RDGL Reference Manual

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

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 usual.

Lexical Conventions

RDGL is comprised of three token types: identifiers, constraints and modifiers. These tokens together make up expressions. Expressions dictate what strings should be generated. A collection of one or more expressions are called statements. More formally, an oversimplified and incomplete representation of the RDGL grammar has be presented here for demonstration purposes:

```
identifier := @ | # | $ | % | enumeration
constraint := { range }
modifier := [ range ]
enumeration := *{ range }
expr := identifier constraint? modifier?
statement := expr*
```

Although in the context of different programming languages, terminology such as *identifiers*, *modifiers* and maybe even *constraints* are different – the terms have be re-appropriated and redefined in the RDGL perspective for this language reference manual.

Primary Examples

The principle behind RDGL is to facilitate the most common data generation scenarios in an intuitive and compact syntax. Therefore, to understand the syntax, it is necessary to understand the common scenarios. Take for instance the following examples to generate:

1. A *single* number, character, symbol or any combination therein:

   ```
   @ or # or $ or & or %
   ```

2. A number between 00 and 99:

   ```
   ##
   ```

3. An alphanumeric *string* of length 6:

   ```
   &[6]
   ```

4. A *string* of symbols of length 4 or 7:

   ```
   $[4,7]
   ```

5. A *single* number between 43 and 68 including 14 but excluding 50-59:

   ```
   #{14, 43-68, !50-59}
   ```

6. A *string* of characters between A and F but not including D with a +,- or empty suffix:

   ```
   @{A-F, !D} {+, -, ?}
   ```

7. A random element from an *enumeration*:

   ```
   *{Basketball, Baseball, Football, Soccer, Billiards}
   ```

8. A *string* representing a zero-padded month value (e.g. 01, 02, 03, .., 12:

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

9. A *string* of numbers of length 4 to 9, not including 5:

   ```
   #[4-9, !5]
   ```

10. A *single* number greater than 12 (minimum).

    ```
    #{12-*}
    ```

The above examples help illustrate the two major requirements of a given expression: the *content* and the *length* of the string that is to be generated. Identifiers define the data type that is going to be generated. Each identifier can then be constrained and/or modified as appropriate.

Tokens

Identifiers

There are five data types where each is denoted by a single token: *character* (@), *number* (#), *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. An *enumeration* (*) identifier also exists, however, since the asterisk itself does not generate any data by itself – it has been omitted as a data type as defined above. The asterisk (*) is only used to maintain syntactical consistency throughout the language.

```
@  characters (a to z | A to Z)
#  numbers (0 to 9)
&  alphanumeric (characters | numbers)
$  symbols ~(characters | numbers)
%  wildcard (alphanumeric and symbols)
*  enumeration (used as *{..})
```

It is important to note that the idea behind single character identifiers is in line with the overall RDGL goal to create a concise syntax for the most common scenarios.

Constraints

If an identifier is used without any constraints, the range of potential values is assumed to be in *normal* range of that identifier type. For example, the character identifier (`@`) would include any character between a to Z. Similarly, the number identifier (`#`) would include any digit between 0 and 9. However, it is readily apparent that it is more often necessary to generate a specific subset or range of values.

To accomplish this, constraints have been developed to restrict or define the data set that is to be generated. Therefore, to randomly select an integer between four and eight, the RDGL syntax is simply:

```
#{4-8}
```

There are four major actions which can be leveraged inside a constraint (or modifier). They include:

Or

The most basic operation is to facilitate a series of values from which one must be picked. The expression `#{1,3,7}` returns either a one, three or seven. Although this is similar to the concept of enumerations, the non-asterisk identifier forces all values to be of the same identifier data-type.

Range

This operation is shorthand for specifying a range of numbers. If non-numeric values are used (e.g. for `@`, `$` or `%`), the ASCII values are interpreted and ranged. To pick a number between 37 and 54 can simply be written as `#{37-54}`. Moreover, to select a character between J and K, `@{J-K}`. Ranges can be used in negation. For more information, see the Not operation below.

Not

Excludes specific values or range of values. This is most useful when there's a gap (e.g. one to ten not including three and seven): `#{1-10,!3,!7}`. It can also be used as part of a range.

Null

The `?` allows the entire expression to become *optional*. That is, by providing `?` as a constraint element, it is possible that none of the values are selected. For more information, see Nulls under Keywords.

It is important to note that any combination of the above four operations can be used together. As such, this is a perfectly valid RDGL expression: `#{3,!7-9, ?, 34-43}`.

Modifiers

It is often times necessary to create strings of varying (or random) lengths. In the current construct of RDGL, it is possible to achieve this using the `LOOP` feature. Take the following expression:

```
LOOP #: #;
```

This expression generates a random number between zero and nine, a random number of times between zero and nine. Therefore, the minimum output is *empty string* while the maximum output would be *nine 9s* (i.e. 999999999). To avoid the above notation (and the excessive use of loops), it is possible to leverage modifiers to achieve the same result in a more succinct and intuitive syntax:

```
#[0-9]
```

This expression is broken up in two: the identifier (`#`) and the modifier (`[0-9]`). The number identifier generates a single value between zero and nine. The modifier specifies a length range between zero and nine. That is, the range will be first evaluated: a single value between zero and nine will be randomly picked. This value is then returned to the modifier which in turn ensures that the length of the string is that value. The range is simply a random and dynamic function of `#[n]` where n is any integer greater than or equal to zero. This specific example could have been rewritten as `#[#]`. However, it is generally more sensible to require strings of a single fixed length (i.e. `[4]`), a series of potential fixed lengths (i.e. `[3,7]`), or a range of potential lengths (i.e. `[3-5]`). Similarly, it is feasible to exclude specific lengths using the `!` operator (i.e. `[1-5,!3]`).

Strings

In addition to randomly generated strings of data, it is also possible to provide hard-coded values as pass-thru. That is, RDGL accepts arbitrary strings of arbitrary lengths that will end up as part of the final data generation. This is useful for providing **static** data in addition to the dynamically generated data.

Example

The following example shows the use of strings to generate an XML document with randomly generated data:

```
"<CustomerInformation>"
"  <FirstName>" @[1-15] "</FirstName>"
```

```
"  <LastName>" @[1-32] "</LastName>"
"</CustomerInformation>"
```

Limitations

There is no string manipulation (e.g. concatenation, splitting, etc) available as the part of RDGL. However, developers are free to use the output of RDGL and run it through their favorite implementation of Regular Expressions to achieve similar effects.

Enumerations

A special-type of constraint that accepts any strings (be it valid or invalid RDGL tokens). Enumerations offer developers the ability to select a single element randomly from a given list. Enumerations must include two or more elements. That is, an enumeration with a single element is a violation of the RDGL grammar. This has been done to simplify the parsing rules for the optional comma. Furthermore, an enumeration with a single element makes little logical sense as it could simply be rewritten as a single string entity. 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 string as elements. This is the primary difference between an enumeration and a comma separated list of values for characters, numbers or symbols.

Generally speaking, 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.

Keywords

Nulls

The `?` keyword provides RDGL developers the ability to arbitrarily add null (or empty string) to any generation pattern. This is useful when you have an enumeration or range of values for which you'd like to include *none* as an option. Take for example the ternary variable which includes true, false and null. In accordance with the RDGL language reference, this can be implemented using: `*{true, false, ?}`. The advantage here is that if the `?` operator is selected, the entire pattern generates an empty string and thereby making the entire expression effectively optional. The null (`?`) operator can be used in constraints, modifiers and enumerations.

Loops

It is often times necessary to repeat a specific expression or set of expressions (statements). For this requirement, the `LOOP` keyword was developed. It offers the ability to repeat for a constant (N = 3, N = 4) or random (N = `#`, N = `#{3,6}`) number of times. The basic usage is as follows:

```
LOOP N: statements;
```

It is important to note that there is no way to prematurely break or terminate the loop. If misused, this 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.

The `LOOP` keyword is case sensitive.

Comments

The language supports single line comments (e.g. //) but only as the first token on a newline. That is, once a line begins with // it will be ignored and the parser will move to the next line.

Tabs, Whitespace and Newlines

The syntax of RDGL is one that strives for conciseness. Consequently, although **tabs** are unrecognized as part of the language syntax (unless provided inside of a string), **whitespaces** are superfluous and merely ignored during parsing.

Newlines, however, are considered due to the single line commenting scheme (e.g. //).

Features

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.

Randomness

This feature is a stretch goal and will only be implemented if time permits and the project as-is proves to be easier to develop than originally anticipated.

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

- -uniform this is 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.

Normal Distribution Sample Scenario

Take for example the requirement of generating a student roster of 16 to 29 students, with a Student ID, First Name, Last Name and final grade (whose distribution is normal). The final compliable RDGL source code would be:

```
LOOP #{16-29}:
"ID: " #[9] "Name: " @[2-13] @[2-13]
"Grade: " *{A+, A, A-, B+, B, B-, C+, C, C-, F, ?} -normal
;
```

The above application assumes that:

1. Student IDs are always 9 digits with no restrictions.
2. First Name is any string between 2 and 13 characters.
3. Last Name is any string between 2 and 13 characters.
4. It is possible for a student to have no grade recorded.

User Scenario and Sample Usage

Assume you are automating a web application which requires some customer information. For most web-forms, you'll see a variation of the following required fields and their client side validation routines:

- First Name (must be 1 to 15 characters)
- Last Name (must be 1 to 32 characters)
- Address1 (must be 1 to 14 characters)
- Address2 (must be 0 to 20 characters)
- City (must be 1 to 36 characters)
- State (must be 2 characters, uppercase)
- Zip Code (must be 5 digits)
- Telephone (must be in the form of xxx-xxx-xxxx)
- Credit Card Type (must be one of the enumerated credit card types)
- Credit Card Expiration (must be in the form of mm/yyyy where mm and yyyy are valid)
- Credit Card Number (depends on card type)
- Credit Card Verification (must be 3 digits, but cannot include 000)

This is a rather complicated yet common scenario. Without the flexibility of RGDL, the supposed developer would need to either (a) hard-code the specific data into his/her unit tests or (b) create wrapper methods for all different types of fields in order to return random (but well-formed) data (i.e. getRandomTelephone() or getRandomCreditCardType(), etc). Neither scenario is ideal. By having a

specialized language around the generation of random strings, the complexities of data generation and the niceties of a dedicated syntax allow:

- First Name @[1-15]
- Last Name @[1-32]
- Address1 #[0-5] &[1-14]
- Address2 &[0-20]
- City @[1-36]
- State @[2]
- Zip Code #[5]
- Telephone #[3] #[3] #[4]
- Credit Card Type *{Visa, MasterCard, Discover, American Express}
- Credit Card Expiration #{1-12:2} "/" #{2007-2039}
- Credit Card Number #[16]
- Credit Card Verification #{!0}[3]

Sample Code

The following compliable sample code which fulfills the above requirements:

```
"First Name: " @[1-15]
"Last Name: " @[1-32]
"Address: " #[0-5] @[1-14] *{Rd, Ave, Blvd, St, Ct}
"City: " @[1-10]
"State: " @[2]
"Zip: " #{10000-99999}
"Phone: " #[3] "-" #[3] "-" #[4]
"Type: " *{Visa, MasterCard, Discover, American Express}
"Number: " #[16]
"Expiration: " @{1-12:2} "/" #{2007-2039}
"CCV: " #{!0}[3]
```

With the given grammar, it is also possible (and often times useful) to directly generate XML documents. To accomplish this, you can simply take advantage of the ".." syntax (and pass through strings):

```
"<CustomerInformation>"
"     <FirstName>" @[1-15] "</FirstName>"
"     <LastName>" @[1-32] "</LastName>"
"</CustomerInformation>"
```