ngless Documentation

Release 0.0

NGLess Authors

Jul 22, 2017

Contents

1 Example **3**

2 Basic features **5**

3 Traditional Unix command line usage **7**
- 3.1 Converting a SAM file to a FASTQ file . . . 7
- 3.2 Getting aligned reads from a SAM file as FASTQ file . . . 7
- 3.3 Reading from STDIN . . . 8

4 Building and installing **9**

5 Authors **11**
- 5.1 Introduction . . . 11
- 5.2 Installation . . . 12
- 5.3 Ocean Metagenomics Functional Profiling . . . 13
- 5.4 Human Gut Metagenomics Functional & Taxonomic Profiling . . . 15
- 5.5 Tutorial . . . 19
- 5.6 NGLess one liners . . . 21
- 5.7 Preprocessing FastQ Data . . . 22
- 5.8 Functions . . . 24
- 5.9 Methods . . . 33
- 5.10 Standard library . . . 34
- 5.11 Modules . . . 36
- 5.12 Constants . . . 40
- 5.13 Available Reference Genomes . . . 40
- 5.14 Taxonomic profiling using mOTUs with ngless . . . 42
- 5.15 Configuration . . . 42
- 5.16 Frequently Asked Questions . . . 44
- 5.17 Advanced options . . . 46
- 5.18 Language . . . 46

NGLess is a domain-specific language for NGS (next-generation sequencing data) processing.

Note: This is *pre-release* software, currently available as a preview only. Please get in touch if you want to use it in your work.For questions, you can also use the ngless mailing list.

NGLess is best illustrated by an example:

CHAPTER 1

Example

```
ngless "0.0"
input = fastq(['ctrl1.fq','ctrl2.fq','stim1.fq','stim2.fq'])
preprocess(input) using |read|:
    read = read[5:]
    read = substrim(read, min_quality=26)
    if len(read) < 31:
        discard

mapped = map(input,
                reference='hg19')
write(count(mapped, features=['gene']),
        ofile='gene_counts.csv',
        format={csv})
```

CHAPTER 2

Basic features

- preprocessing and quality control of FastQ files
- mapping to a reference genome (implemented through bwa)
- annotation and summarization of the alignments using reference gene annotations

Ngless has builtin support for some model organisms:

1. Homo sapiens (hg19)
2. Mus Muscullus (mm10)
3. Rattus norvegicus (rn4)
4. Bos taurus (bosTau4)
5. Canis familiaris (canFam2)
6. Drosophila melanogaster (dm3)
7. Caenorhabditis elegans (ce10)
8. Saccharomyces cerevisiae (sacCer3)

and the standard library includes support for mOTUs.

CHAPTER 3

Traditional Unix command line usage

`ngless` can be used as a traditional command line transformer utility, using the `-e` argument to pass an inline script on the command line.

The `-p` (or `--print-last`) argument tells ngless to output the value of the last expression to `stdout`.

Converting a SAM file to a FASTQ file

Extract file reads from a SAM (or BAM) file:

```
$ ngless -pe 'as_reads(samfile("file.sam"))' > file.fq
```

This is equivalent to the full script:

```
ngless "0.0" # <- version declaration, optional on the command line
samcontents = samfile("file.sam") # <- load a SAM/BAM file
reads = as_reads(samcontents) # <- just get the reads (w quality scores)
write(reads, ofname=STDOUT) # <- write them to STDOUT (default format: FASTQ)
```

This only works if the data in the samfile is single ended as we pipe out a single FQ file. Otherwise, you can always do:

```
ngless "0.0"
write(as_read(samfile("file.sam")),
        ofile="output.fq")
```

which will write 3 files: `output.1.fq`, `output.2.fq`, and `output.singles.fq` (the first two for the paired-end reads and the last one for reads without a mate).

Getting aligned reads from a SAM file as FASTQ file

Building on the previous example. We can add a `select()` call to only output unmapped reads:

```
$ ngless -pe 'as_reads(select(samfile("file.sam"), keep_if=[{mapped}]))' > file.fq
```

This is equivalent to the full script:

```
ngless "0.0" # <- version declaration, optional on the command line
samcontents = samfile("file.sam") # <- load a SAM/BAM file
samcontents = select(samcontents, keep_if=[{mapped}]) # <- select only *mapped* reads
reads = as_reads(samcontents) # <- just get the reads (w quality scores)
write(reads, ofname=STDOUT) # <- write them to STDOUT (default format: FASTQ)
```

Reading from STDIN

For a true Unix-like utility, the input should be read from standard input. This can be achieved with the special file `STDIN`. So the previous example now reads

```
$ cat file.sam | ngless -pe 'as_reads(select(samfile(STDIN), keep_if=[{mapped}]))' >␣
↪file.fq
```

Obviously, this example would more interesting if the input were to come from another programme (not just `cat`).

Full documentation

Frequently Asked Questions (FAQ)

CHAPTER 4

Building and installing

Again, please note that this is pre-release software. Thus, we do not provide any easy to install (pre-built) packages at the moment, but they will be provided once the software is released. However, any comments (including bug and build reports), are more than welcome.

stack is highly recommended. Install it and running `stack build` should (1) download all dependencies with the correct versions and (2) build ngless. It will perform this task in its own sandbox so it will not interfere with any other work.

CHAPTER 5

Authors

- Luis Pedro Coelho (email: coelho@embl.de) (on twitter: [@luispedrocoelho](https://twitter.com/luispedrocoelho))
- Paulo Monteiro
- Ana Teresa Freitas

Introduction

Motivation

Nearly all next generation sequence (NGS) applications rely on sequence alignment as the first analysis step. Before the alignment they require some kind of pre-processing of data, that is always dependent on the researcher interest. Our objective is to allow the creation of a pipeline of work for all the first phase of NGS analysis until the point (inclusive) of annotation. We want to do this while achieving the following goals:

- Ease the development of NGS Tools;
- Enable an easy identification of errors;
- Easily reproduce an experiment;
- Easy configuration and execution of workflows;
- Exploit available computational resources.

Target Users

Bioinformaticians working in a wetlab setting. Every serious biological lab in the world now needs to hire at least one. They know programming (at least basic programming), but are not method developers.

The tool can still be useful for more advanced users.

Basic Properties

- The syntax is a pythonesque syntax with Ruby-like blocks.
- The types are statically and strictly.
- Types are implicit, but limited language allows for type inference and checking.
- Quality control is implicit and mandatory (you get it for free)
- Types are domain types (biological).

Installation

NOTE: As of July 2016, ngless is available only as a pre-release to allow for testing of early versions by fellow scientists. We do not recommend use in production. Please get in touch if you are interested in using ngless in your projects.

At the moment, ngless can be esily obtained in binary form for Windows (experimental) and with either *brew* or *nix* (for Mac OS X or Linux). Of course, you can also compile from source.

Windows

Download and run the Windows Installer. The result is a command line utility, so you need to run it on the command line. After running the installer, typing `ngless` on the terminal should work as the installer will add the right directories to the path variable; you may have to start a new terminal, though. It should also work under Cygwin (but Cygwin is **not** a dependency).

The Windows package includes bwa and samtools. The bwa and samtools executables are available as `ngless-0.0.0-bwa` and `ngless-0.0.0-samtools`, respectively. It has been tested on Windows 10, but this has not had as intensitive testing as the Linux/Mac OS X versions so any bug reports are appreciated.

From source

Stack is the simplest way to install the necessary requirements. You should also have `gcc` installed (or another C-compiler).

The following sequence of commands should download and build the software

```
git clone https://github.com/luispedro/ngless
cd ngless
make
```

The first time you run this, it will take a while as it will download all dependencies. After this ngless is ready to use!

With Nix

If you use nix, you can easily build and install ngless using it (these scripts also install all necessary dependencies):

```
nix-env -i -f https://github.com/luispedro/ngless-nix/archive/master.tar.gz
```

This should download the nix scripts and build ngless.

If you prefer, you can also first clone the repository:

```
git clone https://github.com/luispedro/ngless-nix
cd ngless-nix
# inspect the default.nix & ngless.nix files if you wish
nix-env -i -f .
```

Make targets

The following are targets in the Makefile.

make compiles NGLess and haskell dependencies

clean remove local generated files by compilation

check run tests

bench run benchmarks

Ocean Metagenomics Functional Profiling

In this tutorial, we will analyse a small dataset of oceanic microbial metagenomes.

Note: This tutorial uses the full Ocean Microbial Reference Gene Catalog presented in Structure and function of the global ocean microbiome Sunagawa, Coelho, Chaffron, et al., Science, 2015

This catalog contains ca. 40 million genes and requires **56GiB** of RAM

---

1. Download the toy dataset

First download all the tutorial data:

```
ngless --download-demo ocean-short
```

This will download and expand the data to a directory called `ocean-short`.

This is a toy dataset. It is based on real data, but the samples were trimmed so that they contains only 250k paired-end reads.

The dataset is organized in classical MOCAT style. Ngless does not require this structure, but this tutorial also demonstrates how to upgrade from your existing MOCAT-based projects.:

```
$ find
./SAMEA2621229.sampled
./SAMEA2621229.sampled/ERR594355_2.short.fq.gz
./SAMEA2621229.sampled/ERR594355_1.short.fq.gz
./SAMEA2621155.sampled
./SAMEA2621155.sampled/ERR599133_1.short.fq.gz
./SAMEA2621155.sampled/ERR599133_2.short.fq.gz
./SAMEA2621033.sampled
./SAMEA2621033.sampled/ERR594391_2.short.fq.gz
./SAMEA2621033.sampled/ERR594391_1.short.fq.gz
./tara.demo.short
./process.ngl
```

The whole script we will be using is there as well (`process.ngl`), so you can immediately run it with:

```
ngless process.ngl
```

The rest of this tutorial is an explanation of the steps in this script.

2. Preliminary imports

To run ngless, we need write a script. We start with a few imports:

```
ngless "0.0"
import "parallel" version "0.0"
import "mocat" version "0.0"
import "omrgc" version "0.0"
```

These will all be used in the tutorial.

3. Parallelization

We are going to process each sample separately. For this, we use the `lock1` function from the parallel module (which we imported before):

```
samples = readlines('tara.demo.short')
sample = lock1(samples)
```

The `readlines` function reads a file and returns all lines. In this case, we are reading the `tara.demo.short` file, which contains the three samples (`SAMEA2621229.sampled`, `SAMEA2621155.sampled`, and `SAMEA2621033.sampled`).

`lock1()` is a slightly more complex function. It takes a list and *locks one of the elements* and returns it. It always chooses an element which has not been locked before, so you each time you run _ngless_, you will get a different sample.

3. Preprocessing

First, we load the data (the FastQ files):

```
input = load_mocat_sample(sample)
```

And, now, we preprocess the data:

```
preprocess(input, keep_singles=False) using |read|:
    read = substrim(read, min_quality=25)
    if len(read) < 45:
        discard
```

3. Profiling using the OM-RGC

After preprocessing, we map the reads to the ocean microbial reference gene catalog:

```
mapped = map(input, reference='omrgc', mode_all=True)
```

The line above is the reason we needed to import the `omrgc` module: it made the `omrgc` reference available.

```
mapped = select(mapped, keep_if=[{mapped}, {unique}])
```

Now, we need to `count` the results. This function takes the result of the above and aggregates it different ways. In this case, we want to aggregate by KEGG KOs, and eggNOG OGs:

```
counts = count(mapped,
            features=['KEGG_ko', 'eggNOG_OG'],
            normalization={scaled})
```

5. Aggregate the results

We have done all this computation, now we need to save it somewhere. We will use the `collect()` function to aggregate across all the samples processed:

```
collect(counts
        current=sample,
        allneeded=samples,
        ofile='omgc.profiles.txt')
```

6. Run it!

This is our script. We save it to a file (`process.ngl` in this example) and run it from the command line:

```
$ ngless process.ngl
```

Note that we need a large amount (ca. 64GB) of RAM memory to be able to use the OM-RGC. You also need to run it once for each sample. However, this can be done in parallel, taking advantage of high performance computing clusters.

Full script

Here is the full script:

```
ngless "0.0"
import "parallel" version "0.0"
import "mocat" version "0.0"
import "omrgc" version "0.0"


samples = readlines('tara.demo.short')
sample = lock1(samples)
input = load_mocat_sample(sample)

preprocess(input, keep_singles=False) using |read|:
    read = substrim(read, min_quality=25)
    if len(read) < 45:
        discard

mapped = map(input, reference='omrgc', mode_all=True)
mapped = select(mapped, keep_if=[{mapped}, {unique}])
collect(
        count(mapped,
                features=['KEGG_ko', 'eggNOG_OG'],
                normalization={scaled}),
        current=sample,
        allneeded=samples,
        ofile='omgc.profile.txt')
```

Human Gut Metagenomics Functional & Taxonomic Profiling

In this tutorial, we will analyse a small dataset of human gut microbial metagenomes.

Note: This tutorial uses the Integrated Gene Catalogue and requires ca. **15GiB** of RAM

1. Download the toy dataset

First download all the tutorial data:

```
ngless --download-demo gut-short
```

This will download and expand the data to a directory called `gut-short`.

This is a toy dataset. It is based on real data, but the samples were trimmed so that they contains only 250k paired-end reads.

The dataset is organized in classical MOCAT style. Ngless does not require this structure, but this tutorial also demonstrates how to upgrade from your existing MOCAT-based projects.:

```
$ find
./igc.demo.short
./SAMN05615097.short
./SAMN05615097.short/SRR4052022.single.fq.gz
./SAMN05615097.short/SRR4052022.pair.2.fq.gz
./SAMN05615097.short/SRR4052022.pair.1.fq.gz
./SAMN05615096.short
./SAMN05615096.short/SRR4052021.pair.1.fq.gz
./SAMN05615096.short/SRR4052021.single.fq.gz
./SAMN05615096.short/SRR4052021.pair.2.fq.gz
./SAMN05615098.short
./SAMN05615098.short/SRR4052033.pair.2.fq.gz
./SAMN05615098.short/SRR4052033.pair.1.fq.gz
./SAMN05615098.short/SRR4052033.single.fq.gz
./process.ngl
```

The whole script we will be using is there as well (`process.ngl`), so you can immediately run it with:

```
ngless process.ngl
```

The rest of this tutorial is an explanation of the steps in this script.

2. Preliminary imports

To run ngless, we need write a script. We start with a few imports:

```
ngless "0.0"
import "parallel" version "0.0"
import "mocat" version "0.0"
import "motus" version "0.1"
import "igc" version "0.0"
```

These will all be used in the tutorial.

3. Parallelization

We are going to process each sample separately. For this, we use the `lock1` function from the parallel module (which we imported before):

```
samples = readlines('igc.demo.short')
sample = lock1(samples)
```

The `readlines` function reads a file and returns all lines. In this case, we are reading the `tara.demo.short` file, which contains the three samples (`SAMEA2621229.sampled`, `SAMEA2621155.sampled`, and `SAMEA2621033.sampled`).

`lock1()` is a slightly more complex function. It takes a list and *locks one of the elements* and returns it. It always chooses an element which has not been locked before, so you each time you run _ngless_, you will get a different sample.

3. Preprocessing

First, we load the data (the FastQ files):

```
input = load_mocat_sample(sample)
```

And, now, we preprocess the data:

```
preprocess(input, keep_singles=False) using |read|:
    read = substrim(read, min_quality=25)
    if len(read) < 45:
        discard
```

4. Filter against the human genome

We want to remove reads which map to the human genome, so we first map the reads to the human genome:

```
mapped = map(input, reference='hg19')
```

`hg19` is a built-in reference and the genome will be automatically download it the first time you use it. Now, we discard the matched reads:

```
mapped = select(mapped) using |mr|:
    mr = mr.filter(min_match_size=45, min_identity_pc=90, action={unmatch})
    if mr.flag({mapped}):
        discard
```

The `mapped` object is a set of `mappedreads` (i.e., the same information that is saved in a SAM/BAM file). we use the `as_reads` function to get back to reads:

```
input = as_reads(mapped)
```

Now, we will use the `input` object which has been filtered of human reads.

5. Profiling using the IGC

After preprocessing, we map the reads to the integrated gene catalog:

```
mapped = map(input, reference='igc', mode_all=True)
```

The line above is the reason we needed to import the `igc` module: it made the `igc` reference available.

Now, we need to `count` the results. This function takes the result of the above and aggregates it different ways. In this case, we want to aggregate by KEGG KOs, and eggNOG OGs:

```
counts = count(mapped,
            features=['KEGG_ko', 'eggNOG_OG'],
            normalization={scaled})
```

7. Aggregate the results

We have done all this computation, now we need to save it somewhere. We will use the `collect()` function to aggregate across all the samples processed:

```
collect(counts,
        current=sample,
```

```
        allneeded=samples,
        ofile='igc.profiles.txt')
```

9. Taxonomic profling using mOTUS

Map the samples against the `motus` reference (this reference comes with the motus module we imported earlier):

```
mapped = map(input, reference='motus', mode_all=True)
```

Now call the built-in `count` function to summarize your reads at gene level:

```
counted = count(mapped, features=['gene'], multiple={dist1})
```

To get the final taconomic profile, we call the `motus` function, which takes the gene count table and performs the motus quantification. The result of this call is another table, which we can concatenate with `collect()`:

```
motus_table = motus(counted)
collect(motus_table,
        current=sample,
        allneeded=samples,
        ofile='motus-counts.txt')
```

10. Run it!

This is our script. We save it to a file (`process.ngl` in this example) and run it from the command line:

```
$ ngless process.ngl
```

You also need to run it once for each sample. However, this can be done in parallel, taking advantage of high performance computing clusters.

Full script

Here is the full script:

```
ngless "0.0"
import "parallel" version "0.0"
import "mocat" version "0.0"
import "motus" version "0.1"
import "igc" version "0.0"

samples = readlines('igc.demo.short')
sample = lock1(samples)

input = load_mocat_sample(sample)

preprocess(input, keep_singles=False) using |read|:
    read = substrim(read, min_quality=25)
    if len(read) < 45:
        discard

mapped = map(input, reference='hg19')

mapped = select(mapped) using |mr|:
    mr = mr.filter(min_match_size=45, min_identity_pc=90, action={unmatch})
    if mr.flag({mapped}):
        discard
```

```
input = as_reads(mapped)


mapped = map(input, reference='igc', mode_all=True)

counts = count(mapped,
            features=['KEGG_ko', 'eggNOG_OG'],
            normalization={scaled})

collect(counts,
        current=sample,
        allneeded=samples,
        ofile='igc.profiles.txt')

mapped = map(input, reference='motus', mode_all=True)

counted = count(mapped, features=['gene'], multiple={dist1})

motus_table = motus(counted)
collect(motus_table,
        current=sample,
        allneeded=samples,
        ofile='motus-counts.txt')
```

Tutorial

Example

This example will use data from a real experiment stored at EMBL-EBI. The data can be accessed at http://www.ebi.ac.uk/ena/data/view/SRP023199 and represent **HeLa cells**. The idea is to preprocess the data set, map it against the human genome and count the reads that overlap with known genes.

We will use the fastQ file ftp://ftp.sra.ebi.ac.uk/vol1/fastq/SRR867/SRR867735/SRR867735.fastq.gz that can be accessed in the table, on column **Sample accession**, with value SAMN02179475.

Load fastQ file

Before creating the whole script lets start by understanding our data set. This first step will allow you to perform quality control.

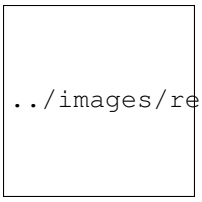
```
ngless "0.0"

/* load the data set */
input = fastq('SRR867735.fastq.gz')
```

You can now save the script (as *test.ngl* for example) to the directory where the file `SRR867735.fastq.gz` is and run ngless:

```
$ ngless test.ngl
```

Using a web browser, you can open the file `test.output_ngless/index.html` to see information about a data set and the ngless job. At 'Before QC' there will be the result of the execution.



We can now see that the data set has:

- +/- 50% of guanine and cytosine.
- Follows the Encoding Sanger.
- Has 32456161 sequences
- And all sequences have the same length (50).

Also, by analyzing the plot we can see that the first 3 base pairs, on average, have the lowest quality (31.0). So, a good preprocess starts by removing the first 3 base pairs.

Feel free to explore all the available statistics.

Preprocess

For the preprocessing of the data we will:

- Remove the first **3** base pairs.
- Substrim with a minimum quality of **15**.
- Discard if the length of a read is **smaller than 20**.

Let's add the following code to the already existent code:

```
preprocess(input) using |read|:
read = read [3:] // Discard from position 0 until 3 (excluded).
read = substrim(read, min_quality=15)
if len(read) < 20:
 discard
```

The *using |var|* syntax is similar to Ruby's blocks or lambda functions in other languages. The whole block after using is executed for each read in *input*, each time assigning it to the variable *read*.

This will generate quality control that will be detailed at the execute section.

Map

After adding the preprocess code, it's time to map against the human genome. Since the human genome is provided by default, you can simply do:

```
/* reference genome */
human = 'hg19'
mapped = map(input, reference=human)
```

Counting

We are only interested in the human genes so lets annotate the mapping results to the corresponding genes. Since we used a genome provided by NGLess, we do not need to specify which annotation file to use (it'll be built in):

```
/* features to annotate */
feats = ['gene']
counts = count(mapped, multiple={dist1}, keep_ambiguous=false, features=feats)
```

You can also see the use of some symbol arguments (symbols are the special strings inside braces, like *{dist1}*). Symbols are like strings, except that when a function takes a symbol, that means that there is a set of predefined values it can take. So, for example, the function *count* takes a *multiple* argument which defines how to count reads which can be assigned to mulitple features. The options are *{all1}* (count all equally as 1), *{1overN}* (distribute equally across all candidates, i.e., increment them by 1/N), or *{dist1}* (distribute multiple features by using the singly mapped features as a baseline). In practice, the difference between strings and symbols is that symbols are, as much as possible, checked at the start of interpretation (if you write *{all2}*, you will immediately get a message "did you mean all1?" before interpretation starts or if you run the script with -n, which just performs this validation).

Write to disk

Finally, we write the results to a file:

```
/* write counts to disk */
write(counts, ofile="samples/CountsResult.txt")
```

Execute

You can now save the script (as **test.ngl** for example) to the directory where the file 'SRR867735.fastq.gz' is and run ngless.

```
$ ngless test.ngl
```

As a result of the execution, should be returned the following:

```
Total reads: 31654060
Total reads aligned: 28095945[88.76%]
Total reads Unique map: 22434229[79.85%]
Total reads Non-Unique map: 5661716[20.15%]
Total reads without enough qual: 0
```

These are statistics of the map of the file against the human genome.

NGLess one liners

`ngless` can be used as a traditional command line transformer utility, using the `-e` argument to pass an inline script on the command line.

The `-p` (or `--print-last`) argument tells ngless to output the value of the last expression to *stdout*.

Converting a SAM file to a FASTQ file

```
$ ngless -pe 'as_reads(samfile("file.sam"))' > file.fq
```

This is equivalent to the full script:

```
ngless "0.0" # <- version declaration, optional on the command line
samcontents = samfile("file.sam") # <- load a SAM/BAM file
reads = as_reads(samcontents) # <- just get the reads (w quality scores)
write(reads, ofname=STDOUT) # <- write them to STDOUT (default format: FASTQ)
```

This only works if the data in the samfile is single ended as we pipe out a single FQ file. Otherwise, you can always do:

```
ngless "0.0"
write(as_read(samfile("file.sam")),
        ofile="output.fq")
```

which will write 3 files: `output.1.fq`, `output.2.fq`, and `output.singles.fq` (the first two for the paired-end reads and the last one for reads without a mate).

Getting aligned reads from a SAM file as FASTQ file

Building on the previous example. We can add a `select()` call to only output unmapped reads:

```
$ ngless -pe 'as_reads(select(samfile("file.sam"), keep_if=[{mapped}]))' > file.fq
```

This is equivalent to the full script:

```
ngless "0.0" # <- version declaration, optional on the command line
samcontents = samfile("file.sam") # <- load a SAM/BAM file
samcontents = select(samcontents, keep_if=[{mapped}]) # <- select only *mapped* reads
reads = as_reads(samcontents) # <- just get the reads (w quality scores)
write(reads, ofname=STDOUT) # <- write them to STDOUT (default format: FASTQ)
```

Reading from STDIN

For a true Unix-like utility, the input should be read from standard input. This can be achieved with the special file `STDIN`. So the previous example now reads:

```
$ cat file.sam | ngless -pe 'as_reads(select(samfile(STDIN), keep_if=[{mapped}]))' >␣
↪file.fq
```

Obviously, this example would more interesting if the input were to come from another programme (not just `cat`).

Preprocessing FastQ Data

Preprocessing FastQ files consists of quality trimming and filtering of reads as well as (possible) elimination of reads which match some reference which is not of interest.

Quality-based filtering

Filtering reads based on quality is performed with the `preprocess` function, which takes a block of code. This block of code will be executed for each read. For example:

```
ngless "0.0"

input = fastq('input.fq.gz')

preprocess(input) using |r|:
    r = substrim(r, min_quality=20)
    if len(r) < 45:
        discard
```

If it helps you, you can think of the `preprocess` block as a `foreach` loop, with the special keyword `discard` that removes the read from the collection. Note that the name `r` is just a variable name, which you choose using the `|r|` syntax.

Within the preprocess block, you can modify the read in several ways:

- you can trim it with the indexing operator: `r[trim5:]` or `r[:-trim3]`
- you can call `substrim` or `endstrim` to trim the read based on quality scores. `substrim` finds the longest substring such that all bases are above a minimum quality (hence the name, which phonetically combines substring and trim). `endstrim` just chops bases off the ends.
- you can test for the length of the sequence (before or after trimming). For this, you use the `len` function (see example above).
- you can test for the average quality score (using the `avg_quality()` method).

You can combine these in different ways. For example, the behaviour of the fastx quality trimmer can be recreated as:

```
preprocess(input) using |r|:
    r = endstrim(r, min_quality=20)
    if r.fraction_at_least(20) < 0.5:
        discard
    if len(r) < 45:
        discard
```

Handling paired end reads

When your input is paired-end, the preprocess call above will handle each mate independently. Three things can happen:

1. both mates are discarded,
2. both mates are kept (i.e., not discarded),
3. one mate is kept, the other discarded.

The only question is what to do in the third case. By default, the `preprocess` call keep the mate turning the read into an unpaired read (a single), but you can change that behaviour by setting the `keep_singles` argument to `False`:

```
preprocess(input, keep_singles=False) using |r|:
    r = substrim(r, min_quality=20)
    if len(r) < 45:
        discard
```

Now, the output will consist of only paired-end reads.

Filtering reads matching a reference

It is often also a good idea to match reads against some possible contaminant database. For example, when studying the host associated microbiome, you will often want to remove reads matching the host. It is almost always a good to at least check for human contamination (during lab handling).

For this, you map the reads against the human genome:

```
mapped_hg19 = map(input, reference='hg19')
```

Now, `mapped_hg19` is a set of mapped reads. Mapped reads are reads, their qualities, plus additional information of how they matched. Mapped read sets are the internal ngless representation of SAM files.

To filter the set, we will `select`. Like `preprocess`, `select` also uses a block for the user to specify the logic:

```
mapped_hg19 = select(mapped_hg19) using |mr|:
    mr = mr.filter(min_match_size=45, min_identity_pc=90, action={unmatch})
    if mr.flag({mapped}):
        discard
```

We first set a minimum match size and identity percentage to avoid spurious hits. **We keep the reads** but **unmatch** them (i.e., we clear any information related to a match). Then, we discard any reads that match by checking the flag `{mapped}`.

Finally, we convert the mapped reads back to simple reads using the `as_reads` function (this discards the matching information):

```
input = as_reads(mapped_hg19)
```

Now, `input` can be passed to the next step in the pipeline.

Functions

These are the built-in ngless functions. Make sure to check the standard library as well.

fastq

Function to load a FastQ file:

```
in = fastq('input.fq')
```

Argument:

String

Return:

ReadSet

Arguments by value:

Name	Type	Required	Default Value
encoding	Symbol ({auto}, {33}, {64}, {sanger}, {solexa})	no	{auto}

Possible values for `encoding` are:

- `{sanger}` or `{33}` assumes that the file is encoded using sanger format. This is appropriate for newer Illumina outputs.
- `{solexa}` or `{64}` assumes that the file is encoded with a 64 offset. This is used for older Illumina/Solexa machines.
- `{auto}`: use auto detection. This is the default.

When loading a data set, quality control is carried out and statistics can be visualised in a graphical user interface (GUI). Statistics calculated are:

- percentage of guanine and cytosine (%GC)
- number of sequences
- minimum/maximum sequence length
- mean, median, lower quartile and upper quality quartile for each sequence position

If not specified, the encoding is guessed from the file.

Gzip and bzip2 compressed files are transparently supported (determined by file extension, `.gz` and `.bz2` for gzip and bzip2 respectively).

paired

Function to load a paired-end sample, from two FastQ files:

```
in = paired('input.1.fq', 'input.2.fq', singles='input.3.fq')
```

`paired()` is an exceptional function which takes **two** unnamed arguments, specifying the two read files (first mate and second mate) and an optional `singles` file (which contains unpaired reads).

Argument:

String, String

Return:

ReadSet

Arguments by value:

Name	Type	Required	Default Value
encoding	Symbol ({auto}, {33}, {64}, {sanger}, {solexa})	no	{auto}
singles	String	no	•

The `encoding` argument has the same meaning as for the `fastq()` function:

- `{sanger}` or `{33}` assumes that the file is encoded using sanger format. This is appropriate for newer Illumina outputs.
- `{solexa}` or `{64}` assumes that the file is encoded with a 64 offset. This is used for older Illumina/Solexa machines.
- `{auto}`: use auto detection. This is the default.

samfile

Loads a SAM file:

```
s = samfile('input.sam')
```

This function takes no keyword arguments. BAM files are also supported (determined by the filename), as are `sam.gz` files.

Returns

MappedReadSet

countfile

Loads a TSV file:

```
c = countfile('table.tsv')
```

This function takes no keyword arguments. If the filename ends with ".gz", it is assumed to be a gzipped file.

Returns

CountTable

as_reads

Converts from a `MappedReadSet` to a `ReadSet`:

```
reads = as_reads(samfile('input.sam'))
```

unique

Function that given a set of reads, returns another which only retains a set number of copies of each read (if there are any duplicates). An example:

```
input = unique(input, max_copies=3)
```

Argument:

ReadSet

Return:

ReadSet

Arguments by value:

Name	Type	Required	Default Value
max_copies	Integer	no	2

The optional argument **max_copies** allows to define the number of tolerated copies (default: 2).

Two short reads with the same nucleotide sequence are considered copies, independently of quality and identifiers.

This function is currently limited to single-end samples.

preprocess

This function executes the given block for each read in the ReadSet. Unless the read is **discarded**, it is transferred (after transformations) to the output. The output is assigned to the same name as the inputs. For example:

```
preprocess(inputs) using |read|:
    read = read[3:]
```

Argument:

ReadSet

Return:

Void

Arguments by value:

Name	Type	Required	Default Value
keep_singles	bool	no	true

When a paired-end input is being preprocessed in single-mode (i.e., each mate is preprocessed independently, it can happen that on eof the mates is discarded, while the other is kept). The default is to collect these into the singles pile. If `keep_singles` if false, however, they are discarded.

This function also performs quality control on its output.

map

The function map, maps a ReadSet to reference. For example:

```
mapped = map(input, reference='sacCer3')
mapped = map(input, fafile='ref.fa')
```

Argument:

ReadSet

Return:

MappedReadSet

Arguments by value:

Name	Type	Required	Default Value
reference	String	no	•
fafile	String	no	•
mode_all	Bool	no	•

The user must provide either a path to a FASTA file in the `fafile` argument or the name of a builtin reference using the `reference` argument.

NGLess provides the following builtin datasets:

Name	Description	Assembly
sacCer3	saccharomyces_cerevisiae	R64-1-1
ce10	caenorhabditis_elegans	WBcel235
dm3	drosophila_melanogaster	BDGP5
gg4	gallus_gallus	Galgal4
canFam2	canis_familiaris	CanFam3.1
rn4	rattus_norvegicus	Rnor_5.0
bosTau4	bos_taurus	UMD3.1
mm10	mus_musculus	GRCm38
hg19	homo_sapiens	GRCh38

To use any of these, pass in the name as the reference value:

```
mapped_hg19 = map(input, reference='hg19')
```

Ngless does not ship with any of these datasets, but they are downloaded lazily: i.e., the first time you use them, ngless will download and cache them.

The option `mode_all=True` can be passed to include all alignments of both single and paired-end reads in the output SAM/BAM.

mapstats

Computes some basic statistics from a set of mapped reads (number of reads, number mapped, number uniquely mapped).

Argument

MappedReadSet

Return

CountTable

select

select filters a MappedReadSet. For example:

```
mapped = select(mapped, keep_if=[{mapped}])
```

Argument:

MappedReadSet

Return:

MappedReadSet

Arguments by value:

Name	Type	Required	Default Value
keep_if	[Symbol]	no	•
drop_if	[Symbol]	no	•
paired	Bool	no	true

At least one of `keep_if` or `drop_if` should be passed, but not both. They accept the following symbols:

- `{mapped}`: the read mapped
- `{unmapped}`: the read did not map
- `{unique}`: the read mapped to a unique location

If `keep_if` is used, then reads are kept if they pass **all the conditions**. If `drop_if` they are discarded if they fail to **any condition**.

By default, `select` operates on a paired-end read as a whole. If `paired=False` is passed, however, then link between the two mates is not considered and each read is processed independently.

count

Given a file with aligned sequencing reads (ReadSet), `count()` will produce a counts table depending on the arguments passed. For example:

```
counts = count(mapped, min=2, mode={union}, multiple={dist1})
```

Argument:

MappedReadSet

Return:

CountTable

Arguments by value:

Name	Type	Required	Default value
gff_file	String	no*	•
functional_map	String	no*	•
features	[String]	no	'gene'
mode	Symbol	no	{union}
multiple	Symbol	no	{dist1}
strand	Bool	no	false
normalization	Symbol	no	{raw}
include_minus1	Bool	no	false
min	Integer	no	0
discard_zeros	Bool	no	false

If the features to count are `['seqname']`, then each read will be assigned to the name of reference it matched and only an input set of mapped reads is necessary. For other features, you will need extra information. This can be passed using the `gff_file` or `functional_map` arguments. If you had previously used a `reference` argument for the `map()` function, then you can also leave this argument empty and ngless will do the right thing.

`features`: which features to count.

`mode` indicates how to handle reads that partially overlap a features. Possible values for `mode` are `{union}`, `{intersection-strict}`, and `{intersection-nonempty}` (default: `{union}`). For each read position are obtained features that intersect it, which is known as sets. The different modes are:

- `{union}` the union of all the sets.
- `{intersection-strict}` the intersection of all the sets.
- `{intersection-nonempty}` the intersection of all non-empty sets.

How to handle multiple mappers (inserts which have more than one "hit" in the reference) is defined by the `multiple` argument:

- `{unique_only}`: only use uniquely mapped inserts
- `{all1}`: count all hits separately. An insert mapping to 4 locations adds 1 to each location

- {1overN}: fractionally distribute multiple mappers. An insert mapping to 4 locations adds 0.25 to each location
- {dist1}: distribute multiple reads based on uniquely mapped reads. An insert mapping to 4 locations adds to these in proportion to how uniquely mapped inserts are distributed among these 4 locations.

Argument strand represents whether the data are from a strand-specific (default is false). When the data is not strand-specific, a read is always overlapping with a feature independently of whether maps to the same or the opposite strand. For strand-specific data, the read has to be mapped to the same strand as the feature.

min defines the minimum amount of overlaps a given feature must have, at least, to be kept (default: 0, i.e., keep all counts). If you just want to discard features that are exactly zero, you should set the discard_zeros argument to True.

normalization specifies if and how to normalize to take into account feature size:

- {raw} (default) is no normalization
- {normed} is the result of the {raw} mode divided by the size of the feature
- {scaled} is the result of the {normed} mode scaled up so that the total number of counts is identical to the {raw} (within rounding error)

Unmapped inserts are included in the output if {include_minus1} is true (default: False).

substrim

Given a read, returns another that is the biggest sub-sequence with a given minimum quality. For example:

```
read = substrim(read, min_quality=25)
```

Argument:

ShortRead

Return:

ShortRead

Arguments

Name	Type	Required	Default Value
min_quality	Integer	yes	

min_quality parameter defines the minimum quality accepted for the sub-sequence.

endstrim

Given a read, trim from both ends (5' and 3') all bases below a minimal quality. For example:

```
read = endstrim(read, min_quality=25)
```

Argument:

ShortRead

Return:

ShortRead

Arguments

Name	Type	Required	Default Value
min_quality	Integer	yes	

`min_quality` parameter defines the minimum quality value.

write

Writes an object to disk.

ReadSet

Argument:

Any

Return:

Void

Arguments by value:

Name	Type	Required	Default Value
ofile	String	yes	•
format	String	no	•

The argument `ofile` is where to write the content.

The output format is typically determined from the `ofile` extension, but the `format` argument overrides this. Supported formats:

- CountsTable: `{tsv}` (default) or `{csv}`: use TAB or COMMA as a delimiter
- MappedReadSet: `{sam}` (default) or `{bam}`
- ReadSet: FastQ format, optionally compressed (depending on the extension).

print

Print function allows to print a NGLessObject to IO.

Argument:

NGLessObject

Return:

Void

Arguments by value:

none

readlines

Reads a text file and returns a list with all the strings in the file

Argumment

string: the filename

Example

`readlines` is useful in combination with the parallel module, where you can then use the `lock1` function to process a large set of inputs:

```
sample = lock1(readlines('samplelist.txt'))
```

Methods

Methods are invoked using an object-oriented syntax. For example:

```
mapped = select(mapped) using |mr|:
    mr = mr.pe_filter()
```

They can also take arguments

```
mapped = select(mapped) using |mr|:
    mr = mr.filter(min_match_size=30)
```

Mapped reads

Mapped reads contain several methods. *None of these methods changes its argument, they return new values.* The typical approach is to reassign the result to the same variable as before (see examples above).

- `pe_filter`: only matches where both mates match are kept.
- `flag`: Takes one of `{mapped}` or `{unmapped}` and returns true if the reads were mapped (in a paired-end setting, a read is considered mapped if at least one of the mates mapped).
- `some_match`: Takes a reference name and returns True if the read mapped to that reference name.

filter

`filter` takes a mapped read and returns a mapped read according to several criteria:

- `min_match_size`: minimum match size
- `min_identity_pc`: minimum percent identity (considered over the matching location, trimming on the left and right are excluded).

If more than one test is specified, then they are combined with the AND operation (i.e., all conditions have to be fulfilled for the test to be true).

The default is to discard mappings that do not pass the test, but it can be changed with the `action` argument, which must be one of `{drop}` (default: the read is excluded from the output), or `{unmatch}` (the read is changed so that it no longer reports matching).

You can pass the flag `reverse` (i.e., `reverse=True`) to reverse the sign of the test.

Standard library

The standard library in ngless are a set of modules which are built in to the language.

Parallel module

This module allows you to run several parallel computations. It provides two functions: `lock1` and `collect`.

`lock1 :: [string] -> string` takes a list of strings and returns a single element. It uses the filesystem to obtain a lock file so that if multiple processes are running at once, each one will return a different element. Ngless also marks results as *finished* once you have run a script to completion.

The intended usage is that you simply run as many processes as inputs that you have and ngless will figure everything out.

For example

```
ngless "0.0"
import "parallel" version "0.0"

samples = ['Sample1', 'Sample2', 'Sample3']
current = lock1(samples)
```

Now, current will be one of `'Sample1'`, `'Sample2'`, or `'Sample3'`. You can use this to find your input data:

```
input = paired("data/" + current + ".1.fq.gz", "data/" + current + ".2.fq.gz")
```

Often, it's a good idea to combine `lock1` with `readlines` (a function which returns the contents of all the non-empty lines in a file as a list of strings):

```
samples = readlines('samples.txt')
current = lock1(samples)
input = paired("data/" + current + ".1.fq.gz", "data/" + current + ".2.fq.gz")
```

You now use `input` as in any other ngless script:

```
mapped = map(input, reference='hg19')
write(input, ofile='outputs/'+current+ '.bam')
counts = count(mapped)
write(counts, ofile='outputs/'+current+ '.txt')
```

This will result in both BAM files and counts being written to the `outputs/` directory. The module also adds the `collect` function which can paste all the counts together into a single table, for convenience:

```
collect(
    counts,
    current=current,
    allneeded=samples,
    ofile='outputs/counts.txt.gz')
```

Now, only when all the samples in the `allneeded` argument have been processed, does ngless collect all the results into a single table.

Full "parallel" example

```
ngless "0.0"
import "parallel" version "0.0"

sample = lock1(readlines('input.txt'))
input = fastq(sample)
mapped = map(input, reference='hg19')
collect(count(mapped, features=['seqname']),
    current=sample,
    allneeded=readlines('input.txt'),
    ofile='output.tsv')
```

Now, you can run multiple `ngless` jobs in parallel and each will work on a different line of `input.txt`.

Parallel internals

Normally this should be invisible to you, but if you are curious or want to debug an issue, here are the gory details:

The function `lock1()` will create a lock file in a sub-directory of `ngless-locks`. This directory will be named by the hash value of the script. Thus, any change to the script will force all data to be recomputed. This can lead to over-computation but it ensures that you will always have the most up to date results (ngless' first priority is correctness, performance is important, but not at the risk of correctness). Similarly, `collect()` will use hashed values which encode both the script and the position within the script (so that if you have more than one `collect()` call, they will not clash).

Lock files have their modification times updated once every 10 minutes while ngless is running. This allows the programme to easily identify stale files. The software is very conservative, but any lock file with a modification time older than one hour is considered stale and removed. Note that because ngless is extremely careful about how writing its outputs atomically, the worse that can happen from mis-identifying a stale lock (for example, you had a compute

node which lost network connectivity, but it comes back online after an hour and resumes processing) is that extra computation is wasted, **the processes will never interfere in a way that you get erroneous results**.

Samtools module

This module exposes the samtools sorting functionality through the `samtools_sort` function.

```
ngless '0.0'
import "samtools" version "0.0"
to_sort = samfile('input.bam')
sorted = samtools_sort(to_sort)
write(sorted, ofile='input.sorted.bam')
```

`samtools_sort :: mappedreadset -> mappedreadset` returns a sorted version of the dataset.

Internally, this function calls ngless' version of samtools while respecting your settings for the use of threads and temporary disk space. When combined with other functionality, ngless can also often stream data into/from samtools instead of relying on intermediate files (these optimizations should not change the visible behaviour, only make the computation faster).

Mocat module

```
import "mocat" version "0.0"
```

This is a MOCAT compatibility layer to make it easier to adapt projects from MOCAT to ngless.

Functions

`load_mocat_sample :: string -> readset` this function takes a directory name and returns a set of reads by scanning the directory for (compressed) FastQ files. This is slightly more flexible than MOCAT2 as it also accepts files with the extension `fastq` or `fastq.gz` as well as `_1` and `_2` to indicate the two mate files.

`coord_file_to_gtf :: string -> string` this function takes a MOCAT-style `.coord`, converts it internally to a GTF file and returns it.

Example usage:

```
ngless "0.0"
import "mocat" version "0.0"

sample = load_mocat_sample('Sample1')
mapped = map(sampled, fafile='data/catalog.padded.fna')
write(count(mapped, gff_file=coord_file_to_gtf('data/catalog.padded.coord')),
    ofile='counts.txt')
```

This module can be combined with the parallel module (see above) to obtain a very smooth upgrade from MOCAT to ngless.

Modules

To add a module to ngless there are two options: *external* or *internal* modules. External modules are the simplest option.

External modules

External modules can perform two tasks:

1. Add new references to ngless
2. Add functions to ngless

Adding references makes them available to the `map()` call using the `reference` argument and (optionally) allows for calls to `count()` without specifying any annotation file.

Like everything else in ngless, these are versioned for reproducibility so that the resulting script implicitly encodes the exact version of the databases used.

Functions in external modules map to command line calls to a script you provide.

How to define an external module

You can use the example module in the ngless source for inspiration. That is a complete, functional module.

A module is defined by an `YaML` file.

Every module has a name and a version:

```
name: 'module'
version: '0.0.0'
```

Everything else is optional.

References

References are added with a *references* section, which is a list of references. A reference contains a `fasta-file` and (optionally) a `gtf-file`. For example:

```
references:
    -
        name: 'ref'
        fasta-file: 'data/reference.fna'
        gtf-file: 'data/reference.gtf.gz'
```

Note that the paths are relative to the module directory. The GTF file may be gzipped.

Inialization

An `init` section defines an initialization command. This will be run **before** anything else in any script which imports this module. The intention is that the module can check for any dependencies and provide the user with an early error message instead of failing later after. For example:

```
init:
    init_cmd: './init.sh'
    init_args:
        - "Hello"
        - "World"
```

will cause ngless to run the command `./init.sh Hello World` whenever a user imports the module.

A note about paths: paths you define in the module.yaml file are *relative to the Yaml file itself*. Thus you put all the necessary scripts and data in the module directory. However, the scripts are run with the current working directory of wherever the user is running the ngless protocol (so that any relative paths that the user specifies work as expected). To find your data files inside your module, ngless sets the environmental variable `NGLESS_MODULE_DIR` as the path to the module directory.

Functions

To add new functions, use a `functions` section, which should contain a list of functions encoded in YaML format. Each function has a few required arguments:

`nglName` the name by which the function will be called **inside** of an ngless script.

`arg0` the script to call for this function. Note that the user will never see this.

For example:

```
functions:
    -
        nglName: "test"
        arg0: "./run-test.sh"
```

will enable the user to call a function `test()` which will translate into a call to the `run-test.sh` script (see the note above about paths).

You can also add arguments to your function, naturally. Remember that ngless functions can have only one unnamed argument and any number of named arguments. To specify the unnamed argument add a `arg1` section, with the key `atype` (argument type):

```
        arg1:
            atype: <one of 'readset'/'mappedreadset'/'counts'/'str'/'flag'/'int'/
↪'option'>
```

The arguments of type *readset*, *mappedreadset*, and *counts* are passed as paths to a file on disk. **Your command is assumed to not change these, but make a copy if necessary. Bad things will happen if you change the files.** You can specify more details on which kind of file you expect with the following optional arguments:

```
            filetype: <one of "tsv"/"fq1"/"fq2"/"fq3"/"sam"/"bam"/"sam_or_bam"/"tsv">
            can_gzip: true/false
            can_bzip2: true/false
            can_stream: true/false
```

The flags `can_gzip`/`can_bzip2` indicate whether your script can accept compressed files (default: *False*). `can_stream` indicates whether the input can be a pipe (default: *False*, which means that an intermediate file will always be used).

For example, if your tool wants a SAM file (and never a BAM file), you can write:

```
        arg1:
            atype: mappedreadset
            filetype: sam
```

`ngless` will ensure that your tool does receive a SAM file (including converting BAM to SAM if necessary).

Finally, additional argument are specified by a list called `additional`. Entries in this list have exactly the same format as the `arg1` entry, except that they have a few extra fields. The extra field `name` is mandatory, while everything else is optional:

```
    additional:
        -
            name: <name>
            atype: <as for arg1: 'readset'/'mappedreadset'/...>
            def: <default value>
            required: true/false
```

Arguments of type `flag` have an optional extra argument, `when-true` which is a list of strings which will be passed as extra arguments when the flag is true. You can also just specify a single string. If `when-true` is missing, ngless will pass an option of the form `--name` (i.e., a double-dash then the name used). For example:

```
    additional:
        -
            name: verbose
            atype: bool
            def: false
            when-true: "-v"
        -
            name: complete
            atype: bool
            def: false
            when-true:
                - "--output=complete"
                - "--no-filter"
```

All other argument types are passed to your script using the syntax `--name=value` if they are present or if a default has been provided.

Arguments of type `option` map to symbols in ngless and require you to add an additional field `allowed` specifying the universe of allowed symbols. Ngless will check that the user specifies arguments from the allowable universe. For example:

```
    additional:
        -
            atype: 'option'
            name: 'verbosity'
            def: 'quiet'
            allowed:
                - 'quiet'
                - 'normal'
                - 'loud'
```

If you do not have a fixed universe for your argument, then it should be a `str` argument.

The `required` flag determines whether the argument is required. Note that arguments with a default argument are automatically optional (ngless may trigger a warning if you mark an argument with a default as required).

To return a value, you must request that ngless generate a new temporary file for the script to generate output to. Therefore, you need to specify a `return` section, with three parameters: `rtype` (return type, see below), `name` the name of the argument to use, and `extension` the file extension of the output type.

```
return:
    rtype: "counts"
    name: "ofile"
    extension: "sam"
```

`rtype` must be one of `"void"`, `"counts"`, `"readset"`, or `"mappedreadset"`.

Citation

Finally, if you wish to, you can add a citation:

```
citation: "A paper which you want to be listed when users import your module"
```

This will be printed out whenever users use your module and thus will help you get exposure.

Internal Modules

This is very advanced as it requires writing Haskell code which can then interact very deeply with the rest of ngless.

For an example, you can look at the example internal module. If you want to get started, you can ask about details on the ngless user mailing list.

Constants

In *ngless*, any variable written in uppercase is a constant.

Built in constants

- ARGV

This is string array which contains the arguments passed to the script

- STDIN

Use in place of a filename to read from standard input

- STDOUT

Use in place of a filename to write to standard output

For example:

```
ngless '0.0'

input = samfile(STDIN)
input = select(input) using |mr|:
    if mr.flag({mapped}):
        discard
write(input, ofile=STDOUT, format={bam})
```

This file reads a sam stream from stdin, filters it (using the `select` call) and writes to standard output in `bam` format.

Available Reference Genomes

NGLess provides builtin support for the most widely used model organisms (human, mouse, yeast, C. elegans, ...; see the full table below). This makes it easier to use the tool when using these organisms as some knowledge is already built in.

Genome references available

NGLess provides archives containing data sets of organisms. Is also provided gene annotations that provide information about protein-coding and non-coding genes, splice variants, cDNA and protein sequences, non-coding RNAs.

The following table represents organisms provided by default:

Name	Description	Assembly
hg19	homo_sapiens	GRCh38
sacCer3	saccharomyces_cerevisiae	R64-1-1
ce10	caenorhabditis_elegans	WBcel235
dm3	drosophila_melanogaster	BDGP5
canFam2	canis_familiaris	CanFam3.1
rn4	rattus_norvegicus	Rnor_5.0
bosTau4	bos_taurus	UMD3.1
mm10	mus_musculus	GRCm38

These archives are all created using version 75 of Ensembl.

Automatic installation

The builtin datasets are downloaded the first time they are used. They are downloaded to the user home directory and stored in **home/**.ngless/genomes.

Manual installation

Is possible to install data sets locally, before running any script. They can be installed in **User** mode or in **Root** mode.

To install locally (organism bos taurus), use the following command:

```
$ ngless --install-reference-data bosTau4
```

If you install as a super-user, then the dataset will be available for all users:

```
$ sudo ngless --install-reference-data bosTau4
```

When attempting to install an organism if is returned **True** it means that the organism is already installed, and there is no reason to install again. Otherwise, a progress bar is displayed to provide information on the download.

Data Set Structure

This section provides the technical details necessary if you wish to build your own reference for others to use automatically. For most users, it will likely be easier to directly specify the references in the ngless script.

The archives provided by NGLess contain BWA index files, the genome reference file and a gene annotation file.

```
Name.tar.gz
 |
 |--- Sequence
 |      |
 |      |-- BWAIndex
 |              |-- genome.fa.gz
 |              |-- genome.fa.gz.amb
 |              |-- genome.fa.gz.ann
 |              |-- genome.fa.gz.bwt
```

```
|              |-- genome.fa.gz.pac
|              |-- genome.fa.gz.sa
|
|--- Annotation
       |-- annot.gtf.gz
```

The basename of Description.tar.gz (Description) will have the description name of the respective organism (i.e, Mus_musculus.tar.gz).

Taxonomic profiling using mOTUs with ngless

You can use ngless to compute mOTU profiles.

This requires the use of the (standard) motus module:

```
ngless "0.0"
import "motus" version "0.1"
```

This module (with the motus database) will be downloaded the first time you use it.

You can use all the ngless functionality to load and preprocess your data:

```
input = paired('input.1.fq.gz', 'input.2.fq.gz')

preprocess(input, keep_singles=False) using |read|:
    read = substrim(read, min_quality=25)
    if len(read) < 45:
        discard
```

Producing the motus tables is done in three steps.

1. Map the samples against the `motus` reference (this reference comes with the motus module we imported earlier):

   ```
   mapped = map(as_reads(mapped), reference='motus', mode_all=True)
   ```

2. call the built-in `count` function to summarize your reads at gene level:

   ```
   counted = count(mapped, features=['gene'], multiple={dist1})
   ```

3. call the `motus` function, which takes the gene count table and performs the motus quantification. The result of this call is another table, which can then be written out with the standard `write` call:

   ```
   table = motus(counted)
   write(table, ofile='motus-counts.txt')
   ```

This function is the only special function introduced by the `motus` module, everything else is standard ngless.

You can see a full worked out example in the examples/motus.ngl recipe

Configuration

Note: ngless' results do not change because of configuration or command line options. **The ngless script always has complete information on what is computed**. What configuration options change are details of _how_ the results are computed such as where to store intermediate files and how many CPU cores to use.

Ngless gets its configuration options from the following sources:

1. Defaults/auto-configuration
2. A global configuration file
3. A user configuration file (typically `$HOME/.config/ngless.conf`)
4. A configuration file present in the current directory‘
5. A configuration file specified on the command line
6. Command line options

In case an option is specified more than once, the order above determines priority: later options take precedence.

Options

`jobs`: number of CPUs to use. You can use the keyword `auto` to attempt auto-detection (see below).

`temporary-directory`: where to keep temporary files. By default, this is the system defined temporary directory (either `/tmp` or the value of the `$TEMPDIR` environment variable on Unix).

`color`: whether to use color output. Defaults to `auto` (i.e., print color if the output is a terminal), `no` (never use color), `force` (use color even if writing to a file or pipe), `yes` (synonym of `force`).

`print-header`: whether to print ngless banner (version info...).

`user-directory`: user writable directory to cache downloads (default is system dependent, on Linux, typically `$HOME/.local/share/ngless/`.

`user-data-directory`: user writable directory to cache data (default is a `data` directory inside the `user-directory` [see above]).

`global-data-directory`: global data directory.

Debug options

`keep-temporary-files`: whether to keep temporary files after the end of the programme.

`trace` (only command line): print a lot of internal information.

Auto-detection of the number of CPUs

If the option `auto` is passed as the number of jobs (either on the command line or in the configuration file), ngless will inspect the environment looking for a small set of clues as to how many CPUs to use. In particular, it will make use of these variables:

- `OMP_NUM_THREADS`
- `NSLOTS`
- `LSB_DJOB_NUMPROC`
- `SLURM_CPUS_PER_TASK`

If none are found (or they do not contain a single number), an error is produced.

Frequently Asked Questions

This is a list of questions we have regularly gotten on the project. See below for questions about the ngless language.

Why a new domain-specific language instead of a library in Python (or another existing language)?

There are several advantages:

1. Fast error checking which can speed up the development process. For example, static type checking, which is known to many programmer. In general, we do a lot of error checking before even starting interpretation. We perform syntax and error checking, but we can also check some conditions that can be tricky to express with simple types only (e.g., certain parameter combinations can be illegal). We also pre-check all the input files (so even if you only use a particular file in step 5 of your process, we check if it exists even before running steps 1 through 4). We even do some things like: if you use step 1 to compute to name of the input file that will be used in step 5, we will check it immediately after step 1. Same for output files. If you issue a `write()` call using `output/results.txt` as your output filename, we will check if a directory named output exists and is writable. We also try to be helpful in the error messages (mispelled a parameter value? Here's an error, but also my best guess of what you meant + all legal values). I really care about error messages.
2. Another important advantage is that Haskell has a compiled implementation and performance matters. I use Python all the time and like the ecosystem a lot, but for some of the things that ngless does, it'd just be too slow (and if using PyPy, I'd miss out on numpy, so that's not very feasible either).
3. By controlling the environment more than would be typical with a Python library (or any other language), we can also get some reproducibility guarantees. Note too that we declare the version of every script so that we can update the interpreter in the future without silently changing the behaviour of older ones.
4. Using a domain specific language makes the resulting scripts very readable even for non-experts as there is little boilerplate.

Is the language extensible?

Yes.

While the basic types and syntax of the language are fixed, it is not hard to add external modules that introduce new functions. These can be described with a YAML file and can use any command line tool.

Add new model organisms can similarly be done with simple YAML file.

More advanced extensions can be done in Haskell, but this is considered a solution for advanced users.

Couldn't you just use Docker/Bioboxes?

Short answer: Bioboxes gets us part of the way there, but not all of it; however, if we think of these technologies as complements, we might get more out of them.

Longer answer:

Several of the goals of ngless can be fulfilled with a technology such as bioboxes. Namely, we can obtain reproducibility of computation, including across platforms using bioboxes without having to bother with ngless. However,

the result is less readable than an ngless script, which is another important goal of ngless. An ngless script can be easily be submitted as supplemented methods to a journal publication and even be easily scrutinized by a knowledgeable reviewer in an easier way than a Docker container.

Furthermore, the fact that we work with a smaller domain than a Docker-based solution (we only care about NGS) means that we can provide the users a better experience than is possible with a generic tool. In particular, when the user makes a mistake (and all users will make mistakes), we can diagnose it faster and provide a better error message than is possible to do with Bioboxes.

What is the relationship of ngless to the Common Workflow Language?

Like for the question above, we consider ngless to be related to but not overlapping with the CWL (Common Workflow Language).

In particular, much of functionality of ngless can also be accessed in CWL workflow, using our CWL wrappers.

They are also available via `pip install ngless_cwl`

How does ngless interact with job schedulers and HPC clusters?

Generally speaking, it does not. It can be used with HPC clusters, whereby you simply run a job that calls the ngless binary.

The parallel module can be used to split large jobs in many tasks, so that you can run multiple ngless instances and they collaborate. It is written such that does not depend on the HPC scheduler and can, thus, be used in any HPC system (or even, for smaller jobs, on a single machine).

Questions about the ngless language

Can I pass command line arguments to a script?

Yes, you can. Just add them as additional arguments and they will be available inside your script as `ARGV`.

What are symbols (in the ngless language)?

If you are familiar with the concept, you can think of them as `enums` in other languages.

Whenever a symbol is used in the argument to a function, this means that that function takes only one of a small number of possible symbols for that argument. This improves error checking and readibility.

Does the select function work on inserts (considering both mates) or per-read (treating the data as single-ended)?

By default, `select` considers the insert as a whole, but you can have it consider each read as single-end by using setting the `paired` argument to `False`.

Advanced options

Subsample mode

Subsample mode simply *throws away >90% of the data*. This allows you to quickly check whether your pipeline works as expected and the output files have the expected format. Subsample mode should never be used in production. To use it, pass the option `--subsample` on the command line:

```
ngless --subsample script.ngl
```

will run `script.ngl` in subsample mode, which will probably run much faster than the full pipeline, allowing to quickly spot any issues with your code. A 10 hour pipeline will finish in a few minutes (sometimes in just seconds) when run in subsample mode.

Note: subsample mode is also a way to make sure that all indices exist. Any `map()` calls will check that the necessary indices are present: if a `fafile` argument is used, then the index will be built if necessary; if a `reference` argument is used, then the necessary datasets are downloaded if they have not previously been obtained.

Subsample mode also changes all your `write()` so that the output files include the `subsample` extension. That is, a call such as:

```
write(output, ofile='results.txt')
```

will automatically get rewritten to:

```
write(output, ofile='results.txt.subsample')
```

This ensures that you do not confuse subsampled results with the real thing.

Language

This is a semi-formal definition of the NGLess language.

Basics

Script-style (# to EOL), C-style (/* to */) and C++-style (// to EOL) comments are all recognised.

Strings are denoted with single or double quotes and standard backslashed escapes apply (\n for newline, ...).

A symbol is denoted as a token surrounded by curly braces (e.g., `{symbol}` or `{gene}`).

Integers are specified as decimals `[0-9]+` or as hexadecimals `0x[0-9a-fA-F]+`.

Booleans are denoted as `true` or `false`.

Version declaration

The first line of an NGLess file should be a version declaration:

```
ngless "0.0"
```

Future versions of ngless will increase the string value. Also serves as a magic constant for other tools.

Module Import Statments

Following the version statement, it is possible to add module import statements, for example:

```
import "batch" version "1.0"
```

This statement specifies that the `batch` module, version `1.0` should be used in this script. Module versions are independent of ngless versions.

Use statements are not legal except as the first block.

Comments

Single line comments start with `#` or `//` and run to the end of the line:

```
i = 10 // Assign ten to variable 'i'
```

Multi-line comments start with `/*` and end with `*/`. As in C, they cannot be nested. For example:

```
/* This is a very long comment
 * It goes on and on..
 */
```

Data types

NGless supports the following basic types:

- String
- Integer
- Bool
- Symbol
- Filename
- Shortread
- Shortreadset
- Mappedread
- Mappedreadset

In addition, it supports the composite type List of X where X is a basic type. Lists are built with square brackets (e.g., [1,2,3]). All elements of a list must have the same data type.

String

A string can start with either a quote `(U+0022, ")` or a single quote `(U+0027,')` or and end with the same character. They can contain any number of characters.

Special sequences start with a ****. Standard backslashed escapes can be used as **LF** and `CR` (`\\n` and `\\r` respectively), quotation marks (`\\'`) or slash (`\\\`).

Integer

Integers are specified as decimals `[0-9]+` or as hexadecimals `0x[0-9a-fA-F]+`. Use `-` to specify a negative number.

Boolean

The two boolean constants are `True` and `False` (which can also be written `true` or `false`).

Symbol

A symbol is denoted as a token surrounded by curly braces (e.g.. `{symbol}` or `{drop}`). A symbol can be thought of as a nothing more than a string, but they are used for function arguments to indicate that there is only a limited set of allowed values for that argument.

Variables

NGless is a statically typed language and variables are typed. Types are automatically inferred from context.

Assignment is performed with = operator:

```
variable = value
```

A variable that is all upper-case is a constant and can only be assigned to once.

Operators

Unary

The operator **(-)** returns the symmetric of its integer argument.

The operator **len** returns the length of a ShortRead.

The operator `not` negates its boolean argument

Binary

All operators can only be applied to integers. The operators described are available:

```
+ - < > >= <= == !=
```

Indexing

Can be used to access only one element or a range of elements in a ShortRead. To access one element, is required an identifier followed by an expression between brackets. (e.g, x[10]).

To obtain a range, is required an identifier and two expressions separated by a ':' and between brackets. Example:

x[:]	returns from position 0 until length of variable x
x[10:]	returns from position 10 util length of variable x
x[:10]	returns from position 0 until 10

Conditionals

Conditionals work as in Python. For example:

```
if 5 > 10:
   val = 10
else:
   val = 20
```

Functions

Functions are called with parentheses:

```
result = f(arg, arg1=2)
```

Functions have a single positional parameter, all other must be given by name:

```
unique(reads, max_copies=2)
```

The exception is constructs which take a block: they take a single positional parameter and a block. The block is passed using the using keyword:

```
preprocess(reads) using |read|:
  block
  ...
```

There is no possibility of defining new functions. Only the built-in functions are available. The |*read*| syntax defines an unnamed (lambda) function, which takes a variable called *read*. The function body is the following block.

Blocks

Blocks are defined by indentation in multiples of 4 spaces. Tab characters are not allowed (so that nobody ever mixes tabs and spaces, ever).

Blocks are used for conditionals (if statements) and using statments.

Pure functions

The following functions are pure functions:

- unique
- substrim
- map
- count
- as_reads
- select

The result of calling a pure function **must** be assigned to a variable or an error is raised.

In the first version, there is no possibility of defining new functions. Only the builtin functions are available.

Auto-comprehension

A function of type `A -> * -> B` can be automatically used as `[A] -> * -> [B]`:

```
in1,in2 = fastq(["in1.fq", "in2.fq"])
```

This allows for a pipeline which runs in parallel over many input filenames.

Encodings/Tokenization

Tokenization follows the standard C-family rules. A word is anything that matches `[A-Za-z_]`. The language is case-sensitive. All files are UTF-8.

Both LF and CRLF are accepted as line endings (Unix-style LF is preferred).

A semicolon (;) can be used as an alternative to a new line. Any spaces (and only space characters) following a semicolon are ignored. *This feature is intended for inline scripts at the command line (passed with the ‘‘-e‘‘ option), its use for scripts is heavily discouraged and may trigger an error in the future.*