Anatomy of a Snakemake rule

Basic structure of a rule

```
### Contents of Snakefile in cwd ##
rule:
    output: "results/sample1.stats.txt"
    shell:
        """
        echo -e "sample1\t50%" > {output}
        """
```

```
$ snakemake -j 1 results/sample1.stats.txt
Building DAG of jobs...
Using shell: /bin/bash
Provided cores: 1 (use --cores to define parallelism)
Rules claiming more threads will be scaled down.
Job counts:
        count   jobs
        1       1
        1
Select jobs to execute...

[Tue Sep 28 11:50:45 2021]
rule 1:
    output: results/sample1.stats.txt
    jobid: 0

[Tue Sep 28 11:50:45 2021]
Finished job 0.
1 of 1 steps (100%) done

$ cat results/sample1.stats.txt
sample1 50%
```

More commonly, rules are named and have both input and output:

```
rule generate_stats:
    input: "results/sample1.bam"
    output: "results/sample1.stats.txt"
    shell:
        """
        samtools flagstat {input} > {output}
        """
```

Wildcards generalize a workflow:

```
rule generate_stats:
    input: "results/{sample}.bam"
    output: "results/{sample}.stats.txt"
    shell:
        """
        samtools flagstat {input} > {output}
        """
```

Input can also come directly from functions:

```
### Contents of config file ###
samples:
  sample1: "results/genomeMap/sample1.bam"
  sample2: "results/transcriptomeMap/sample2.bam"
```

```
def get_bamfile(wildcards):
    return config["samples"][wildcards.sample]

rule generate_stats:
    input: get_bamfile
    output: "results/{sample}.stats.txt"
    shell:
        """
        samtools flagstat {input} > {output}
        """
```

This can also be written in the form of lambda expressions, e.g.:

```
    input: lambda wildcards: config["samples"][wildcards.sample]
```

SciLifeLab

NBIS

Ambiguities can arise when two rules produce the same output:

```
rule generate_stats:
    input: "results/{sample}.bam"
    output: "results/{sample}.stats.txt"
    shell:
        """
        samtools flagstat {input} > {output}
        """

rule print_stats:
    input: "results/{sample}.log"
    output: "results/{sample}.stats.txt"
    shell:
        """
        grep "% alignment" {input} > {output}
        """

rule make_report:
    input: "results/{sample}.stats.txt"
    output: "results/{sample}.report.pdf"
```

This can be handled either via the `ruleorder` directive:

```
ruleorder: generate_stats > print_stats
```

Or by specifically referring to the output of a certain rule:

```
rule make_report:
    input: rules.generate_stats.output
    output: "results/{sample}.report.pdf"
```

Messages and logfiles add descriptions and help with debugging:

```
rule generate_stats:
    input: "results/{sample}.bam"
    output: "results/{sample}.stats.txt"
    log: "results/{sample}.log"
    message: "Generating stats for sample {wildcards.sample}"
    shell:
        """
        samtools flagstat {input} > {output} 2>{log}
        """
```

Compute resources can be set with e.g. threads and resources:

```
rule generate_stats:
    input: "results/{sample}.bam"
    output: "results/{sample}.stats.txt"
    log: "results/{sample}.log"
    message: "Generating stats for sample {wildcards.sample}"
    threads: 4
    resources:
        mem_mb=100
    shell:
        """
        samtools flagstat --threads {threads} {input} > {output} 2>{log}
        """
```

Rule parameters can be set with the params directive:

```
rule generate_stats:
    input: "results/{sample}.bam"
    output: "results/{sample}.stats.txt"
    log: "results/{sample}.log"
    message: "Generating stats for sample {wildcards.sample}"
    threads: 4
    resources:
        mem_mb=100
    params:
        verbosity = 2
    shell:
        """
        samtools flagstat --verbosity {params.verbosity} \
          --threads {threads} {input} > {output} 2>{log}
        """
```

Software environments can be set for each rule using `conda:` :

```
rule generate_stats:
    input: "results/{sample}.bam"
    output: "results/{sample}.stats.txt"
    log: "results/{sample}.log"
    message: "Generating stats for sample {wildcards.sample}"
    threads: 4
    resources:
        mem_mb=100
    params:
        verbosity = 2
    conda: "envs/samtools.yml"
    shell:
        """
        samtools flagstat --verbosity {params.verbosity} \
          --threads {threads} {input} > {output} 2>{log}
        """
```

```
### Contents of envs/samtools.yml ###
name: samtools
channels:
  - bioconda
dependencies:
  - samtools
```

Or by using `envmodules`, e.g. in compute clusters:

```
rule generate_stats:
    input: "results/{sample}.bam"
    output: "results/{sample}.stats.txt"
    log: "results/{sample}.log"
    message: "Generating stats for sample {wildcards.sample}"
    threads: 4
    resources:
        mem_mb=100
    params:
        verbosity = 2
    conda: "envs/samtools.yml"
    envmodules:
        "bioinfo-tools",
        "samtools"
    shell:
        """
        samtools flagstat --verbosity {params.verbosity} \
          --threads {threads} {input} > {output} 2>{log}
        """
```

If no targets are given on the command line, Snakemake will run the first rule specified. By convention this rule is named `all` and is used as a 'pseudo-rule' to define what the workflow will generate.

```
samples = ["sample1","sample2"]

rule all:
    input: expand("results/{sample}.stats.txt", sample = samples)

rule generate_stats:
    input: "results/{sample}.bam"
    output: "results/{sample}.stats.txt"
    log: "results/{sample}.log"
    message: "Generating stats for sample {wildcards.sample}"
    threads: 4
    resources:
        mem_mb=100
    params:
        verbosity = 2
    conda: "envs/samtools.yml"
    envmodules:
        "bioinfo-tools",
        "samtools"
    shell:
        """
        samtools flagstat --verbosity {params.verbosity} \
          --threads {threads} {input} > {output} 2>{log}
        """
```

What else?

Snakemake is constantly being updated with new features. Check out the documentation (https://snakemake.readthedocs.io/), and specifically the section about writing rules.

Questions?