Grapevine User Manual v0.0.2

1. Overview

On a high level, Grapevine is an abstraction layer that converts any syntax into any other syntax. The translation from *source* to *target* is defined by *grammars*. A grammar is a set of rules that determine valid input, together with instructions as to how to map it to valid output.

In context of bioinformatics, Grapevine allows for easily implementing a workflow or pipeline that contains multiple processing steps. The input data is specified in a table that describes file locations and other attributes, which allows for processing data in batches.

Conceptually, Grapevine is designed to separate functionality such that components can be replaced without affecting the overall workflow. For example, the choice of a read aligner is defined by the grammar, so simple replacing or changing the grammar can switch from one program to another, given that the input and output files adhere to the same format (fastq in, sam or bam out).

For processing input with metadata, Grapevine takes a specified labeled table, from which variables can be accessed in the script. The number of output scripts is the number of data rows in this table. Together with a header file, the workflow can be configured so that the script does not change, while the header and table provide all information for the current run.

2. Grapevine projects

A Grapevine project typically consist of four components:

1. The main script
2. The data description table
3. The external header (optional)
4. A set of pre-defined or custom grammars

2.1 The main script

The main script first contains instructions for setting up the environment. If the target language is e.g. bash, and assuming SLURM as the scheduler, the first lines of the script might look like this:

```
#!/bin/bash -l

#SBATCH -A myProject
#SBATCH -p core
#SBATCH -n 16
#SBATCH -t 6:00:00

module load bioinfo-tools
export PATH=/home/manfredg/Software/minute-chip/bin/:$PATH
```

Next, the script defines what grammars to load, following this sytax:

```
>grammar = workflow/grammars/demux.gram
>grammar = workflow/grammars/mapping.gram
```

Variables are defined as follows:

```
@barcode = TGTCCAAT
```

Commands are then given in order following the syntax that is define by the grammar:

```
demultiplex folder first_run file base ONE barcode @barcode sample sample1
```

Where return variables can be collected like this:

```
demultiplex folder first_run file base ONE barcode @barcode sample sample1 > @out1 @out2
```

In this example, running Grapevine will result in the following script:

```
#!/bin/bash -l

#SBATCH -A myProject
#SBATCH -p core
#SBATCH -n 16
#SBATCH -t 6:00:00

module load bioinfo-tools
export PATH=/home/manfredg/Software/minute-chip/bin/:$PATH


>grammar = workflow/grammars/demux.gram
>grammar = workflow/grammars/mapping.gram


@barcode = TGTCCAAT


DeMuxFastq  -i1 ONE_R1.fastq.gz -i2 ONE_R2.fastq.gz -o1 first_run/ONE_sample1_R1.demux.fastq -o2
first_run/ONE_sample1_R2.demux.fastq -b TGTCCAAT -mis 1 -off 6
```

Where the variables @out1 and @out2 are set to “first_run/ONE_sample1_R1.demux.fastq” and “first_run/ONE_sample1_R2.demux.fastq” respectively, storing the locations of the output files that can then be passed on to subsequent steps.

Note that the script can contain any instructions in the target language, which will no be modified. This allows for rapid prototyping and debugging.

2.2 The data description table

To process multiple samples, a table can be specified in the main script or header:

```
@table = sampledata/table.txt
```

The table is blankspace (blank or tab) delimited with column headers in the first row, followed by entries for each sample or data set, e.g.:

```
analysis  sample       folder          filebase  barcode     taglen   description
H33       H33TAG_ctrl  20170828AA_MC   H3        CTACCAGG    6        Control
H33       H33TAG_P1    20170828AA_MC   H3        CATGCTTA    6        Experiment
H33       H33TAG_P2    20170828AA_MC   H3        GCACATCT    8        Experiment
H33       H33TAG_P3    20170828AA_MC   H3        AGCAATTC    7        Experiment
```

In the script or header, the entries can be directly accessed via @table. and the header names:

```
demultiplex folder @table.folder file base @table.filebase barcode @table.barcode sample
@table.sample
```

For each row in the table, Grapevine generates an executable script by filling in the value tables.

2.3 The external header

The header file will be included in the script, follows the same syntax, and allows for specifying a particular data set or options for processing the data, e.g. the variables:

```
@table = sampledata/table15.txt
@ref = /data1/common/references/mm10/mm10
```

might preferably be given in the header to specify a particular run that is processed against a particular database or genome reference.

2.4 A set of pre-defined or custom grammars

Grapevine uses the JSGF grammar format with particular extensions. It supports recursion as well as wild cards. The grammar header should specify the version and name:

```
#JSGF V1.0 ISO8859-1 en;
grammar com.test.dedup;
```

Grammars can import other grammars and access any public rule, e.g.:

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

Note that the depth of inclusion is currently limited to 1.

The grammar is a set of left hand rules (LHR) and right hand rules (RHR), specified as:

```
public <top> = process <dedup> | ignore;
```

terminated by a semicolon, and using the vertical bar to specify an OR relation. The 'public' keyword specifies that this rule may be accessed from other grammars. Private rules omit the 'public' keyword. There is no limitation on how deeply nested rules can be.

In the execution tags, delimited by curly brackets, variables are processed:

```
<file1> = in1 {this.in1 = "input1"}
```

where values are either specified explicitly, or via '*', which returns the input:

```
<file1> = in1 {this.R1 = *}
```

Multiple statements within an execution tag are separated by semicolons. Wildcards are specified via the '%' symbol, e.g.:

```
<file2> = in2 <wildcard> {this.R2 = this.wc; this.param = "in2"};

<wildcard> = % {this.wc = $};
```

Note that Grapevine will supply a set of grammars for common bioinformtics tools, with different sets of input syntax and supporting different target languages, such as bash and python.

3. Syntax

The global delimiter in a Grapevine scripts is whitespace (blank or tab). Each line is parsed separately. The # symbol is used for comments, which extend to the end of each line.

3.1 Statics

A Grapevine script may contain statements intended for the target platform, e.g. bash commands. These are routed through directly to the processed script.

Grapevine keeps a dictionary of all words in loaded grammars and parses each line. If more than half of the words in a line are dictionary words, but the line does not parse against any grammar, a warning is issued, together with a comment in the processed script:

```
##>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
## WARNING!!! This line looks like a command, but wasn't understood:
```

3.2 Keywords

Built-in script keywords are:

```
>grammar
>collapse
>loop
<loop
@table
@index
@
#
```

3.3 Variables

Variables are indicated by a leading @ character. Variable assignments follow the syntax:

```
@variable = value
```

where ‘value’ is a character string. Note that blanks are not permitted in variable values. For additional variable manipulation, including numerical operations, grammars with a freely defined syntax can be used. Variables can only be used in context of a grammar and are not accessible to route-through commands.

3.4 The table variable

The @table variable is reserved for reading a table from a blank- or tab-delimited text file:

```
@table = table.txt
```

The content of the table is accessed via the table header in the first row, e.g.

```
@table.sample
```

whereas the file name of the table is returned when using

```
@table
```

without any header.

A table is typically used to provide sample the descriptions, e.g. the location of input files, attributes, etc. Grapevine assumes that the number of rows in the table (minus the header) equals the number of scripts to be exectuted in parallel. Rows can be collapsed using the

```
>collapse table <col>
```

command, to allow for processing multiple samples at once. Note that collapsed tables are valid throughout a single script, and that tables cannot be uncollapsed. For example, after collapsing the table:

```
sample experiment
s1     control
s2     control
s3     case1
s4     case1
```

via

```
>collapse table experiment
```

two scripts are being generated, one in which @table.sample returns “s1 s2” and @table.control returns “control”, the other in which @table.sample returns “s3 s4” and @table.control returns “case”.

Note that table entries can be manipulated using grammar operations, e.g. for attaching prefixes or extensions to file names.

Tables are editable via grammar commands (tableops.gram), which allows for storing result information in the table, as well as passing information from one script on to another.

Add a column (noop if column exists):

```
table @table add column <col>
```

Remove a column (noop if column doesn’t exist):

```
table @table remove column <col>
```

Set table content (row is a zero-based integer, typically provided by the @index variable):

```
table @table set column <col> row <row> to <value>
```

Set table content from a file containing a single entry:

```
table @table add to column <col> from file <file> row <row>
```

Set table content from file containing multiple entries:

```
table @table fill column <col> from file <file> row <row>
```

3.5 Loops

Grapevine uses the following syntax to define loops:

```
>loop <N>
commands…
<loop
```

Unwraps the command N times. Alternatively,

```
>loop variable = <var1> <var2> <var3>
commands….
<loop
```

executes the commands as often as values are assigned to the variable.

4. Execution

A Grapevine workflow is executed by:

```
RunWorkflow
```

Available arguments:

```
-i<string> : list of scripts
-head<string> : data header file (def=)
-dryrun<bool> : do not submit to the job queue (def=0)
-m<string> : e-mail address for notifications (def=)
-d<string> : scripts directory (def=grapevine_scripts)
-s<string> : queueing system (SLURM, local) (def=SLURM)
```

A script is provided using the –i option. Multiple scripts can be given when separated by commas (no blanks), e.g.:

```
RunWorkflow –i first.grapevine,second.grapevine
```

Scripts are processed in the order in which they are listed. Note that the scripts are parsed immediately before execution, i.e. data can be passed from script to script using the table edit functions. However, when using the –dryrun option, which does not run any executables, missing table entries may produce errors.

Currently, the only job submission system supported is SLURM. RunWorkflow runs on the login node and submits as many jobs as are specified in the table, and waits for the execution to be finished before continuing to the next script. The –m option allows for e-mail notifications after each script is complete.