# The Grapevine Grammar Format

I've heard it through the...

## 1. Grammar syntax

The Grapevine Grammar Format (GGF) is based on the Java Speech Grammar Format (JSGF) Version 1.0. It has proprietary extension for semantic code execution, or "execution tags".

Context-free grammars have been used as part of natural language understanding and dialog systems, for example to format numbers or dates ("nine hundred seventy five" -> "975"). A grammar consists of two types of tokens: terminals, i.e. tokens that cannot expand into anything else, and non-terminals, essentially pointers to other nodes in the grammar. A non-terminal is denoted by the <> brackets, e.g. <start>. Whitespaces are not permitted in non-terminals. Terminals are either spelled out, or can be grouped by parentheses ("), e.g.: *do something* or "*do something*".

The structure of a grammar is defined by left-hand side (LHS) and right-hand side (RHS) tokens, for example:

```
<start> = do something;
```

Here, the non-terminal <start> is the LHS, the terminals *do something* define the RHS, forming a rule. Each rule has to be terminated by a semicolon (;). A rule can have multiple branches on the RHS, for example:

```
<start> = do something | let it be;
```

This rule has two legal input phrases, namely *do something* and *let it be*. Non-terminal can also reside on the RHS, as long as these terminals are also defined on the LHS of another rule. For example:

```
<start> = <do> | let it be;
```

is an illegal expression until <do> is defined as well:

```
<start> = <do> | let it be;
<do> = do something;
```

In the GGF, rules are private by default. This means that all entry points have to be declared as public, e.g. here:

```
public <start> = <do><something> | let it be;
<do> = do;
<something> = something;
```

the phrases *do* and *something* are not legal, whereas *do something* is.

Both terminals and non-terminals can be optional, which is indicated by square brackets, e.g. here:

```
public <start> = <do><something> | let [it] be;
<do> = do;
<something> = something;
```

Both *let it be* and *let be* are legal expressions. Terminals can also consist of wildcards, which are defined by the % symbol. For example, in this rule:

```
public <start> = do % %;
```

Any three-word phrase that starts with *do* is legal.

## 2. Include files

To share functionality, GGF grammars allow for including other grammars. Grammars are therefore required to start with a tag:

```
#JSGF V1.0 ISO8859-1 en;
```

and a unique identifier tag, e.g.:

```
grammar com.test.do;
```

It is good practice to ensure that the last part of the grammar identifier matches the file name of the grammar. The command:

```
import <com.test.do.*>
```

will now import all public rules of com.test.do into the grammar.

## 3. Action tags

When parsing a phrase against a grammar, the parser returns whether the phrase is valid, but nothing else. Action tags are single or multiple instructions that are executed according to the semantics of the phrase. Each RHS terminal and non-terminal can contain action one tag, in which all instructions are grouped together by a pair of curly brackets ({}). Depending on the terminals and non-terminals visited by the phrase, action tags are ordered and executed in that order. Multiple instructions need to be terminated by a semicolon (;), and are executed in the order in which they are given within a single tag.

Variables are automatically declared whenever they are used first. Variables consist of a domain and an identifier, e.g.

```
something.important
```

Here, *something* is the domain, and *important* the identifier. Variables with the domain *this* are public and accessible with the grammar result. Private domains are important for imported grammars as to avoid name clashes with the importing grammar. Variables can be numeric or strings. As output, all numerals are converted to strings. Once a variable is unambiguously a string, it cannot be converted back to a numeric value.

For example, the rule

```
<start> = do {this.string = “success”; this.number = 55;};
```

assigns a string value to the variable *this.string* and a numeric value to this.number. String values have to be given in quotes and may contain whitespace characters.

The GGF supports the following basic operations:

+ addition (numeric) or concatenation (string)
- subtraction (numeric) or substring removal (string)
* multiplication (numeric) or runtime error (string)
/ division (numeric) or runtime error (string)

For example:

```
<start> = do {this.out = “success”; this.out = this.out + “ is sweet”;};
```

Will output the phrase *success is sweet*. This rule:

```
<start> = do {this.num = 10; this.num = this.num * 5;};
```

will return 50 in the variable *this.num*.

Variables can also be assigned the literal of its preceding terminal, e.g.:

```
<start> = do {this.out = *};
```

returns *do* in *this.out*. Phrases can be assigned when grouped by parentheses.

## 4. Standard variables interfacing with the workflow

Grapevine reserves certain variables to query the grammars. The variables are:

*this.command*: a correctly formatted command (bash, pyhton, etc.). Note that the output can be multi-line by inserting *n* to indicate line breaks in the text.

*this.package*: the name or identifier of the package the commands depend on. This variable is queried when the grammar finds a legal phrase, i.e. unused grammars will not contribute to the list of packages that Grapevine lists as required.

Grapevine can query any other public variable by specifying

```
> @variable
```

at the end of the line in the Grapevine script. The variable name has to match the grammar variable minus the *this.* Part, e.g.:

```
run program > @outfile
```

queries the grammar variable *this.outfile*.

## 5. Execution order

The execution order follows the order in which the terminals or non-terminals are visited. For example:

```
<start> = align {this.command = "Satsuma "} <genome> {this.command = this.command + "
-t " + this.genome} to <genome> {this.command = this.command + " -q " + this.genome} on
<number> {this.command = this.command + " -n " + this.number} cores;

<genome> = % {this.genome = $};
<number> = % {this.number = $};
```

Here, since only one command is legal, the execution order is:

```
this.command = "Satsuma "
this.command = this.command + " -t " + this.genome
this.command = this.command + " -q " + this.genome
this.command = this.command + " -n " + this.number
```

If written more flexibly:

```
<start> = <option><option><option><option>;
<option> = align {this.command = this.command + "Satsuma "}
           | <genome> {this.command = this.command + " -t " + this.genome}
           | to <genome> {this.command = this.command + " -q " + this.genome}
           | on <number> {this.command = this.command + " -n " + this.number} cores
           ;
```

then different phrases are legal that result in different output:

*align on 5 cores mouse.fasta to human.fasta* → Satsuma -n 5 -t mouse.fasta -q human.fasta
*mouse.fasta to human.fasta on 5 cores align* → -t mouse.fasta -q human.fasta -n5Satsuma

where the latter is not a valid command. The use of more variables can keep flexibility, while ensuring proper command formation:

```
<start> = <option><option><option><option> {this.command = this.program + this.t + this.q
+ this.cores};
```

```
<option> = align {this.program = “Satsuma ”}
         | <genome> {this.t = “ -t ” + this.genome}
         | to <genome> {this.q = “ -q ” + this.genome}
         | on <number> {this.cores = “ -n ” + this.number} cores
         ;
```

Here, the output command does not depend on the order of the input phrase.