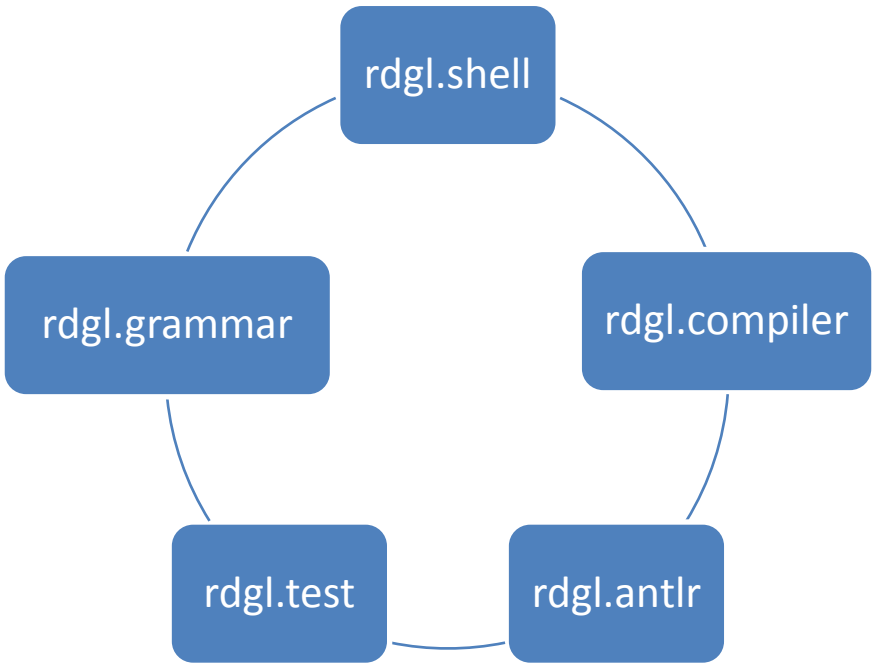
New BSD License. Last but certainly not least, ANTLR2 was used to generate the Lexer, Parser and TreeWalker classes.

Architectural Design

At a high level, RDGL is comprised of a Shell, an Engine and the ANTLR-generated Java code.

Components

The RDGL project is organized into five different packages. The rdgl.antlr package contains all the auto-generated code from the grammar and walker (.g) files.



rdgl.shell

This is the main entry-point into the project. It facilitates an interactive console used to interpreter RDGL expressions directly from a command-line interface. Furthermore, it offers a `parse(String)` method to enable tests or other assemblies to directly consume and execute RDGL expressions.

rdgl.compiler

The compiler package contains the data structures and Engine classes used to generate and store our random data. The data structures are manipulated directly from the RDGLWalker.

rdgl.antlr

The antlr package contains all of the code that is generated from the ANTLR grammar (.g) files. This code is not checked in to nor modified – it is merely linked to the Eclipse project under the rdgl.antlr package for compilation purposes.

rdgl.test

The test package contains all the unit tests (via JUnit) for the project. The tests are divided into different feature sets (NumberTests, LetterTests, AlphaNumericTests, EnumTests, LoopTests, etc). Furthermore, to ensure easy execution – they have been wrapped into a single test suite called AllTests.

This facilitates one-click execution of all the unit tests. The unit tests are run after each significant change or refactoring to ensure there have been no regressions. Tests are **always** run prior to any check-in. This ensures that most regressions are caught before they are committed to the official code base.

rdgl.grammar

The grammar package simply encapsulates the ANTLR grammar files (.g) which are used to generate the lexer, parser and walker Java code. The grammar files are at the heart of the project as is evident by the number of hours I spent debugging and tweaking the mere 200 lines of walker code!

Test Plan

The majority of unit tests (approximately 35) were developed before the project had really started. This allowed me to focus on a single (most basic) end-to-end scenario first. Once the bare infrastructure was in place, it was trivial (in hindsight, at least) to ensure unit tests started passing.

Regressions

The primary motivation behind the testing effort was to reduce the chance of regressions. It was originally anticipated that a software project such as a compiler requires a strong test suite to validate changes from build to build. In the end, the automation testing suite was a lifesaver and one of the few "lessons that fortunately didn't need to be learned".

Automation

Besides a series of basic smoke tests to ensure that the project was compile-able and not utterly broken – much emphasis was made to ensure that the tests were automated. This is primarily due to the fact that I wanted to analyze code coverage results to minimize dead code and increase overall code quality.

Code Coverage

This proved to be the most successful part of my project planning. The nature of a compiler is one which lends itself to well statement coverage. With an integrated tool inside my IDE, it was phenomenally interesting to see what paths had been exercised and which paths had gone untouched. I was able to craft unit tests for special cases as a way to mitigate regressions.

In more than one occasion, the unit tests in conjunction with the code coverage results helped uncover scenarios (bugs) that had not explicitly covered or tested before.

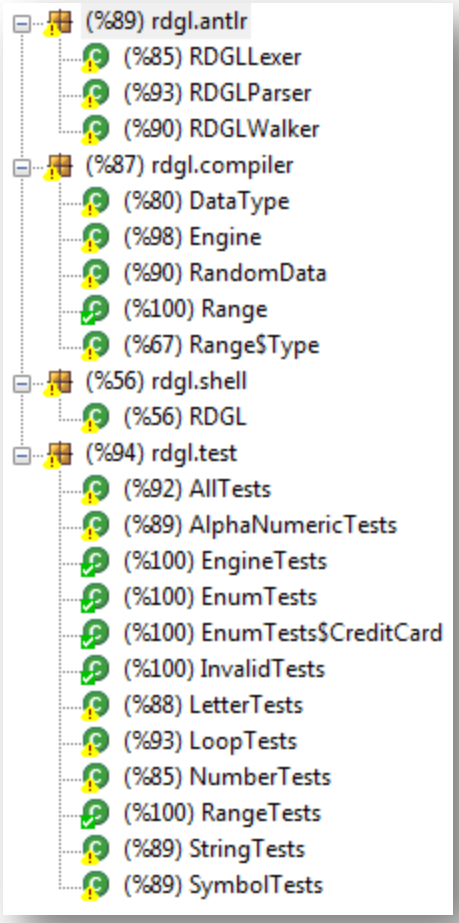


Figure 1: Screenshot of Code Coverage Summary Results

Lessons Learned

As with all date-driven projects, there are a number of key takeaways (or lessons learned) which we, as developers and/or planners, often take (and should take) from assignment to assignment. In the course of developing RDGL, I learned the following:

1. To develop a complete programming language of any serious complexity is quite a challenging undertaking. I've always known it's not an easy task – but there's a new found sort of respect for language designers and developers and even specification documents after I've been through the process.
2. It is difficult to write unit tests and validation routines for components which rely heavily on randomness as a key attribute. Each test becomes either overly complex or generally ineffective. However, I do not believe this is something I could have avoided in this circumstance given that my project is, after all, a data generation tool. Nevertheless, it is important to remember these pitfalls when planning and costing similar projects in the future.

3. Time. Yes, it was clearly highlighted at the beginning of the semester yet it seems like *if* only one had more time.

Overall, I am quite pleased with RDGL and although I have a long wish-list of things to improve or add in the coming months, I believe I have the basic foundation of a *real* and useful language. My top three pieces of advice for future teams:

1. Really, *really*, think through your design. Challenge any additions or "off-the-cusp" features.
2. KISS -- Keep It Simple Stupid.
3. Use some sort of unit test **and** code coverage framework.

Future Work Items

The following is a list of work items or features that I believe would further enhance RDGL that for one reason or another didn't make it into this initial release.

No Illegal Tokens

The parser is currently very stringent on the input data it accepts. Ideally, I would like to make it very forgiving (ignore incomprehensible characters and pass them through). Thereby, allowing users the flexibility to embed RDGL expression in virtually any text-based source application.

Configurable Distribution / Randomness

In order to maximize flexibility, the definition of "random" could be modified to a set of built-in and well-known distributions. The modifiers could potentially include but may not be limited to:

```
-uniform        // current default behavior
-normal
-poisson
-bernoulli
-chi
```

By allowing the probabilities to be generated using non-uniform distributions, we open the door for a number of other interesting uses in fields such as machine learning and data analysis.

Performance

Ranges are currently computed in a very primitive manner. There are many optimizations which could be made in future iterations.

Comments

The grammar currently does not accept any comments. This is contrary to the original language manual. However, it was consciously decided to abandon comments as I do not see their necessity given the short, simple and interactive aspect of RDGL.

More Options / Granular Control

It should be possible to exert more control over how the data is generated. For example, allowing both constraints and modifiers on non-number data types. This would enable users shorthanded notation for a number of scenarios whose expressions are more verbose than I would like.

Formatting

It is often times necessary to format generated data to a specific value. For example, it may be necessary to ensure that all numbers are zero padded to two integers (e.g. months). Therefore, one way to accomplish this requirement is with the following expression:

```
#{1-12:2}
```

This generates one of the following strings: 01, 02, 03, 04, 05, 06, 07, 08, 09, 10, 11, or 12. In the case where contradictory (or ambiguous) information is stated such as with the expression:

```
#{1-12:2}[4]
```

The padded values will be generated N times where in the above example N = 4. That is, the potential strings generated will be in the form of (for example): 01040312. You can only format (or pad) numbers.

Source

rdgl.grammar

RDGL.g

```
/*
 * Lexer and Parser
 * Random Data Generator Language (RDGL)
 * by Navid Azimi, Summer 2007
 */

header {package rdgl.antlr;}
class RDGLLexer extends Lexer;

options {
    k = 2;
    charVocabulary = '\3'..'\377';
    testLiterals = false;
    exportVocab = RDGL;
}

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

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

POUND       : '#';      // number
AT                : '@';      // letter
AMPERSAND   : '&';      // alphanumeric
```

```
DOLLAR            : '$';      // symbol
PERCENT           : '%';      // wildcard
ASTERISK    : '*';      // enum (nullable)
QUESTION    : '?';      // enum
L_BRACKET   : '[';
R_BRACKET   : ']';
L_PAREN     : '(';
R_PAREN     : ')';
L_SQUIGLY   : '{';
R_SQUIGLY   : '}';
COMMA       : ',';
COLON       : ':';
MINUS       : '-';

NUMBER            : (DIGIT)+;
WORD        : (LETTER)+;

WS          : (' ' | '\t')+ { $setType(Token.SKIP); };
NL          : ('\n' | ('\r' '\n') => '\r' '\n' | '\r')
                { $setType(Token.SKIP); newline(); }
            ;

/*****************************************************************************
*/

class RDGLParser extends Parser;

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

tokens {
      PROGRAM;
      STATEMENT;
      ENUMERATION;
      VALUES_LIST;
      RANGE;
      LENGTH;
      LOOP;
}

program
      : (loop | enumeration | expression)+
            {#program = #([PROGRAM, "PROGRAM"], program); }
      ;

numbers_range
      : NUMBER MINUS^ NUMBER
      ;

values_list
      : (NUMBER | numbers_range) (COMMA! (NUMBER | numbers_range))*
            {#values_list = #([VALUES_LIST, "VALUES_LIST"], values_list); }
      ;
```

```
length
      : L_BRACKET! values_list R_BRACKET!
            {#length = #([LENGTH, "LENGTH"], length); }
      ;

range
      : L_SQUIGLY! values_list R_SQUIGLY!
            {#range = #([RANGE, "RANGE"], range); }
      ;

expression
      : POUND^ (range | length)?
      | (AT^ | AMPERSAND^ | DOLLAR^ | PERCENT^ ) (length)?
      ;

values
      : (WORD | NUMBER) (COMMA! (WORD | NUMBER))*
      ;

enumeration
      : (QUESTION^ | ASTERISK^) L_SQUIGLY! values R_SQUIGLY!
            {#enumeration = #([ENUMERATION, "ENUMERATION"], enumeration); }
      ;

expr_block
      : (expression)+
            {#expr_block = #([STATEMENT, "STATEMENT"], expr_block); }
      ;

loop
      : L_PAREN! values_list COLON! expr_block R_PAREN!
            {#loop = #([LOOP, "LOOP"], loop); }
      ;
```

Walker.g

```
/*
 * Walker and TreeParser
 * Random Data Generator Language (RDGL)
 * by Navid Azimi, Summer 2007
 */

header {package rdgl.antlr;}
{
import java.util.*;
import rdgl.compiler.Engine;
import rdgl.compiler.Range;
import rdgl.compiler.Range.Type;
import rdgl.compiler.RandomData;
import rdgl.compiler.DataType;
}

class RDGLWalker extends TreeParser;

options {
    importVocab = RDGL;
```

```
}

{
	public void out(String line) {
		//System.out.println(line);
	}
}

program returns [ String result ]
{
	result = new String();
}
	: #(PROGRAM (prog:.
		{
			RandomData r = expr(#prog);
			result += r.getValue();
		})*)
	;

expr returns [ RandomData r ]
{
	r = new RandomData();
	Range i = null;
}
	: #(STATEMENT (smnt:.{ r.append(expr(#smnt).calculate()); })+)
	| #(LOOP i=modifier e:STATEMENT)
		{
			out("> LOOP");
			int value = Integer.parseInt(Engine.getRandomNumber(0,
i.getSize() - 1));
			out("iters = " + i.get(value));
			for(int j = 0; j < Integer.parseInt(i.get(value)); j++) {
				RandomData temp = expr(e);
				out("TEMP: " + temp.toString());
				r.append(temp.getValue());
				out("R: " + r.toString());
			}
			out("< LOOP");
		}
	| #(POUND { r.setType(DataType.Number); r.calculate(); } (num:. {
			out("> POUND");
			r.setRange(modifier(#num));
			r.calculate();
			out("< POUND");
		})*)
	| #(AT { r.setType(DataType.Letter); r.calculate(); } (letter:. {
			out("> AT");
			r.setRange(modifier(#letter));
			r.calculate();
			out("< AT");
		})*)
	| #(AMPERSAND { r.setType(DataType.AlphaNumeric); r.calculate(); }
(alphanumeric:. {
			out("> AMPERSAND");
			r.setRange(modifier(#alphanumeric));
			r.calculate();
			out("< AMPERSAND");
```

```
            })*)
      | #(DOLLAR { r.setType(DataType.Symbol); r.calculate(); } (symbol:. {
                  out("> DOLLAR");
                  r.setRange(modifier(#symbol));
                  r.calculate();
                  out("< DOLLAR");
            })*)
      | #(PERCENT { r.setType(DataType.Any); r.calculate(); } (wildcard:. {
                  out("> PERCENT");
                  r.setRange(modifier(#wildcard));
                  r.calculate();
                  out("< PERCENT");
            })*)
      | #(ENUMERATION { r.setType(DataType.Enum); r.calculate(); } (enu:. {
                  out("> ENUMERATION");
                  r.setRange(enumeration(#enu));
                  r.calculate();
                  out("< ENUMERATION");
            })*)
      ;

modifier returns [ Range range ]
{
      range = new Range();
}
      : #(RANGE (rng:. {
                  out("> RANGE");
                  range = modifier(#rng);
                  range.setType(Type.Range);
                  out("< RANGE");
            })*)
      | #(LENGTH (len:. {
                  out("> LENGTH");
                  range = modifier(#len);
                  range.setType(Type.Length);
                  out("< LENGTH");
            })*)
      | #(VALUES_LIST (vals:. {
            out("> VALUES_LIST");
            range.add(values_list(#vals));
            out("< VALUES_LIST");
            })*)
      ;

values_list returns [ List<String> values ]
{
      values = new ArrayList<String>();
      List<String> a, b;
}
      : num:NUMBER {
                  out("> NUMBER");
                  String number = num.getText();
                  out("Number: " + number);
                  values.add(number);
                  out("< NUMBER");
            }
      | val:WORD {
```

```
                  out("> WORD");
                  String value = val.getText();
                  out("Value: " + value);
                  values.add(value);
                  out("< WORD");
            }
      | #(MINUS a=values_list b=values_list) {
                  int start = Integer.parseInt(a.get(0));
                  int end = Integer.parseInt(b.get(0));

                  if(start > end) {
                        int temp = start;
                        start = end;
                        end = temp;
                  }

                  for(int i = start; i <= end; i++) {
                        values.add(i + "");
                  }
            }
      ;

enumeration returns [ Range range ]
{
      range = new Range();
}
      : #(QUESTION (ques:. {
                  out("> QUESTION");
                  range.add(values_list(#ques));
                  out("< QUESTION");
            })*)
      | #(ASTERISK (star:. {
                  out("> ASTERISK");
                  range.add(values_list(#star));
                  // create an empty item in the list for nullable
enumeration
                  List<String> list = new ArrayList<String>();
                  list.add("");
                  range.add(list);
                  out("< ASTERISK");
            })*)
      ;
```

rdgl.compiler

Engine.java

```
package rdgl.compiler;

import java.util.Random;

public abstract class Engine {

      private static Random rnd = new Random();
      private final static char[] letters = {
            'A', 'B', 'C', 'D', 'E', 'F', 'G', 'H', 'I', 'J', 'K', 'L', 'M',
            'N', 'O', 'P', 'Q', 'R', 'S', 'T', 'U', 'V', 'W', 'X', 'Y', 'Z',
```

```
		'a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j', 'k', 'l', 'm',
		'n', 'o', 'p', 'q', 'r', 's', 't', 'u', 'v', 'w', 'x', 'y', 'z',
};
private final static char[] symbols = {
		'~', '!', '@', '#', '$', '%', '^', '&', '*', '*', '(', ')', '-',
		'+', '`', '_', '+', '<', '>', '/', '?', ',', '.', ':', ';', '"',
		'{', '}', '\\', '|', '[', ']', '=', '*',
};

/**
 * Returns a single random number from [0,9] inclusive.
 * @return single random number as string
 */
public static String getRandomNumber() {
		return getRandomNumber(1);
}

/**
 * Returns a random number of the length specified.
 * @param length
 * @return random number as string
 */
public static String getRandomNumber(int length) {
		String value = new String();
		for(int i = 0; i < length; i++) {
			value += rnd.nextInt(10);
		}
		return value;
}

/**
 * Returns a random number from [min, max] inclusive.
 * @param min
 * @param max
 * @return random number as string
 */
public static String getRandomNumber(int min, int max) {
		int value = rnd.nextInt(max + 1 - min) + min;
		return value + "";
}

/**
 * Returns a random character (regardless of case).
 * @return single random character as string
 */
public static String getRandomCharacter() {
		return getRandomCharacter(1);
}

/**
 * Returns a random character of the length specified.
 * @param length
 * @return random character(s) as string
 */
public static String getRandomCharacter(int length) {
		String value = new String();
		for(int i = 0; i < length; i++) {
```

```
            value += letters[rnd.nextInt(letters.length)];
      }
      return value;
}

/**
 *
 * @return
 */
public static String getRandomAlphaNumeric() {
      return getRandomAlphaNumeric(1);
}

/**
 * Returns a random number of the length specified.
 * @param length
 * @return
 */
public static String getRandomAlphaNumeric(int length) {
      String value = new String();
      for(int i = 0; i < length; i++) {
            if(rnd.nextBoolean()) {
                  value += getRandomCharacter();

            } else {
                  value += getRandomNumber();
            }
      }
      return value;
}

/**
 * Returns a random symbol of length 1.
 * @return
 */
public static String getRandomSymbol() {
      return getRandomSymbol(1);
}

/**
 * Returns a random symbol of the length specified.
 * @param length
 * @return
 */
public static String getRandomSymbol(int length) {
      String value = new String();
      for(int i = 0; i < length; i++) {
            value += symbols[rnd.nextInt(symbols.length)];
      }
      return value;
}

/**
 * Returns a random string of the length 1.
 * @return
 */
public static String getRandomString() {
```

```
			return getRandomString(1);
		}
		/**
		 * Returns a random string of the length specified.
		 * @param length
		 * @return
		 */
		public static String getRandomString(int length) {
			String value = new String();
			for(int i = 0; i < length; i++) {
				int index = rnd.nextInt(3); // [0, 1, 2]
				switch(index) {
					case 0: value += getRandomNumber();
						break;
					case 1: value += getRandomCharacter();
						break;
					case 2: value += getRandomSymbol();
						break;
				}
			}
			return value;
		}
}
```

RandomData.java

```
package rdgl.compiler;

public class RandomData {

	private DataType type;
	private Range range;
	private String value;

	/**
	 * Default constructor.
	 *
	 */
	public RandomData() {
		this.range = new Range();
		this.value = new String();
	}

	/**
	 * Set the data type.
	 * @param type
	 */
	public void setType(DataType type) {
		this.type = type;
	}

	/**
	 * Set the range.
	 * @param range
	 */
	public void setRange(Range range) {
		this.range = range;
	}
```

```
	/**
	 * Appends the current string to value.
	 * @param val
	 */
	public void append(String val) {
		this.value += val;
	}

	/**
	 * Returns the current value of the expression.
	 * @return
	 */
	public String getValue() {
		return this.value;
	}

	/**
	 * Override the default implementation of toString() for this class.
	 */
	public String toString() {
		return "{type: " + this.type + ", value: " + this.value + "}";
	}

	/**
	 * Outputs the current value of the expression after it has been
	 * randomly generated without a newline.
	 *
	 */
	public void print() {
		System.out.print(this.value);
	}

	/**
	 * Outputs the current value of the expression after it has been
	 * randomly generated with a newline.
	 *
	 */
	public void println() {
		System.out.println(this.value);
	}

	/**
	 * Returns the randomly generated data.
	 *
	 */
	public String calculate() {
		if(this.range.getSize() == 0) {
			this.value = calculateLength(1);
			return this.value;
		}

		int index = Integer.parseInt(Engine.getRandomNumber(0,
this.range.getSize() - 1));
		String contents = this.range.get(index);

		if (this.type == DataType.Enum) {
```

```
                this.value = contents;
            } else if (this.range.getType() == Range.Type.Length) {
                this.value = calculateLength(Integer.parseInt(contents));
            } else if (this.range.getType() == Range.Type.Range) {
                this.value = contents;
            }
            return this.value;
        }

        /**
         *
         * @param length
         * @return
         */
        private String calculateLength(int length) {
            if(this.type == DataType.Number) {
                return Engine.getRandomNumber(length);
            } else if (this.type == DataType.Letter) {
                return Engine.getRandomCharacter(length);
            } else if (this.type == DataType.AlphaNumeric) {
                return Engine.getRandomAlphaNumeric(length);
            } else if (this.type == DataType.Symbol) {
                return Engine.getRandomSymbol(length);
            } else if (this.type == DataType.Any) {
                return Engine.getRandomString(length);
            } else {
                assert false;
                return null;
            }
        }
}
```

DataType.java

```
package rdgl.compiler;

public enum DataType {
        Number,                 // only numbers
        Letter,                 // only letters
        AlphaNumeric,       // numbers and letters
        Symbol,                 // no numbers or letters
        Any,                // symbols and alphanumeric
        Enum,               // enumeration of values
}
```

Range.java

```
package rdgl.compiler;



import java.util.ArrayList;

import java.util.List;
```

```
public class Range {
	private List<String> range;
	private Type type = Type.Length;

	public enum Type {
		Length,
		Range
	}

	public Range() {
		this.range = new ArrayList<String>();
	}

	public void add(List<String> values) {
		this.range.addAll(values);
	}

	public int getSize() {
		return this.range.size();
	}

	public String get(int index) {
		return this.range.get(index);
	}
```

```
	public Type getType() {
		return this.type;
	}

	public void setType(Type type) {
		this.type = type;
	}

	public String toString() {
		String value = new String();
		for (String v : range) {
			value += v + ',';
		}
		return value;
	}
}
```

rdgl.test

AllTests.java

```
package rdgl.test;

import junit.framework.Test;
import junit.framework.TestSuite;

public class AllTests {
```

```
	public static Test suite() {
		TestSuite suite = new TestSuite("All Tests");
		//$JUnit-BEGIN$
		suite.addTestSuite(NumberTests.class);
		suite.addTestSuite(LetterTests.class);
		suite.addTestSuite(AlphaNumericTests.class);
		suite.addTestSuite(SymbolTests.class);
		suite.addTestSuite(StringTests.class);
		suite.addTestSuite(EnumTests.class);
		suite.addTestSuite(LoopTests.class);
		suite.addTestSuite(RangeTests.class);
		suite.addTestSuite(EngineTests.class);
		suite.addTestSuite(InvalidTests.class);
		//$JUnit-END$
		return suite;
	}

}
```

AlphaNumericTests.java

```
package rdgl.test;

import junit.framework.TestCase;

import rdgl.shell.RDGL;

public class AlphaNumericTests extends TestCase {
```

```
protected void setUp() throws Exception {
        super.setUp();
}

protected void tearDown() throws Exception {
        super.tearDown();
}

public void testAlphaNumeric01() {
        String value = RDGL.parse("&");
        assertEquals(1, value.length());
}

public void testAlphaNumeric02() {
        String value = RDGL.parse("&[5]");
        assertEquals(5, value.length());
}

public void testAlphaNumeric03() {
        String value = RDGL.parse("&[0]");
        assertEquals(0, value.length());
}

public void testAlphaNumeric04() throws Exception {
        String value = RDGL.parse("&[1,10]");
```

```
		switch(value.length()) {
			case 1:
			case 10:
				break;
			default:
				fail();
		}
	}

	public void testAlphaNumeric05() throws Exception {
		String value = RDGL.parse("&[5-8]");
		if(value.length() < 5 || value.length() > 8) {
			fail();
		}
	}

	public void testAlphaNumeric06() throws Exception {
		String value = RDGL.parse("&[4-8, 6, 7, 8]");
		if(value.length() < 4 || value.length() > 8) {
			fail();
		}
	}

	public void testAlphaNumeric07() throws Exception {
		String value = RDGL.parse("&[1,3,5,7-10]");
```

```
                switch(value.length()) {
                        case 1:
                        case 3:
                        case 5:
                        case 7:
                        case 8:
                        case 9:
                        case 10:
                                break;
                        default:
                                fail();
                }
        }

        public void testAlphaNumeric08() {
                String value = RDGL.parse("&[1]");
                assertEquals(1, value.length());
        }
}
```

EngineTests.java

```
package rdgl.test;

import junit.framework.TestCase;
import rdgl.compiler.Engine;

public class EngineTests extends TestCase {
```

```
	protected void setUp() throws Exception {
		super.setUp();
	}

	protected void tearDown() throws Exception {
		super.tearDown();
	}

	public void testEngine01() {
		String value = Engine.getRandomAlphaNumeric();
		assertEquals(1, value.length());
	}

	public void testEngine02() {
		String value = Engine.getRandomString();
		assertEquals(1, value.length());
	}
}
```

EnumTests.java

```
package rdgl.test;

import junit.framework.TestCase;

import rdgl.shell.RDGL;

public class EnumTests extends TestCase {
```

```
private enum CreditCard {
	Visa,
	MasterCard,
	Discover,
	AmEx
};
private String creditCard = new String();

protected void setUp() throws Exception {
	super.setUp();
	for(CreditCard item : CreditCard.values()) {
		this.creditCard += item.toString() + ",";
	}
	this.creditCard = this.creditCard.substring(0, this.creditCard.length() - 1);
}

protected void tearDown() throws Exception {
	super.tearDown();
}

public void testEnum01() throws IllegalArgumentException {
	String value = RDGL.parse("?{" + this.creditCard + "}");
	CreditCard.valueOf(value);
}
```

```
	public void testEnum02() {
		RDGL.parse("?{1,a,2,b,3,c}");
	}

	public void testEnum03() {
		String value = new String();
		while(!value.equals("7")) {
			value = RDGL.parse("*{China,7,Jason,530}");
		}
	}
}
```

InvalidTests.java

```
package rdgl.test;

import junit.framework.TestCase;
import rdgl.shell.RDGL;

public class InvalidTests extends TestCase {

	protected void setUp() throws Exception {
		super.setUp();
	}

	protected void tearDown() throws Exception {
		super.tearDown();
```

```
}

public void testInvalid01() {
        RDGL.parse("a");
}

public void testInvalid02() {
        RDGL.parse("#a $[ %");
}

public void testInvalid03() {
        RDGL.parse("(4:");
}

public void testInvalid04() {
        RDGL.parse("(4:#");
}

public void testInvalid05() {
        RDGL.parse("*{");
}

public void testInvalid06() {
        RDGL.parse("?");
}
```

```
	public void testInvalid07() {
		RDGL.parse("%[-");
	}

	public void testInvalid08() {
		RDGL.parse("*{1,.");
	}

}
```

LetterTests.java

```
package rdgl.test;

import junit.framework.TestCase;
import rdgl.shell.RDGL;

public class LetterTests extends TestCase {

	protected void setUp() throws Exception {
		super.setUp();
	}

	protected void tearDown() throws Exception {
		super.tearDown();
	}
```

```
public void testLetter01() {
	String value = RDGL.parse("@");
	assertEquals(1, value.length());
}

public void testLetter02() {
	String value = RDGL.parse("@[5]");
	assertEquals(5, value.length());
}

public void testLetter03() {
	String value = RDGL.parse("@[0]");
	assertEquals(0, value.length());
}

public void testLetter04() throws Exception {
	String value = RDGL.parse("@[1,10]");
	switch(value.length()) {
		case 1:
		case 10:
			break;
		default:
			fail();
	}
}
```

```
public void testLetter05() throws Exception {
	String value = RDGL.parse("@[5-8]");
	if(value.length() < 5 || value.length() > 8) {
		fail();
	}
}

public void testLetter06() throws Exception {
	String value = RDGL.parse("@[4-8, 6, 7, 8]");
	if(value.length() < 4 || value.length() > 8) {
		fail();
	}
}

public void testLetter07() throws Exception {
	String value = RDGL.parse("@[1,3,5,7-10]");
	switch(value.length()) {
		case 1:
		case 3:
		case 5:
		case 7:
		case 8:
		case 9:
		case 10:
```

```
                        break;
                default:
                        fail();
            }
        }
}
```

LoopTests.java

```
package rdgl.test;


import junit.framework.TestCase;

import rdgl.shell.RDGL;


public class LoopTests extends TestCase {


    protected void setUp() throws Exception {
        super.setUp();
    }


    protected void tearDown() throws Exception {
        super.tearDown();
    }


    public void testLoop01() {
        String value = RDGL.parse("(1:@)");
        assertEquals(1, value.length());
    }
```

```
	public void testLoop02() {
		String value = RDGL.parse("(3:%[10])");
		assertEquals(30, value.length());
	}

	public void testLoop03() {
		String value = RDGL.parse("(1-4:#)");
		if(value.length() < 1 ||value.length() > 4) {
			fail();
		}
	}
}
```

NumberTests.java

```
package rdgl.test;

import junit.framework.TestCase;
import rdgl.shell.RDGL;

public class NumberTests extends TestCase {

	protected void setUp() throws Exception {
		super.setUp();
	}

	protected void tearDown() throws Exception {
```

```
        super.tearDown();
}

public void testNumber01() {
        String value = RDGL.parse("#");
        assertEquals(1, value.length());
}

public void testNumber02() {
        String value = RDGL.parse("#[5]");
        assertEquals(5, value.length());
}

public void testNumber03() {
        String value = RDGL.parse("#[0]");
        assertEquals(0, value.length());
}

public void testNumber04() throws Exception {
        String value = RDGL.parse("#[1,10]");
        switch(value.length()) {
                case 1:
                case 10:
                        break;
                default:
```

```
                fail();
        }
}

public void testNumber05() throws Exception {
        String value = RDGL.parse("#[5-8]");
        if(value.length() < 5 || value.length() > 8) {
                fail();
        }
}

public void testNumber06() throws Exception {
        String value = RDGL.parse("#[4-8, 6, 7, 8]");
        if(value.length() < 4 || value.length() > 8) {
                fail();
        }
}

public void testNumber07() throws Exception {
        String value = RDGL.parse("#[1,3,5,7-10]");
        switch(value.length()) {
                case 1:
                case 3:
                case 5:
                case 7:
```

```
                case 8:
                case 9:
                case 10:
                        break;
                default:
                        fail();
                }
        }

        public void testNumber08() {
                String value = RDGL.parse("#{100-423}");
                int temp = Integer.parseInt(value);

                if(temp < 100 || temp > 423) {
                        fail();
                }
        }

        public void testNumber09() {
                String value = RDGL.parse(null);
                if(!value.equals("")) {
                        fail();
                }
        }
}
```

RangeTests.java

```
package rdgl.test;


import java.util.ArrayList;

import java.util.List;


import junit.framework.TestCase;

import rdgl.compiler.Range;


public class RangeTests extends TestCase {


	protected void setUp() throws Exception {

		super.setUp();

	}


	protected void tearDown() throws Exception {

		super.tearDown();

	}


	public void testRange01() {

		Range range = new Range();

		Range.Type type = Range.Type.Length;

		range.setType(type);

		assertEquals(type, range.getType());

		range.toString();

	}
```

```
	public void testRange02() {
		Range range = new Range();
		Range.Type type = Range.Type.Range;
		List<String> values = new ArrayList<String>();
		values.add("Hello");
		values.add("World");
		range.add(values);
		range.setType(type);
		assertEquals(type, range.getType());
		range.toString();
	}
}
```

StringTests.java

```
package rdgl.test;

import junit.framework.TestCase;
import rdgl.shell.RDGL;

public class StringTests extends TestCase {

	protected void setUp() throws Exception {
		super.setUp();
	}

	protected void tearDown() throws Exception {
```

```
        super.tearDown();
}

public void testString01() {
        String value = RDGL.parse("%");
        assertEquals(1, value.length());
}

public void testString02() {
        String value = RDGL.parse("%[5]");
        assertEquals(5, value.length());
}

public void testString03() {
        String value = RDGL.parse("%[0]");
        assertEquals(0, value.length());
}

public void testString04() throws Exception {
        String value = RDGL.parse("%[1,10]");
        switch(value.length()) {
                case 1:
                case 10:
                        break;
                default:
```

```
                fail();
        }
}

public void testString05() throws Exception {
        String value = RDGL.parse("%[5-8]");
        if(value.length() < 5 || value.length() > 8) {
                fail();
        }
}

public void testString06() throws Exception {
        String value = RDGL.parse("%[4-8, 6, 7, 8]");
        if(value.length() < 4 || value.length() > 8) {
                fail();
        }
}

public void testString07() throws Exception {
        String value = RDGL.parse("%[1,3,5,7-10]");
        switch(value.length()) {
                case 1:
                case 3:
                case 5:
                case 7:
```

```
                        case 8:
                        case 9:
                        case 10:
                                break;
                        default:
                                fail();
                }
        }

        public void testString08() {
                String value = RDGL.parse("%[1]");
                assertEquals(1, value.length());
        }

        public void testString09() {
                String value = RDGL.parse("@%[14]$[3]#{10-20}#&");
                assertEquals(22, value.length());
        }
}
```

SymbolTests.java

```
package rdgl.test;

import junit.framework.TestCase;
import rdgl.shell.RDGL;

public class SymbolTests extends TestCase {
```

```
protected void setUp() throws Exception {
        super.setUp();
}

protected void tearDown() throws Exception {
        super.tearDown();
}

public void testSymbol01() {
        String value = RDGL.parse("$");
        assertEquals(1, value.length());
}

public void testSymbol02() {
        String value = RDGL.parse("$[5]");
        assertEquals(5, value.length());
}

public void testSymbol03() {
        String value = RDGL.parse("$[0]");
        assertEquals(0, value.length());
}

public void testSymbol04() throws Exception {
```

```
		String value = RDGL.parse("$[1,10]");
		switch(value.length()) {
			case 1:
			case 10:
				break;
			default:
				fail();
		}
	}

	public void testSymbol05() throws Exception {
		String value = RDGL.parse("$[5-8]");
		if(value.length() < 5 || value.length() > 8) {
			fail();
		}
	}

	public void testSymbol06() throws Exception {
		String value = RDGL.parse("$[4-8, 6, 7, 8]");
		if(value.length() < 4 || value.length() > 8) {
			fail();
		}
	}

	public void testSymbol07() throws Exception {
```

```
		String value = RDGL.parse("$[1,3,5,7-10]");
		switch(value.length()) {
			case 1:
			case 3:
			case 5:
			case 7:
			case 8:
			case 9:
			case 10:
				break;
			default:
				fail();
		}
	}

	public void testSymbol08() {
		String value = RDGL.parse("$[15-5]");
		assertEquals(10, value.length());
	}
}
```

rdgl.antlr

These files are generated by ANTLR using the RDGL.g and Walker.g files and therefore been omitted as per requested.