NGLess
A domain specific language for next generation sequence data analysis

Paulo Ricardo Ribeiro Monteiro

Thesis to obtain the Master of Science Degree in Information Systems and Computer Engineering

Supervisors: Prof. Ana Teresa Correia de Freitas
            Dr. Luís Pedro Coelho

Examination Committee
Chairperson:    Prof. Pedro Manuel Moreira Vaz Antunes de Sousa
Supervisor:     Prof. Ana Teresa Correia de Freitas
Members of the Committee: Prof. José Luís Brinquete Borbinha
                        Prof. David Manuel Martins de Matos

November 2014
Dedicated to my family, friends and girlfriend.
Acknowledgments

I would like to thank Prof. Ana Teresa Freitas, my supervisor and mentor, for taking time out from her, much, busy schedule to help me throughout the whole dissertation and for giving me opportunities and encouragement to continuously do a good job. I also appreciate the scholarships provided during the thesis.

I would like to express, as well, my gratitude to my co-supervisor, Dr. Luis Pedro Coelho, whose expertise, understanding, and most importantly patience, added considerably to my professional experience. His vast knowledge and skill in many different areas was essential to this dissertation.

Thanks also goes to Prof. José Borbinha, chairperson in the thesis project, as his feedback was really important, at that time, to tackle the dissertation in a different manner. I must also acknowledge Prof. David Matos, researcher at INESC-ID in the group Spoken Language System Lab (L2F), for providing key advices, in the final phase of the project, on the creation of the language specification.

I would like to thank the other members of my group, Knowledge Discovery and Bioinformatics (KDBIO), for the assistance they provided at all the meetings, reunions and debates throughout the dissertation. I feel fortunate to have had the opportunity to work in his group and participate in the exciting field of computational biology.

I would also like to thank my family for the support and inspiration they provided me through my entire life intending to encouraged me to pursue my interests. Last but not least, I would like to thank my girlfriend for the unconditional support.
Abstract

Nearly all bioinformatics tasks require interfacing and execution of many programs to process data. Computational analysis is, in many cases, the bottleneck for many scientific facilities as the time required to setup pipelines and to interpret the results reduces productivity. Only a small but growing number of laboratories around the world have the required expertise in both biology and computation.

Presently, most molecular biology laboratories need to deal with huge amounts of sequence data, making the development of more sophisticated computational tools an urgent problem to be solved. In this context, the development of tools with the ability to set up simple analysis pipelines plays an important role in bioinformatics. However, the interfaces between the existent tools are still a particularly acute problem due to the use of non-standards formats. Other important aspects of computational tools are its usability, scalability and robustness. Nonetheless, there are other problems inherent that also require our attention, such as data reproducibility.

To tackle all these problems we propose a domain specific language (DSL), named NGLess, that can be used to specify a series of operations leading to the analysis of next generation sequence data. Unlike Make-like tools, NGLess is able to detect semantic errors, make certain types of errors impossible, and, generally, allow for faster pipeline design.

Keywords: Domain Specific Language, Next Generation Sequence, Bioinformatics
Resumo

Grande parte das tarefas desempenhadas por bioinformáticos requer a execução e interacção com diversos programas para processar dados. A análise computacional é, em muitos casos, o bottleneck para muitas instalações científicas já que o excesso de tempo na configuração de pipelines e a respectiva interpretação dos resultados reduz, em demasia, a produtividade dos investigadores. Apenas um pequeno número, mas crescente, de laboratórios no mundo tem os conhecimentos necessários em biologia e computação, em simultâneo.

Atualmente, a maioria dos laboratórios de biologia molecular lida com enormes quantidades de dados de sequenciamento, fazendo com que o desenvolvimento de ferramentas computacionais mais sofisticadas seja um problema urgente a ser resolvido. Neste contexto, a criação de ferramentas que permitam o desenvolvimento, de forma simples, de pipelines de análise, desempenha um papel importante na bioinformática. Porém, devido ao uso de formatos não normalizados a comunicação entre as ferramentas existentes sofre de graves problemas. Outros aspectos importantes em ferramentas computacionais são a sua usabilidade, escalabilidade e robustez. No entanto, existem outros problemas inerentes que também exigem a nossa atenção, como a reprodutibilidade de dados.

Para combater todos estes problemas, propomos uma linguagem de domínio específico (DSL), chamada NGLess, que pode ser usada para especificar uma série de operações para a análise de dados de sequências de nova geração (NGS). Ao contrário de ferramentas Make-like, NGLess é capaz de detectar erros semânticos, tornar certos tipos de erros impossíveis, e, geralmente, permitir um desenvolvimento, mais rápido, de pipelines.

Palavras-chave: Linguagem de domínio específico, Bioinformática, Sequenciamento de nova geração
Contents

Acknowledgments ................................................................. iii
Abstract ........................................................................ iv
Resumo ........................................................................... v
List of Tables ................................................................... ix
List of Figures ................................................................... x
Glossary ........................................................................... xi

1 Introduction ....................................................................... 1
  1.1 Motivation .................................................................. 2
  1.2 Domain Specific Languages ........................................ 2
    1.2.1 Definition ......................................................... 3
    1.2.2 Design ............................................................. 4
    1.2.3 Implementation ................................................ 4
    1.2.4 Trade-offs ......................................................... 5
  1.3 Next Generation Sequence ........................................... 7
    1.3.1 Input Data ........................................................ 7
    1.3.2 NGS Operations ............................................... 9
  1.4 NGS data analysis methodologies .................................. 12
    1.4.1 NGS Tools ....................................................... 12
    1.4.2 NGS Analysis .................................................. 13

2 NGLess definition and implementation ............................... 27
  2.1 NGLess script example ............................................... 28
    2.1.1 Highlights ....................................................... 29
  2.2 Key algorithms ........................................................ 31
    2.2.1 Unique ............................................................ 31
    2.2.2 Annotate .......................................................... 33
  2.3 Interpreter Structure .................................................. 34
    2.3.1 Lexer ............................................................... 35
    2.3.2 Parser .............................................................. 35
List of Tables

1.1 The range values for the three possible FastQ variants ........................................ 8
1.2 SAM Format ............................................................................................................. 10
1.3 Mandatory fields in the GFF format ....................................................................... 11

2.1 Operation costs for data structure Interval Tree ..................................................... 33

3.1 Mean and standard deviation for the fastq function execution time ....................... 45
3.2 Minimum and maximum (small) file sizes ............................................................. 46
3.3 Mean and standard deviation for the unique function execution time ..................... 46
3.4 Mean and standard deviation for the preprocess function execution time ............... 47
3.5 Memory usage for the preprocess function ............................................................ 48
3.6 Mean and standard deviation for the annotate function execution time ................. 48
3.7 Annotation speedup of NGLess against HTSEQ-count .......................................... 49
3.8 Tarball, organism, sizes with and without preprocess ............................................. 50
3.9 Mean and standard deviation for the map function execution time ....................... 51
3.10 Speed up of map operation .................................................................................... 51
List of Figures

1.1 Common NGS data analysis pipeline .............................................. 3
1.2 Evolution of sequencing cost ........................................................... 7
1.3 Taverna Workbench overview .......................................................... 16
1.4 Galaxy interface environment .......................................................... 17
1.5 Kepler Workflow and Subworkflow ................................................... 18
1.6 Cloudburst Algorithm ...................................................................... 24
1.7 The Myrna Pipeline ......................................................................... 25

2.1 Unique algorithm ............................................................................ 32
2.2 Overlap modes in annotation ............................................................ 34
2.3 NGLess interpreter structure .............................................................. 35
2.4 Home of NGLess visualising page ..................................................... 38
2.5 Script flow from interpreter to browser ............................................. 39
2.6 Statistics for a sample file ................................................................. 39
2.7 Top 10 genes for file ‘CountResults’ ................................................ 40
2.8 Script creation options .................................................................... 40
2.9 Code coverage of NGLess ................................................................. 43

3.1 Memory usage of unique function .................................................... 46
3.2 Memory usage for the function annotate .......................................... 49
3.3 BWA Index calculation vs transmission ............................................. 50
3.4 Functions percentage on total execution time .................................... 51
## Glossary

<table>
<thead>
<tr>
<th>Acronym</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>DSL</strong></td>
<td>A domain specific language is a programming language specialised in a particular domain.</td>
</tr>
<tr>
<td><strong>FastQ</strong></td>
<td>Is a format for storing both a biological sequence and its corresponding quality scores.</td>
</tr>
<tr>
<td><strong>GFF</strong></td>
<td>General feature format describes genes and other features associated with DNA, RNA and Protein sequences.</td>
</tr>
<tr>
<td><strong>GPL</strong></td>
<td>A general purpose language is a programming language to develop software in a wide variety of application domains.</td>
</tr>
<tr>
<td><strong>NGS</strong></td>
<td>Next generation sequence is a term used to describe a number of different modern sequence technologies.</td>
</tr>
<tr>
<td><strong>QC</strong></td>
<td>Quality control is the process of ensuring a good quality dataset.</td>
</tr>
<tr>
<td><strong>SAM</strong></td>
<td>Sequence Alignment/Map format is a generic format to store large sequence alignments.</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

On a day to day basis, bioinformaticians require interfacing and executing many programs to process data. Computational analysis is, generally, a bottleneck for many laboratories and scientific facilities [13]. In contrast to what one may think, it is not computation time, but the setting up of pipelines and the interpretation of the results that takes most time. This happens because only a small, but growing number of laboratories around the world have the required expertise in both biology and computation. Other groups may be lucky enough to have a biologically savvy ‘computational expert’ at hand to delegate such tasks to [4]. So setting up simple analysis pipelines has become an important task that must be possible to be performed by any one that works with biological data.

The task of putting together different programs and tools in order to create a workflow for data analysis is still a time consuming task. This problem is particularly acute when we look at the time that is needed to create parsers, due to the use of non-standard formats.

With the current commercialisation of various affordable desktop sequencers, next generation sequence (NGS) is being made available to more and more biologists. Consequently, NGS data analysis can happen anywhere, e.g. in a laboratory or even at home. NGS data analysis has many different applications, and so it is impossible to define a strict pipeline process as it depends on the context.

Currently there are multiple ways to define such pipelines and they will be described in Section 1.4, as well as their advantages and disadvantages.

It is very important in bioinformatics, as in other areas, to have applications that have good usability, scalability and robustness. But there are other inherent problems that also require our attention, such as data reproducibility [41], either by the researcher or by some third-party, enabling the verification of data.

To solve this problem we propose a solution based on a domain specific language (DSL), named NGLess, where the user can specify a series of operations leading to the desired result. Unlike Make-like tools, NGLess is able to detect semantic errors, makes certain types of errors
impossible, and generally allows for faster pipeline design than is currently possible.

At this chapter we will start by describing the motivation of the work (Section 1.1), then give an overview both at DSL (Section 1.2) and NGS data analysis (Section 1.3.2) domains. Afterwards, we describe the state of the art of NGS data analysis methodologies (Section 1.4) which contains the current, available, solutions to create pipelines for NGS data analysis.

1.1 Motivation

The complete NGS data analysis process is complex, includes multiple analysis steps, is dependent on a multitude of programs and databases and involves handling large amounts of heterogeneous data. Due to the enormous success of NGS projects, a flood of tools has been created to support specific parts of the analysis workflow. It is clear that the appropriate choice of tools is a non-trivial task, especially for inexperienced users [38].

Most NGS applications rely on sequence alignment as the first analysis step [35]. Before the alignment researchers require some kind of pre-processing of data, that is always dependent on the researcher interest. Figure 1.1, represents a generic NGS analysis pipeline, based on a survey created by Pabinger et al. [38] for variant analysis.

The main goal of our work is to create a tool that enables the creation of a pipeline of work for all first phase of NGS data analysis until the point (inclusive) of annotation. We want to do this while achieving the following objectives:

- Improve the development of NGS data analysis tools;
- Facilitate the configuration and execution of pipelines;
- Enable data analysis reproducibility;
- Process a dataset of any size.

We also want to provide the user with an easy to use web application that enables the result visualisation and the creation of scripts using an editor or a wizard.

1.2 Domain Specific Languages

In engineering, approaches that are generic can be distinguished from those that are specific. A general approach provides a generic solution for many problems but it might not be optimal. As for specific approaches, they provide a much better solution for a smaller set of problems.

Domain-specific languages have been in use for decades. In Unix there is a tradition of such small languages [11] including sh and bash (for shell scripting), lex and yacc (for lexical analysis
and parsing), and make (for automated software builds). These are external DSL systems, that typically use Unix built-in tools to help with translation.

Older programming languages as Cobol, Fortran and Lisp appeared as domain languages to solve problems in certain areas (business processing, numeric computation, and symbolic processing). These programming languages have gradually grown into general purpose solutions and the need, for support, to solve specialised domains re-emerged.

A DSL offers possibilities for analysis, verification, optimisation, parallelization, and transformation in terms of DSL constructs, that would be much harder or unfeasible if a general purpose language (GPL) had been used. Most of the time the GPL source code patterns involved are too complex or not well defined \[34\].

1.2.1 Definition

A domain specific language (DSL) is defined by Deursen et al. [18] as:

Definition 1.2.1 (Domain Specific Language). Is a programming language or executable specification language that offers, through appropriate notations and abstractions, expressive power focused on, and usually restricted to, a particular problem domain.
In a shorter definition, a domain specific language is a computer programming language of limited expressiveness and specific to a given domain.

1.2.2 Design

Deursen et al. suggests that a typical life cycle for a DSL is:

Analysis

1. Identify the problem domain
2. Gather all relevant knowledge in this domain
3. Cluster this knowledge in a handful of semantic notions and operations on them
4. Design a DSL that concisely describes applications in the domain.

Implementation

5. Construct a library that implements the semantic notions
6. Design and implement a compiler that translates DSL programs to a sequence of library calls
7. Provide tools to develop programs in the DSL (e.g., editors and wizards).

Use

8. Write DSL programs for all desired applications and compile them.

The analysis step, from (1) to (4), allows for a better understanding of the application domain. To acquire such an understanding, researchers in the domain area must provide guidelines. The implementation step, (5) and (6), can be carried out using several approaches that are defined in section 1.2.3.

This master thesis addressed the whole 8 steps, but the core work was the implementation and use.

1.2.3 Implementation

Different approaches can be taken to create a DSL, depending on the purpose of the tool. As Fowler defines, domain specific languages can be divided into three main categories [21]:

External DSL is a language that is separate from the main language used for the development. Usually, a custom syntax is developed for this new language, but is based on some other existent language syntax. A script, written in the external DSL, is parsed by code in the host using text parsing techniques. Examples of external DSLs are SQL and Awk.
**Internal DSL** is a particular way of using a general-purpose language. A script, in an internal DSL, is valid code for the general-purpose language that the DSL is developed in. It will only use a small subset of all the available features to handle one small aspect. Many Ruby libraries, in particular the framework Rails, is often seen as a collection of DSLs.

**Language Workbench** is a specialised integrated development environment (IDE) for defining and building DSLs. It can either be used either to determine a given DSL structure or as a editing environment to write DSLs.

It is important to mention that there is **not** a best approach. One of the possible options may apply better for different problems, even at the same domain context. It is not easy to choose the best path to follow, but should be highly thought through as it completely changes the DSL structure. In this work an **external approach** was used.

### 1.2.4 Trade-offs

There are many advantages and disadvantages when adopting a DSL approach to a given domain. It is important to consider if the opportunities outweigh the risks for a given context. The advantages of domain specific languages have been described by [Mernik et al., Deursen et al., Spinellis] at [34, 18, 47] as:

- **Development productivity.** DSL increases productivity as it can, more clearly, communicate the intent of a system. The language is expressed at the idiom of the domain experts and so they can: understand, validate, modify, and often even develop DSL programs.
- **Concise, domain specific notations.** DSL programs are concise, self-documenting to a large extent, and can be reused for different purposes.
- **Analysis and verification.** The easier it is to read the code, the easier it will also be to detect and fix programming errors.
- **Platform independence.** DSLs can abstract from a particular platform.
- **Abstraction.** The hassle of dealing with technical details (e.g., non-standard formats) is minimised by providing a language that allows to work at a high abstraction level.
- **Domain-specific tooling support.** Due to the limited scope of the DSL, there is an opportunity to offer a better tooling as part of an integrated development environment (IDE).

As mentioned before, there are also disadvantages when using a DSL. One of the biggest disadvantages of a DSL is what Fowler [21] calls the **cacophony problem.** It is the concern that languages are hard to learn, so using many languages will be much more complicated than using a single one. Multiple languages makes it harder for new people to join new projects as the
A key aspect, in domain specific languages, is the expressiveness of the code, as it allows a bioinformatician to improve his productivity by developing pipelines in a much simpler way.

Many of the pipelines created for NGS data analysis may take many days to complete and an error may occur a long time after the beginning of the pipeline resulting in many lost hours. When using a DSL, it is possible to have a much richer type validator, oriented to NGS data analysis, that identifies errors before the interpretation.

By using a DSL it is possible to easily reproduce an analysis by simply sharing a DSL script. This allows the scientific community to share the pipelines (e.g., scientific papers) that lead to their results.

We believe that the advantages of using a DSL for NGS data analysis outweighs the disadvantages.
1.3 Next Generation Sequence

Between 1990 and the publication of a working draft in 2001, more than 200 scientists joined forces in a 3 billion dollar effort to read the roughly 3 billion bases of DNA that comprise the human genome. The cost was very high and it was of key importance to develop a new method of sequencing that was both cheaper and faster.

This demand has lead to the development of next generation sequence (NGS) platforms. In the past decade, several NGS platforms have been developed to provide low-cost and high-throughput sequencing. There are three main companies that built machines that allow NGS analysis:

- Illumina
- Applied Biosystems
- Roche

In the past few years genome sequencing technologies have exceeded Moore’s Law and the speed of genome sequencing has more than doubled every two years since 2003 due to NGS platforms, see Figure 1.2 [14].

![Figure 1.2: Evolution of sequencing cost](image)

1.3.1 Input Data

All companies follow the standard defined for FastQ, where each read follows the format:
Typically, a FastQ file has from 1 to 10GB compressed, and each step in the analysis generates more reads, resulting on a huge amount of Disk/Ram required. The meaning of each line of the read and the possible values it can contain are:

1. **Title line** often holds just an identifier. This is a free format field with no length limit.

2. **Sequence line** is a string of text written in an alphabet of 4 characters and representing a biological sequence of DNA. The characters in this type of sequence are named base pair (BP), and can be one of the four bases Adenine (A), Guanine (G), Cytosine (C), and Thymine (T), sometimes also called nucleotides.

3. **Delimiter** the character + signals the end of the sequence line and the start of the quality string. Originally this also includes a full repeat of the title line text.

4. **Quality line** encodes the quality values for the sequence. Different encodings might be used, see Table 1.1.

The quality line must be equal in length to the sequence line.

<table>
<thead>
<tr>
<th>Description</th>
<th>ASCII characters</th>
<th>Quality score</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sanger standard</td>
<td>33–126</td>
<td>0 to 93</td>
</tr>
<tr>
<td>Solexa / early Illumina</td>
<td>59–126</td>
<td>-5 to 62</td>
</tr>
<tr>
<td>Illumina 1.3+</td>
<td>64–126</td>
<td>0 to 62</td>
</tr>
</tbody>
</table>

Table 1.1: The range values for the three possible FastQ variants

An example of a FastQ file with a single read looks like the following (abbreviated in the sequence and quality field):

```
@SRR014849.1 EIXKN4201CFU84 length=93
GGGGGGGGGGGGGGCTTTTTGGTTTGGAACCGAAAGG
+SRR014849.1 EIXKN4201CFU84 length=93
3+&$#""""""""7F@71,‘’;C?,B;?6B;:EA1EA
```

**Quality uncertainty**

There are two different equations, as Cock et al. mentions [16], that can be used to calculate the quality of each nucleotide. The difference in both equations is the encoding of the odds \( \frac{p}{1-p} \) versus \( p \) (where \( p \) is the estimated error probability for a base pair), but they generate similar results differing only at low quality values.
Also, different sequencers may encode the quality sequence using different ASCII range values. These possible ranges can be consulted in Table 1.1.

It is important to know the exact encoding used to generate the FastQ file, as different encodings lead to different results.

1.3.2 NGS Operations

Before proceeding to the currently existing tools, it is important to explain the most commonly used operations in NGS analysis, see Figure 1.1.

Quality Control

The quality of the data is affected by several factors regardless of the NGS platform. Quality control (QC) is an essential step to ensure that the data used for analysis is not replete with low quality sequences that can lead to incorrect conclusions [44]. The most common way of performing QC is by looking at summary statistics of the data. There are different programs that can produce those statistics and the results can be seen through a web application.

Pre-process

Pre-processing is a key operation known in many other areas as “garbage in, garbage out”, where the idea is to filter everything that could possibly generate bad results. Normally, pre-process is applied after QC. A different pre-processing approach can be applied to the same data and is completely dependent on what the user finds of key importance.

There are two key operations that are the most commonly used when performing pre-processing.

Duplicate Removal

Sequence replication can occur during different steps of the sequencing protocol and can therefore generate artificial duplicates[24]. A common approach is to remove duplicate occurrences of a given read. Another is to keep at most N copies.

Trim

There are two types of trimming: static and based on quality.

Static trimming is the process of reducing sequences to a specific length by removing, at least one of, the edges of a read.

Trimming based on quality is the process of reducing the quality sequence to the biggest sub-sequence with a minimum quality.
Alignment

When two DNA sequences are arranged so that their most similar base pairs (BP) are overlapped, they are said to be aligned. The overlaps do not have to be exact and, for each BP in a alignment it can either be a match (if both BP are equal), a mismatch (if BPs are different) or a gap (if BPs from one sequence is aligned with spaces, representing the insertion or deletions of portions of DNA from one of the sequencers).

Many bioinformatics tasks depend upon successful alignments. The SAM (Sequence Alignment/Map) file format, which can be optimised to BAM (Binary Alignment/Map), is the standard defined to describe sequence alignments. All alignment algorithms use this format as the output format.

SAM Format

The SAM Format allows storing alignment data in a series of tab delimited columns. It is the standard format to store alignment results.

This standard is divided in two sections, the header and the alignment. The first contains information about the entire file and additional alignment information. The latter contains the information for each sequence about the alignment.

Each line in the alignment section has 11 mandatory fields. Table 1.2 shows each field and its type.

<table>
<thead>
<tr>
<th>Column</th>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>QNAME</td>
<td>Query NAME of the read or the read pair</td>
</tr>
<tr>
<td>2</td>
<td>FLAG</td>
<td>Bitwise FLAG (pairing, strand, mate strand, etc.)</td>
</tr>
<tr>
<td>3</td>
<td>RNAME</td>
<td>Reference sequence NAME</td>
</tr>
<tr>
<td>4</td>
<td>POS</td>
<td>1-Based leftmost POSition of clipped alignment</td>
</tr>
<tr>
<td>5</td>
<td>MAPQ</td>
<td>MAPping Quality (Phred-scaled)</td>
</tr>
<tr>
<td>6</td>
<td>CIGAR</td>
<td>Extended CIGAR string (operations: MIDNSHP)</td>
</tr>
<tr>
<td>7</td>
<td>RNEXT</td>
<td>Mate Reference NaMe (= if same as RNAME)</td>
</tr>
<tr>
<td>8</td>
<td>PNEXT</td>
<td>1-Based leftmost Mate POSition</td>
</tr>
<tr>
<td>9</td>
<td>TLEN</td>
<td>Inferred Insert SIZE</td>
</tr>
<tr>
<td>10</td>
<td>SEQ</td>
<td>Query SEQuence on the same strand as the reference</td>
</tr>
<tr>
<td>11</td>
<td>QUAL</td>
<td>Query QUALity (ASCII-33=Phred base quality)</td>
</tr>
</tbody>
</table>

Table 1.2: Mandatory fields in the SAM format [32].

We would like to highlight the importance of the CIGAR field. The CIGAR string defines the operations made to the sequence to perform the alignment. The standard CIGAR defines three operations: ‘M’ for match/mismatch, ‘I’ for insertion compared with the reference and ‘D’ for deletion. The extended CIGAR proposed in SAM added four more operations: ‘N’ for skipped bases on the reference, ‘S’ for soft clipping, ‘H’ for hard clipping and ‘P’ for padding[32]. CIGAR is what allows to calculate the size of the alignment.
Annotation

An annotation, independently of the context, is the process of adding a note by way of explanation or commentary. Genome annotation is the process of identifying the locations of genes and all of the coding regions in a genome and determining what those genes do. Once the sequences are aligned (Section 1.3.2), annotation can be performed using a General Feature Format (GFF) file.

There are many publicly accessible databases that contain those GFF files as Ensembl\(^1\), NCBI\(^2\) and UCSC\(^3\).

GFF Format

General Feature Format (GFF) is a format for describing genomic features. GFF files must have nine columns, are tab-delimited and are plain text files. It allows for the development of features and have them tested without having to maintain a complete feature finding system.

<table>
<thead>
<tr>
<th>Column</th>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>SEQID</td>
<td>The ID of the landmark.</td>
</tr>
<tr>
<td>2</td>
<td>SOURCE</td>
<td>Algorithm or operating procedure.</td>
</tr>
<tr>
<td>3</td>
<td>TYPE</td>
<td>The type of the feature.</td>
</tr>
<tr>
<td>4</td>
<td>START</td>
<td>The start position relative landmark given in column 1.</td>
</tr>
<tr>
<td>5</td>
<td>END</td>
<td>The end position relative landmark given in column 1.</td>
</tr>
<tr>
<td>6</td>
<td>SCORE</td>
<td>The score of the feature.</td>
</tr>
<tr>
<td>7</td>
<td>STRAND</td>
<td>+ for positive, - for minus, and . for unstranded.</td>
</tr>
<tr>
<td>8</td>
<td>PHASE</td>
<td>Feature start with reference to the reading frame.</td>
</tr>
<tr>
<td>9</td>
<td>ATTRIBUTES</td>
<td>List of features in format tag=value</td>
</tr>
</tbody>
</table>

Table 1.3: Mandatory fields in the GFF format [19].

The last field, attributes, is a list of features that are separated by a semicolon. It is, without a doubt, the most ambiguous field as there is no clear standard in how to present the tags. There are tags that have a predefined meaning but most of the time they are not used when they should. For example, the tag gene_id sometimes is present to represent the same information that the ID tag is meant to.

\(^1\)http://www.ensembl.org/index.html  
\(^3\)http://www.genome.ucsc.edu
1.4 NGS data analysis methodologies

This section provides a review of the work that has already been done in the area of next-generation sequence (NGS) data analysis. We will start by giving an overview on the existent NGS tools and how can one develop new tools (Section 1.4.1) and then describe systems that allow to create pipelines for NGS analysis (Section 1.4.2).

1.4.1 NGS Tools

There are currently many different NGS tools that are commonly used in a normal NGS data analysis pipeline, such as:

- **FastQC**, PRINSEQ[44], NGS QC[39] to generate statistics about FastQ files;
- **Samtools[32], BWA[31], BLAST[7]** to align sequences against a genome;
- **HTSEQ[8]** to annotate a data set;
- **IGV[40]** to visualise genomic datasets.

To ease the development of new NGS tools a programming language, named Bellman’s GAP, was designed.

Bellman’s GAP is a new programming system, designed to ease the development of bioinformatics tools based on the dynamic programming technique[42]. It supports a specific programming method, using dynamic programming over the data, but not a particular application domain, and so domain specific knowledge is provided by the user.

Bellman’s GAP is divided into GAP-L that allows the description of a Dynamic Programming algorithm in a abstract way and GAP-C that automatically generates a efficient C++ implementation of what is specified in GAP-L. So, GAP-L allows the re-use and extension of Dynamic program algorithms.

The fact that Bellman’s GAP does not support a specific domain, allows them to have a language with high abstraction. Thus, the input could be a biological sequence as well as a sequence of matrix dimensions, an arithmetic formula or a logical clause (3SAT-problem) [42].

For an example on an alignment GAP-L signature construct, see Listing 1.

After the signature is created, multiple implementations of the signature can be made for totally different domains. In which GAP-C will generate C++ code that can be compiled. For example could be created different implementations of the Levenshtein distance where each have different penalty costs for the repetition, deletion and insertion.

\[^{4}\text{http://www.bioinformatics.babraham.ac.uk/projects/fastqc}\]
1.4.2 NGS Analysis

There are, currently, many different ways to create a pipeline to perform NGS data analysis:

- NGS analysis using Linux/Unix tools
- NGS analysis using Workflow Engines systems
- NGS analysis using Libraries
- NGS analysis using Hadoop/MapReduce/HBase frameworks

In the following sections, we provide an overview on the most commonly used tools and also a critical review on each of approach.

**Linux/Unix tools**

This is the most popular approach to perform NGS analysis, due to the fact that it requires no setup time, only an UNIX/Linux environment.

In order to perform NGS analysis using UNIX/Linux tools, a researcher must know a few things about the Linux command-line tools, *e.g.* grep, awk. Also, one is required to have some expertise in some programming language in order to generate results and be familiar with the available NGS tools.

A normal NGS analysis will result in multiple invocations in the Linux terminal. The researcher normally creates a shell script or Makefile\(^5\) to manage the used invocations and to allow analysis reproducibility.

**Make**

Make is a build automation tool that allows to automate a wide variety of tasks that are normally done repeatedly. It allows to run a pipeline of NGS analysis without having to worry about making anything manually.

\(^5\)https://www.gnu.org/software/make/
Make searches the current directory for the makefile to use and then runs the default or specified target from that file. A target can be the name of a file that is generated by a program or the name of an action to carry out, such as clean.

Makefiles contain five kinds of things: explicit rules, implicit rules, variable definitions, directives, and comments [3, Chapter 16].

**Explicit rule** details when and how to remake files.

**Implicit rule** says when and how to remake a group of files based on their names.

**Variable definition** specifies a text value for a variable.

**Directive** allows to perform something special while reading the makefile. (e.g., reading another makefile)

**Comment** is a character # at the beginning of a line.

**Disadvantages of this approach**

Even though this kind of approach eases the required time to setup a pipeline to allow NGS analysis, it comes with some disadvantages.

**Low usability** The Unix system can be hard to use, due to the fact that it requires a lot of programmatic expertise. As Bodi[13] mentions “Researchers unfamiliar with the Unix command line may be unable to use these tools, or face a steep learning curve in trying to do so”.

This approach has a lot of different programs that one can use, making data analysis harder. Also, in most of the cases, learning a programming language is required and researchers might end up taking a few weeks learning a language, sometimes just to do simple tasks with the data.

**Type Validation** The biggest problem is that if a researcher makes a simple mistake as a wrong variable name in the script, one might lose hours of work due to the fact that an error will only be returned when that exact part of the script is executed.

This is a big disadvantage, because the execution of a NGS analysis pipeline might take several hours, and in case of an error the pipeline execution has to start over.

**Data reproducibility** To achieve data reproducibility using a Shell Script or Makefile, one needs to store every single operation. Problems can arise from the number of programs needed to be created and from the management of program versions.

The only way to achieve a perfectly reproducible script is to manually annotate all versions of all programs in use.

To improve some of the disadvantages, workflow engine systems were created to allow an interface for researchers with less programmatic expertise.
Workflow engine systems

Workflow engines systems (WES) are the most used tools in major research centers that deal with NGS analysis. With the use of interfaces, a workflow of processes is easy to create, but the systems are not easy to setup. The following sub-sections will describe the most used workflow engine systems on bioinformatics to do NGS analysis and at the end will describe disadvantages of this kind of solution.

Taverna

Taverna is a workflow engine developed (in Java) by “The myGrid team”\(^6\), and can be used as a:

- Command line;
- Workbench \(^7\) which allows one to create/edit/run workflows on his computer (Figure 1.3).

Before Taverna appeared, integration of tools and databases available on the web could only be made in two possible manners, either by using "copy-and-paste" or using "screen-scraping" on web pages using script languages like PERL [26].

Taverna made easy the integration of the growing number of available web services and made possible a central point of access to web services. By consequence, Taverna permits easily the creation of workflows using different services.

It comes with a wide range of services available that can be used to create workflows. To add extensibility, one can add external tools either by using scripts as Java Beanshell (requiring an API Consumer service to generate the service) or via external plug-ins.

Also, Taverna has a wide range of repositories configured by default but with the growing amount of new services available online, finding the correct one is hard. A project called BioCatalogue [12] has been created to provide a mechanism to register, browse and annotate Web Services.

To allow people to share and discover Taverna Workflows, "The myGrid team" created a project called myExperiment[17] that is a social network site that allows collaborations between researchers. As De Roure et al. mentioned [17] this is achievable by allowing workflows to be pooled and shared, traded and reused, within communities and across communities.

Taverna allows the construction of workflows to perform a range of different analyses, even ones without any correlation with biology, and is used in many different areas of expertise.

The Service executions in Taverna are implicitly parallel\(^8\), since they only depend on the input and as soon as the input(s) a service require(s) is ready it can start. Figure 1.3 presents, getJpegFromAnnotatedImage and Parse Moby Data SimpleAnnotatedJPEGImage that can start

\(^6\)http://www.mygrid.org.uk/
\(^7\)http://www.taverna.org.uk/download/workbench/
\(^8\)http://taverna.knowledgeblog.org/2010/12/13/parallel-service-invocations/
at the same time, due to the fact that they depend on the same input. Taverna also allows parallelization (specified by the user) of a specific service execution. This last task has some disadvantages since:

- It forces a user to know about data dependency, concurrency, and synchronisation to use a control link called "run after", which allows to run only a specific service after another.

- Sometimes a service is not concurrency-safe which forces a invocation to be made separately, for instance, because both calls attempts to write to the same local file [17].

**Galaxy**

Galaxy [5] is an open-source software package, implemented in the Python language, that can be easily deployed on a existing running system. It allows for rapid analysis through a web-based interface and is also freely available.

The main feature of galaxy is accessibility. This is achieved by allowing users without any programming expertise to use the system through a web-interface. This web-interface can be run locally or in a free public server. This enable users to use domain specific tools without any prior knowledge about the implementation or invocation.

Another important feature of Galaxy is the simplicity of extension. To achieve this, galaxy allows a user to simply add a new domain specific tool by changing an XML configuration file in the directory of workflow system. The tool can be either created by the user or a existent domain specific tool developed by a third-party company. Any executable that could be used in a command line invocation, can be added to the Galaxy workflow. Figure 1.4 [5] shows an example of how to add a tool to Galaxy.
Therefore, workflows are built from tools and the data is sent through those tools in a ordered manner, depending on the workflow created by the user, creating an architecture, conceptually, similar to pipe-and-filters.

With this WES, it is possible to share individual datasets, parts of analysis or even the conclusions of the analysis. One can do this through the share of a web link to a single user, or by publishing it, allowing all users to access it. Galaxy supports reproducibility of the results, by tracking derivation of user’s analyses, ensuring that every detail of every step is recorded and can be later inspected/reproduced.

To conclude, Galaxy works as a interface for Unix tools. It comes with some executables by default but more can be added as a researcher wishes, allowing for a researcher to know little to nothing about the Unix system.

**Kepler**

Kepler [6] gives scientists a powerful and easy-to-use system to create Scientific Workflows (SWFs). It allows to operate on data stored in a big variety of formats and is effective in integrating software components.

Kepler introduced a mechanism that allows to collapse a (sub)workflow into an abstract component (called actor) as long as the (sub) workflows have a well defined input port and output port, in which case can be collapsed and an actor can be composed. As can be seen in Figure 1.5, since the box named CLASSIFIER has well-defined input and output ports, it is called an actor.
and consequently a sub workflow. **Actors** are connected via channels and there are many possible execution semantics that could be assigned to the given workflow, in which they might have their own thread of control or be triggered by new inputs. Kepler comes with some predefined **actors** and there is a Kepler module called “bioKepler” which extends Kepler to execute a set of bioinformatics tools.

![Figure 1.5: Kepler Workflow and Subworkflow (also known as actor)](image)

Execution semantics is defined using an object called **director**. This object allows for the definition of how to execute the **actors** and how they interact with each other. In sum, **actors** define what will be the processing and **directors** define when it will occur.

This workflow engine system contains a graphical user interface, in which users simply select and connect components in order to create workflows. It is also possible to save, reuse and share workflows and customised components with colleagues using the Kepler archive format (KAR).

To obtain a parallel version of workflow components, it is required of the user to choose the correct **director**, e.g., PN Director for parallelism or SDF Director for serial [2, p. 47], and so it forces the user to understand how it works and lose too much time with concepts that is not (in most cases) familiar with neither has the interest to learn.

**Disadvantages with current WES**

The disadvantages of the workflow engine systems on NGS data analysis are quite similar in the different existing approaches. The biggest disadvantage of the WES is the fact that they are too generic, therefore there is no biological context. For example, they can be used either to make a “Sequence alignment” or "Calculate the monthly income" or even both. The following topics are also concerning disadvantages of this approach:
• Complexity to setup a pipeline of work;
• Type validation;
• Error Detection;

A detailed explanation on why these topics can be problems in a WES is presented below.

**Complexity to setup a pipeline of work**  Too much time is lost on the configuration of complex workflow engines to achieve, in most of the cases, a simple result and this reduces the productivity of NGS developers. Sometimes it is not only a matter of how fast the program computes a given dataset, but the simplicity on which they can design and deploy new analysis.

**Type validation**  Type validation should be expressive enough to allow an easy identification of a error in a given script/workflow and most importantly *as soon as possible*, preferable before any interpretation is performed.

Type validations in WES occur only at a high level. This means that this systems will not be able to validate specific aspects related with, for example, NGS data analysis and report them in a readable way.

**Error Detection**  WES are generic systems as they only rely on data coming from one place and redirecting it to another.

In these systems the workflow will be executed until an abort or the end. The problem is that the error may occur in some previous process of the workflow and the user will not know what caused it. Also, the user will think that the pipeline is executing, just to find out that some tool already had some error after a few hours.

**Libraries/Packages**

The use of libraries/packages are another option that users have nowadays to tackle NGS data analyses. They can be of easy use if one is already familiar with that language which incurs in high performance to develop new tools to process data. The next two sections will talk about the most used packages for NGS data analysis and also a recent library that has a different approach. After that, some limitations of these approaches.

**Bioconductor**

Bioconductor uses the R statistical programming language, and is open source and open development. It is an initiative for the collaborative creation of extensible software for computational biology and bioinformatics [22].
It currently has 749 packages that provide tools for the analysis of biologic data. The number of packages has increased in the last few years. Consequently means more functions that experienced users can use to improve productivity.

The packages in Bioconductor are not parallel but it provides packages such as *snow* or *rpvm* (which is now removed) so that the user can parallelize the code in a cluster. The snow package provides a higher level of abstraction that is independent of the communication technology, such as the message-passing interface (MPI) [20]. Every time users want to make a simple program, they are required to use functions in the snow package, which is not so trivial and is a downside to productivity.

**Biopython**

Biopython is a set of freely available tools for biological computation written in Python [15]. It provides Python with libraries that allow one to make NGS data analysis.

It includes modules for reading and writing different sequence file formats (for example Fasta and genbank), sequence alignments using common tools such as *BLAST* or *ClustalW* and also provides numerical methods for statistical learning.

The Biopython sequence representation is made using the object *Seq*. It is very similar to a string in Python but with the addition of some biologically relevant methods. An example of a reading of a dataset and printing the id of every read:

```python
from Bio import SeqIO

handle = open("opuntia.aln", "rU")
for record in SeqIO.parse(handle, "clustal") :  
    print record.id
handle.close()
```

As can be seen in this example, researchers have to worry with things they should not, like the "ru" argument in the *open* function or the format of the file they want to open. Also another disadvantage is that researchers have to worry with file handles and its respective management.

**NGS++**

NGS++ is a programming library developed in C++11 that manipulates both NGS data and genomic information files [36]. It offers a powerful set of generic and flexible options to accelerate development and prototyping of epigenomics analysis tools. Due to the fact that NGS++ takes advantage of C++11 anonymous lambda functions, it allows a experienced developer to implement powerful modifications. The interface uTagExp wraps many of the STL algorithms, enabling rapid parallelism via the OpenMP standard [36].
An example of code in NGS++ that would sort and count a subset of a feature.

```cpp
RegionExp.sortSites(
    [](uRegion item1, uRegion item2)
    {return item1.Score5item2.Score},
    &uRegion::getScore, &uRegion::getScore);

int count = RegionExp.getSubsetCount(0.5, 0.8);
```

The biggest NGS++ problem is the fact that a code library is not the best choice if one has little to no programming experience, as the main objective of NGS++ is to offer functionalities to facilitate the development of new tools to experienced programmers. So if one is interested in performing pre-processing in an efficient manner, he is required to know about lambda functions.

As mentioned before, but important to refer again is that biologically savvy ‘computational experts’ are rare outside the major research centers and there is not many people that have biological and informatics knowledge.

**Disadvantage with this approaches in NGS analysis**

In addition to the problems mentioned before, with NGS++ and Biopython usability and Bioconductor lack of parallel execution of functions, all have a common problem, and that is analysis reproducibility. As can be seen in [41], lately there has been increasing focus on the problem of lack of reproducibility of an analysis. However, few programming tools were designed with this specific goal in mind and little to no programming languages are explicitly designed for analysis reproducibility. One of this recommendations is:

- **Sandve et al. recommends, in Rule 3** to archive the Exact Versions of All External Programs Used.

This rule is not taken into account by any of these libraries, because they use programming languages that are totally independent of the libraries and packages implemented by others. A problem can occur when some other user has a different version of the programming language and tries to reproduce an analysis. For example in Bioconductor case, the package *rpvm* is mentioned in [22] and nowadays is removed from R. So, if someone else tries to run a R script (and has the most recent version of R), that uses the Bioconductor package, will for sure receive an error and will not be able to reproduce the experiment. Same applies to NGS++ and Biopython.

**Hadoop/MapReduce/HBase frameworks**

Another existent approach available to researchers to tackle NGS data analysis is using Hadoop, MapReduce or HBase and typically involves distribution of work across a cluster of machines.
which access a shared file system, hosted on a storage area network [48]. Message Passing Interface (MPI), and recently Hadoop MapReduce API are used to implement this parallel processing across different machines. Another model used is cloud computing [10]. As stated by Taylor in [48], Cloud computing equals high performance computing + web interface + ability to rapidly scale up and down for on-demand use.

First, let's start by giving a very brief overview of what these 3 frameworks allow us to do.

- **Hadoop** is a software framework that can be installed on a commodity Linux cluster to permit large scale, fault tolerant, distributed high performance computing [48].

- **MapReduce** allows distributed computing of large data sets stored in a distributed file system with a parallel, distributed algorithm on a cluster.

- **HBase** is an open source, non-relational, distributed database, written in java and runs on top of the HDFS [48] file system.

It should be kept in mind that not all facilities have the right hardware to have Hadoop running and should be consulted for hardware requirements. Due to the previously mentioned fact, if one wants a high computational power and not have to worry about the setting up, "Amazon Elastic Compute Cloud" is generally the common choice. Amazon Elastic Compute Cloud (Amazon EC2) is a web service that provides resizable compute capacity in the cloud. It is designed to make web-scale computing easier for developers.

Keeping in mind the overview given before, a description of the current approaches using these frameworks, and also a critical analysis.

**Hadoop-BAM**

In a typical case, assigning data chunks in a line-based text format to MapReduce jobs is as simple as finding a boundary in which the lines are delimited by some character, i.e. new line.

When doing NGS data analysis, after the alignment of the data, the result is normally stored as a binary format known as BAM (Binary Alignment/Map) [32]. However, the access of the binary content in BAM files, is not a trivial task. Also when using Hadoop with SAM (BAM file is the binary version of SAM) format would result in several times greater disk and network loads.

To resolve the previously mentioned problems the Hadoop-Bam [35] Java library was created, that "acts as a integration layer between the analysis applications and the stored BAM files in Hadoop Distributed File System". To conclude, Hadoop-BAM solves the issues of the access to a given BAM by creating a API to allow the creation of MapReduce functions.

[9] cloudera.com/blog/2013/08/how-to-select-the-right-hardware-for-your-new-hadoop-cluster/
**SeqPig**

Before starting to explain what SeqPig [45] its important to explain what Apache Pig [37] allows to do.

Pig allows to analyse large data sets through the use of a Language (Pig Latin), that when compiled generates sequences of MapReduces for execution within Hadoop.

An example of a pig latin script can be seen in the following code, that prints the names of the students:

```pig
A = LOAD 'student' USING PigStorage()
   AS (name:chararray, age:int, gpa:float);
B = FOREACH A GENERATE name;
DUMP B;
```

The main objective of Pig is to provide the programmer with a high-level language so that one does not need to be worried with the MapReduce paradigm. Also, Pig provides extensibility by allowing users to create user defined functions (UDFs) which can be written in Java, Python, JavaScript, Ruby, or Groovy.

SeqPig is a library for Apache Pig and allows for the distributed analysis of large sequencing datasets on Hadoop. Simple scripts can be created that will allow to manipulate and analyse data, due to the high-level features of Pig. It provides functions for file formats that are normally used in bioinformatics and also user-defined functions for processing aligned and unaligned sequencing data. The file formats BAM, SAM, FastQ, Qseq, and Fasta are commonly used in data sequencing and SeqPig provides import/export functions. These functions where implemented with the help of Hadoop-BAM [35] library.

**Cloudburst**

The most used Alignments tools, like BLAST [7] and RMAP[46], use a technique called seed-and-extend to improve the process of mapping. This technique first starts by finding sub-strings (seed) that are a match in the reference and the reads, and then extend each sub-string into longer, inexact alignments using a algorithm that allows for mismatches or gaps. Cloudburst[43] found a gap in the market due to the fact that all of these approaches run in a single computing node. To do so, Cloudburst implemented a MapReduce-based read-mapping algorithm that runs in parallel on multiple machines with Hadoop.

As can be seen in Figure 1.6, the algorithm is divided into three main phases, due to the MapReduce paradigm. The three phases are map, shuffle and reduce and can be described as follows:

\[\text{http://pig.apache.org/docs/r0.11.1/udf.html}\]
- **Map** This phase emits k-mers (or a small word of length k) as keys for every k-mer in the reference, and for all non-overlapping k-mers in the reads.

- **Shuffle** The shuffle phase extends the seeds into end-to-end alignments that allow mismatch (mutation) errors only or they can allow insertion or deletion (indel) errors.

- **Reduce** This phase extends the seeds into end-to-end alignments allowing for a fixed number of mismatches or indels.

![Cloudburst Algorithm Diagram](image)

Figure 1.6: **Cloudburst Algorithm**. Two grey reference seeds are compared with a single read creating one alignment with two errors and one alignment with zero errors, while the black shared seed is extended to an alignment with three errors. [43]

**Myrna**

Myrna is a cloud computing tool for calculating differential gene expression in large RNA-seq datasets [29]. It is used to analyse large publicly available RNA-seq datasets.

Myrna uses Bowtie [28] an ultra fast memory-efficient short read aligner and R/Bioconductor [22]. This two tools combined, allow for a unified computational pipeline, that can be seen at Figure 1.7.

Myrna is designed to exploit multiple computers and CPUs and can run in three different modes:

- **Cloud mode** using Amazon Elastic MapReduce\(^\text{12}\);

- **Hadoop mode** using a Hadoop cluster;

- **Singleton mode** using a single computer.

\(^\text{12}\)http://aws.amazon.com/elasticmapreduce/
Figure 1.7: The Myrna Pipeline. (a) Reads are aligned using a parallel version of Bowtie. (b) Reads are aggregated into counts for each genomic feature (for example gene). (c) For each sample a normalisation constant is calculated based on a summary of the count distribution. (d) Statistical models are used to calculate differential expression in the R programming language parallelized across multiple processors. (e) Significance summaries such as P-values and gene-specific counts are calculated and returned. (f) Myrna also returns publication-ready coverage plots for differentially expressed genes. [29]

Although it has a good scalability, there are two specific disadvantages of Myrna:

- As Cock et al. say in [16], FastQ files can have multiple encodings. Myrna does not automatically detect the encoding of the dataset, forcing a user to change it, or to run it with the wrong default (in some cases) encoding;

- Running in singleton mode, requires the user to know and set a number of processors.

**Approach Analysis**

There is two possible solutions to achieve high computational power, either use an existenting cloud like the one sold by Amazon services or create a private cloud.
Publicly available cloud services  With the use of a publicly available cloud, there are two main problems:

- **Bandwidth** Every time a researcher wants to process a dataset, either it is publicly available or he has to send it to some cloud. The problem is sending it from the researcher infrastructure to the cloud as Bonnazi says in [9] "You can put gobs of stuff up there but the problem is bandwidth. Sometimes the quickest way for researchers to move large amounts of data to the Amazon cloud is to physically mail a hard drive to Amazon."

- **Data privacy** Researchers need to be believe that their own privacy and the privacy of human subjects are kept secure (legally) in a third party data warehouse.

Private cloud services  This is a solution that researcher are also trying. This has the advantage of reducing security concerns and also data transfer costs, but it comes with the cost of setting up such a system, requiring a big investment by the research facility, and a maintenance team to keep the system working.

A big disadvantage of this kind of approach is the inability of reproducing the data. If in a paper is publicised a Pig Latin script few bioinformaticians will be able to reproduce it. This systems are more indicated for big research laboratories, due to the fact that they imply a lot of time wasted on configurations and setting up.
Chapter 2

NGLess definition and implementation

As mentioned before, there are some disadvantages when using the tools mentioned in the state of the art. One of these disadvantages is the lack of usability of the current tools as they are all much too complex even for a researcher with computational skills. This forces researchers to lose too much time with complex things that they should not be bothered with. We believe that providing researchers with biological context will allow them to easily develop tools and detect mistakes on the scripts and fix them.

Most of the current approaches require the user to have the hassle of manually configuring all external tools when creating pipelines. We believe that the analysis has to be totally transparent to the researchers, in the sense that they do not have to worry about any dependency or external tool.

Another thing we consider of great importance is to provide the researcher with a simple way of setting up a pipeline for NGS data analysis. This allows to increase, immensely, the productivity of the researcher as the pipeline is simpler to create. As NGS researchers have some programming expertise, they can very easily create DSL scripts.

With this DSL, we want to provide a researcher with a context-aware tool that allows for the whole first phase of NGS data analysis, without having to worry with any of the previous mentioned problems. This first phase mentioned includes:

1. Generate statistics of a data set;
2. Pre-processing of a data set;
3. Mapping against genomes;
4. Annotation of the data set.
And so this dissertation aims at making this vision a reality in code. We start by presenting the language based on an example and discuss some of the highlights. After, we explain some key algorithms developed, the interpreter structure and how we deal with external dependencies. Then, we provide an overview of the visualisation component, developed tools to enable script creation, and, to finalise, a section about software engineering aspects. Also, in Appendix A, we present a **formal specification** of the language.

### 2.1 NGLess script example

An example of our language that would generate statistics to multiple files, remove non-unique sequences, pre-process the dataset with a block of operations in each read, that would map and annotate and finally write to a file the results (in this case counts of annotated genes), can be seen in Listing 2.

```
ngless "0.0"

/*
   File paths
*/
fp = ["ctrl1.fq", "ctrl2.fq"]

input = fastq(fp)
input = unique(input, max_copies=2)

preprocess(input) using |read|:
  read = read[5:]
  read = substrim(read, min_quality=26)
  if len(read) < 31:
    discard

// Human reference
human = "hg19"
m = map(input, reference=human)
a = annotate(m, mode="union", features=[{gene}, {cds}])
c = count(a, counts=[{gene}]
write(c, ofile="gene_counts.csv", format={csv})
```

Listing 2: NGLess script example
2.1.1 Highlights

This section has the objective of explaining the key advantages of our Domain Specific Language based on the previous example. A full specification of the Language is in the Appendix A and in here we intend only to provide some highlights of the example that we believe important to mention.

Version declaration

The first line of the example, has a version declaration and it must be present in any NGLess script. The version declaration is very important, since it allows for a researcher to reproduce analysis[41], following a given version of our DSL. Any researcher can reproduce a NGLess script created by someone else, since this version not only keeps the version of the script but also knows the versions of the other programs used by the DSL, achieving in this manner full analysis reproducibility.

Fastq

This function has two key important features to mention: quality encoding prediction and generation of statistics.

FastQ files can have multiple different quality encodings[16] and so it is of key importance to automatically discover which is the encoding on the file. To achieve that, we do a quality prediction based on the lowest character present for all the quality sequences.

Quality control is mandatory and cannot be skipped and, as such, this function generates statistics about the datasets used to allow a researcher to perform a better pre-processing. These statistics are:

- Number of sequences
- Percentage of guanine and cytosine (% GC)
- Minimum and maximum sequence length
- Mean, median, 25th percentile and 75th percentile for each base pair

These statistics are then written to a JSON file and can subsequently be viewed by our visualisation tool.

Preprocessing

Another highlight in the example is the preprocess function that allows for the filtering of a given dataset.
In the example, many different operators can be used as conditions (if), indexation (i.e. var[10:], var[...], etc.), length (operator len), and many different unary and binary operators (can be consulted in Appendix A).

Functions that are commonly used in the biological domain as substrim that performs a trimming based on a minimum quality [16] are also provided.

At the end of the preprocess, quality control (QC) on the resulting data set is performed.

**Map**

The argument reference can either be a path to a data set or the name of a provided reference genome by NGLess. In case the used reference genome, in this case ‘hg19’, is not available locally, it is automatically installed. It is also possible to install the organisms tarball, before the execution of any script. The installation can occur in User or Root mode. The structure of the installed directory, containing the organism, can be consulted at Appendix B in section Organisms.

We did not implement a local aligner and we utilise one of the most commonly used tools for that purpose, Burrows-Wheeler Aligner (BWA) [31]. BWA as a step before aligning that creates an index for a given organism. This step can take many hours, depending on the organism size and only needs to be performed once.

To increase efficiency in the mapping process we provide the user with the index already calculated. The only downside of this approach is the increase in the tarball size of the organism and consequently increasing the transmission time. For an evaluation on the time transmitting the index versus the calculation of the index, see Section 3.

The available reference genomes can be consulted in the documentation at Appendix B section Organisms.

**Annotate**

Our implementation requirements, for annotation, are very similar to the ones provided by HTSEQ-count [8]. The main difference is that we add a few more parametrised options to allow/deny ambiguity and also we allow to filter by more than one genomic feature. For more details on the implemented algorithm, see section 2.2.2.

**Type Validation**

It is crucial to detect and report an error as soon as possible, and in a concise manner. The fact that the domain is so restricted allows us to type validate everything in a very strict manner.

It is important to mention that it is not required the direct use of a value in a function for type validation to occur. For example, if one of the file paths ‘ctrl1.fq’ or ‘ctrl2.fq’, associated with
variable \( fp \), do not exist an error is reported.

**Auto-Comprehension**

It is essential to notice that the function *FastQ* receives a list, this being a characteristic of our language that any function of type \( A \rightarrow \ast \rightarrow B \) can be automatically used as \( [A] \rightarrow \ast \rightarrow [B] \).

### 2.2 Key algorithms

#### 2.2.1 Unique

Due to the size of the FastQ files, the data may not fit into the main memory of a computing device (usually RAM) and instead they must reside in the slower external memory (hard drive).

To remove duplicates, the trivial way is to sort and then filter an amount of repeated reads. As mentioned before, sort of the whole file is out of the question as it could very quickly run out of RAM. The logical alternative is to apply the same process to smaller chunks of a big file. That may result in an error as a read, even though it appears once in a smaller file, may appear repeated in many other smaller files and therefore would be accepted incorrectly.

To solve this problem, we implemented an algorithm that has 3 main steps: scatter, duplicate removal, and gather.

**Scatter**

Scatter is the process of dividing the reads into smaller files. The number of smaller files created are based on how many blocks of 100MB fit in the size of the original file.

The only way of knowing the amount of times a given read is present is to make sure that equal reads end up in the same small file. To achieve that, we take advantage of one of the properties of the hash function:

**Definition 2.2.1** (Hash property). If two values are equal, then applying the hash method on each of the two values must produce the same result.

The function used to assign all reads to all small files, where \( N \) is the amount of small files allowed, can be seen at 2.1.

\[
\forall \text{read} \in \text{File, hash(read)} \mod N \tag{2.1}
\]

**Duplicate removal**

To remove the duplicates we use the data structure Map, and as key we use the sequence of each read. We only load into memory a given read \( X \) times, being \( X \) the amount that the user tolerates
as duplication. This allows to avoid the problem that a great amount of the reads are equal and would cause a memory overflow.

**Gather**

The order of the reads, in a FastQ file, does not matter and, as such, all small files are merged into a bigger, compressed, file.

An example where N reads (amounting to 300MB) and two duplicates are accepted can be seen in figure 2.1. It is important to notice that in the smaller file 1 the read 1 is reduced from 3 to 2.

![Diagram of the gathering process](image.png)

Figure 2.1: Unique algorithm. Tolerate 2 duplicates. (1) scatter N reads across the 3 smaller files. (2) remove duplicates from the smaller files. (3) merge all the smaller files.
2.2.2 Annotate

Annotation requires two files already detailed before: SAM and GFF. The SAM file is dependent on the flow of the script and as such it can range from a small to a big file even for the same FastQ. As for GFF, it keeps the same structure and is relatively small when compared with the SAM file. In a normal execution, annotation will deal with giant SAM files. Thus, it is of key importance to control memory usage.

All features in the GFF file can be represented as an interval (i.e., a range of positions) on a specific chromosome. Due to this fact, we are able to use an Interval Tree [1, Chapter 10] that is a data structure that allows to efficiently find all intervals that overlap with any given interval or point. Operation costs can be located at 2.1.

<table>
<thead>
<tr>
<th>Operation</th>
<th>Cost (Average)</th>
</tr>
</thead>
<tbody>
<tr>
<td>insert</td>
<td>log(n)</td>
</tr>
<tr>
<td>delete</td>
<td>log(n)</td>
</tr>
<tr>
<td>intersecting</td>
<td>log(n)</td>
</tr>
<tr>
<td>intersection</td>
<td>m + n</td>
</tr>
</tbody>
</table>

Table 2.1: Operation costs for data structure Interval Tree

NGLess provides three distinct modes to detect overlaps between reads and features. Special care must be taken to decide how to deal with reads that overlap more than one feature. Those reads are called ambiguous.

There are three possible results: ambiguous, no feature, and the gene name. With NGLess, is also possible to tolerate ambiguity, and, in that case, all the features are counted and ambiguity is no longer a possible result.

Overlaps

For each position \( i \) in the read, a set \( S(i) \) is defined as the set of all features overlapping position \( i \). The equations for the three modes are: mode union (Equation 2.2), mode intersection strict (Equation 2.3), and intersection non empty (Equation 2.4).

\[
\bigcup_{i=1}^{n} S_i 
\]

(2.2)

\[
\bigcap_{i=1}^{n} S_i 
\]

(2.3)

\[
\bigcap_{i=1}^{n} S_i \setminus \emptyset 
\]

(2.4)

Mode union is implemented as a single query by interval, instead of going point by point. This is possible since we take advantage of the function intersecting provided by the Interval Tree.
data structure, that returns all key/value pairs where the key intervals overlap (intersect) the given interval. This allows to go from the cost $O(N \log(N))$, where $N$ is the read size, to simply $O(\log(n))$ on average. This mode is the recommended and also the most commonly used.

Both intersection modes (strict and non-empty), are implemented as the intersection of each, read, position at a time. The cost, for these two modes is $O(N \log(N))$, where $N$ is the read size.

If $S$ has precisely one feature, the read is counted for this feature. If $S$ is empty the read is marked as no feature. In case ambiguity is not tolerated, and $S$ has more than one feature, all ambiguous results are discarded. Otherwise, all features are counted.

Based on [8], Figure 2.2 shows all possible cases of intersection and the corresponding result when ambiguity is not allowed.

<table>
<thead>
<tr>
<th>union</th>
<th>intersection strict</th>
<th>intersection non empty</th>
</tr>
</thead>
<tbody>
<tr>
<td>read</td>
<td>gene_A</td>
<td>gene_A</td>
</tr>
<tr>
<td>gene A</td>
<td></td>
<td>gene_A</td>
</tr>
<tr>
<td>read</td>
<td>gene_A</td>
<td></td>
</tr>
<tr>
<td>gene A</td>
<td></td>
<td>gene_A</td>
</tr>
<tr>
<td>read</td>
<td>ambiguous</td>
<td>gene_B</td>
</tr>
<tr>
<td>gene A</td>
<td></td>
<td></td>
</tr>
<tr>
<td>gene B</td>
<td>ambiguous</td>
<td>ambiguous</td>
</tr>
<tr>
<td>gene A</td>
<td></td>
<td>ambiguous</td>
</tr>
</tbody>
</table>

Figure 2.2: General results to read and gene overlap

2.3 Interpreter Structure

NGLess translates source code into an intermediate representation, abstract syntax tree (AST), and immediately executes it.

To simplify, Figure 2.3 represents the interpretation structure for a single script line. In reality, a list of tokens and AST are passed between the modules.

The following sections provides an overview on the key modules created: Lexer, Parser, Validator and Interpreter.
2.3.1 Lexer

Lexical analysis is the process of converting raw text into a sequence of meaningful tokens. A token in NGLess can be:

- **TExpr** are constant types of expressions. (e.g., String, Symbol)
- **TBop** keeps all binary operators.
- **TReserved** used for all reserved keywords.
- **TWord** any other word (e.g., function name, variable),
- **TOperator** All single character operators (e.g., ',', '[').
- **TNewLine** represents a new line.
- **TIndent** is a sequence of spaces.

The tokens passed to the parser contain information on the line that originated them. To perform the lexical analysis we used a tool named Parsec [30].

2.3.2 Parser

The lexer provides the parser with a list of tokens that allows to create an abstract syntax tree (AST). AST represents the grammatical structure of the parsed input and to create it, a context-free grammar (CFG) is formally specified, in Backus Normal Form, in Appendix A.

To perform the parse is used a tool named Parsec, that is a parser combinator library for Haskell [30]. Parser combinator is a higher-order function which takes parsers as input and returns
a new parser as output. It uses a **predictive parser algorithm** (LL parser) that is backtrack free. However, with the use of a specific combinator, *Parsec* allows for an arbitrary look-ahead.

An example of a NGLess AST generated from Listing 2 **line 8** is:

```
Assignment
  Variable FunctionCall
     |        |
     "input" Flastq Lookup
         |        |
         Variable
             |        |
             "fp"
```

My co-advisor, Luis Pedro Coelho, has a vast knowledge performing NGS data analysis and had already started to develop a first version of the language, before the start of the dissertation. As such I inherited a first version of the parser.

### 2.3.3 Validator

The validation occurs directly on the AST structure. It has two main steps in the validation: check and validate types.

The type checker makes sure that all the parts in the AST have been connected in a consistent way. This step dynamically infers the data types of expressions, allowing to verify if the return type of a given expression is the expected one. For example, it checks that an expression passed to a function has the expected type.

At the validation step, since we have such a small domain we can make it in a very strict way. For example, we only tolerate a given amount of symbols to be used at all in NGLess. This validation makes sure that:

- The return of important functions is used
- File with the specified name exists
- User has appropriate permissions to open a specified file
- Reference organism exists
- Used symbols to parametrise functions are valid

This validation permits the identification of errors before the interpretation which in a more general purpose analysis pipeline would not be possible.
2.3.4 Interpreter

The interpretation process occurs sequentially, by the line number, and therefore each AST, associated with a line, is interpreted at a time. The interpreter outputs metadata (Section 2.5.1) and may also write to disk or to the standard output, depending on the script used.

See Appendix A for details in the interpretation operations.

2.4 External Dependencies

NGLess can be downloaded in two different ways: source or by a binary distribution.

To build from source we provide a Makefile that in a first stage deals with the build of NGLess and in a second stage with the installation. The installation destination (default /usr/local/) can be chosen by the user.

The binary distribution is a self-contained tarball that contains a NGLess executable and all dependencies.

As such, regardless of the way NGLess is downloaded it can be executed from anywhere. To make sure that all external dependencies (e.g., html libraries) are always accessible by NGLess executable, we only use relative paths. The structure of a NGLess installation at directory dir is as follows:

```
dir/
  bin/
  |__ ngless
  share/
  |__ ngless/
  |__ genomes/
  |__ bwa/
  |__ html/
```

The directory bin/ is where we keep the actual NGLess executable. The directory share/ is where we keep all the dependency data. Independently from where the user is executing the DSL, NGLess knows at all times where the dependencies are. For example, bwa is accessible by doing ../share/bwa/ from NGLess executable location.

2.5 Visualisation

NGLess keeps metadata for the most important results and information, so that the results can be both reproducible and visualised in the future.

We do not know a priori the structure of the page at a given time, and so the whole HTML page has to be generated dynamically. Besides using JavaScript and HTML, we also use AngularJS, that is a structural framework for dynamic web applications.
The web design is responsive (RWD). This means that the web page will adapt across a wide range of devices (from mobile phones to desktop computer monitors).

Using NGLess, a local web server can be launched at a specific port, in figure 2.4 port 8000, and all runs are visualisable.

![Welcome to NGLess](image)

**Figure 2.4: Home of NGLess visualising page**

2.5.1 Interpreter results to Browser

We want the user to be able to analyse his results anytime after the execution. In order to do so, for a given script a directory with the script name concatenated with `.ngless_outputs` is created. All dependencies are installed in that directory. All future executions of that script will also write the results to that directory. Figure 2.5 shows the flow of the script from interpreter until visualisation.

2.5.2 Quality Control

Quality control can be present at two different times for a given script. One is mandatory and happens before any pre-process, the other is optional and occurs after the pre-process.

Figure 2.6 shows the quality control (QC) statistics, before pre-process, for an example sample with only the mean visible. The corresponding statistics are the aforementioned ones at Section 2.1.1.

2.5.3 Counts

This mode of visualisation shows the results of the annotation. It is possible to sort (ascending or descending) by the amount of counts, and in this simple manner create a top 10 of the ge-
nomic feature with most counts. It is also possible to filter a genomic feature by the gene name. Figure 2.7, shows an example with descending applied to column counts.

### 2.6 Create Scripts

NGLess provides two modes to create scripts (Figure 2.8). One targets skilled users as every single character of the code has to be written and the other targets users with fewer programming
Figure 2.7: Top 10 genes for file 'CountResults'

skills. The solutions are an editor and a wizard. The wizard is an asset for researchers that are starting to learn NGLess as it is a good starting point.

Figure 2.8: Script creation options

The wizard, developed in AngularJS, enables the creation of scripts while responding to simple questions related to NGS data analysis. This approach eases the script development process as it is only required to deal with natural language to create a NGLess script. The wizard structure is, continuously, updating dynamically, and therefore can be seen the currently developed script at any time.

In the documentation, Appendix B, we present a ’Getting Started’ guide that contains a full tutorial on how to interact with both solutions.
2.7 Software Engineering

This section provides a review on the programming language chosen to develop the DSL, the version control, the build process, documentation and code coverage.

2.7.1 Programming Language

We wanted to use a pure functional programming language and so we have chosen Haskell [33].

Haskell uses lazy evaluation. This is an evaluation strategy that delays the evaluation of an expression until its value is needed and that also avoids repeated evaluations. However, lazy evaluation can create a big expression thunk (a value that is yet to be evaluated) that might give an overflow. Using Haskell, it is possible to control the evaluation and force the evaluation of an expression.

Using Haskell any code that uses I/O is marked with it forever. This way, it encourages the developer to separate code with side effects (impure) from code without (pure). Haskell makes impure functions available, but isolates them by using monads. A Monad is an object whose methods return monads.

To manage and install the correct version of Haskell library dependencies, used in NGLess, we take advantage of the common architecture for building applications and libraries (Cabal).

Cabal

Cabal is a system for building and packaging Haskell libraries and programs. It has a common interface for package authors and distributors to easily build and share their applications. We use it to generate NGLess executables.

However, Cabal is not a package manager and for that reason many conflicts may occur due to library dependencies during the installation of a new package. This is a known problem named cabal dependency hell. To avoid it, we install all dependencies in a isolated environment.

2.7.2 GitHub

GitHub\(^1\) was used as a version control system during the development. It also allows to make the code publicly, open source, available under the terms of the MIT license \(^2\).

\(^1\)http://github.com/
\(^2\)http://opensource.org/licenses/MIT
2.7.3 Build

To enable an user to build NGLess from source we use the GNU Make ³. The provided Makefile follows the GNU standard targets ⁴ as install, clean, dist, distclean, check, etc..

The Makefile allows to keep the git repository clean of all external dependencies as they are only downloaded at build time and if they are not already present locally.

2.7.4 Documentation

Documentation is automatically generated from Git. To do so, we use a tool named Read The Docs ⁵ that hosts documentation, making it fully searchable and easy to find. This feature is available once the repository is public.

At Appendix B is a simplified version of the documentation, in PDF format, to simply demonstrate its content.

2.7.5 Code Coverage

Code coverage is a measure used to describe the degree to which the source code of a program is tested in a test suite. A program with high code coverage has been tested more thoroughly and has a lower chance of containing software errors than a program with lower code coverage.

We used a tool named HPC [23], Haskell program coverage, to calculate the coverage and the results can be seen at Figure 2.9. Has can be seen, there is 82% total code coverage.

³http://www.gnu.org/software/make/
⁵https://readthedocs.org
<table>
<thead>
<tr>
<th>module</th>
<th>Top Level Definitions</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>% covered / total</td>
</tr>
<tr>
<td>module Tests.Analysis</td>
<td>100% 4/4</td>
</tr>
<tr>
<td>module Tests.Assert</td>
<td>100% 2/2</td>
</tr>
<tr>
<td>module Tests.Complete</td>
<td>100% 10/10</td>
</tr>
<tr>
<td>module Tests.FastQ</td>
<td>100% 9/9</td>
</tr>
<tr>
<td>module Tests.Validation</td>
<td>100% 1/3</td>
</tr>
<tr>
<td>module Tests.Util</td>
<td>100% 25/25</td>
</tr>
<tr>
<td>module Tests.Util</td>
<td>100% 2/2</td>
</tr>
<tr>
<td>module Tests.Util</td>
<td>100% 8/8</td>
</tr>
<tr>
<td>module Tests.Util</td>
<td>100% 3/3</td>
</tr>
<tr>
<td>module Tests.Util</td>
<td>100% 6/6</td>
</tr>
<tr>
<td>module Tests.Util</td>
<td>100% 9/9</td>
</tr>
<tr>
<td>module Tests.Util</td>
<td>100% 4/4</td>
</tr>
<tr>
<td>module Tests.Util</td>
<td>100% 2/2</td>
</tr>
<tr>
<td>module Tests.Util</td>
<td>100% 10/10</td>
</tr>
<tr>
<td>module Tests.Util</td>
<td>100% 3/3</td>
</tr>
<tr>
<td>module Tests.Util</td>
<td>100% 267/267</td>
</tr>
<tr>
<td>module Tests.Util</td>
<td>95% 21/22</td>
</tr>
<tr>
<td>module Tests.Util</td>
<td>93% 15/16</td>
</tr>
<tr>
<td>module Tests.Util</td>
<td>88% 23/26</td>
</tr>
<tr>
<td>module Tests.Util</td>
<td>87% 7/8</td>
</tr>
<tr>
<td>module Tests.Util</td>
<td>85% 17/20</td>
</tr>
<tr>
<td>module Tests.Util</td>
<td>83% 20/24</td>
</tr>
<tr>
<td>module Tests.Util</td>
<td>81% 18/22</td>
</tr>
<tr>
<td>module Tests.Util</td>
<td>80% 4/5</td>
</tr>
<tr>
<td>module Tests.Util</td>
<td>80% 4/5</td>
</tr>
<tr>
<td>module Tests.Util</td>
<td>76% 10/13</td>
</tr>
<tr>
<td>module Tests.Util</td>
<td>72% 45/62</td>
</tr>
<tr>
<td>module Tests.Util</td>
<td>68% 11/16</td>
</tr>
<tr>
<td>module Tests.Util</td>
<td>61% 8/13</td>
</tr>
<tr>
<td>module Tests.Util</td>
<td>58% 24/41</td>
</tr>
<tr>
<td>module Tests.Util</td>
<td>54% 18/33</td>
</tr>
<tr>
<td>module Tests.Util</td>
<td>50% 1/2</td>
</tr>
<tr>
<td>module Tests.Util</td>
<td>47% 9/19</td>
</tr>
<tr>
<td>module Tests.Util</td>
<td>40% 11/27</td>
</tr>
<tr>
<td>module Tests.Util</td>
<td>32% 16/49</td>
</tr>
<tr>
<td>Program Coverage Total</td>
<td>82% 668/809</td>
</tr>
</tbody>
</table>

Figure 2.9: Code coverage of NGLess
Chapter 3

Results and Discussion

This section provides an analysis, memory and space, on key operations provided by NGLess. Time elapsed and memory used (on a given execution) allows to perform an asymptotic analysis, which is a method to describe a limiting behaviour.

Before proceeding to the results, it is better to start by explaining how the results were achieved and what they mean.

Measurements of elapsed time are dependent on many external factors and multiple executions of the same function may lead to completely different results. To remove this uncertainty, we use a tool named criterion that samples a given test 100 times. It uses a technique known as bootstrap, that allows to remove outliers in our measurements (timings that are far from the mean) and are perturbing our results in a significant way. We also take into account the cost of the clock calls.

The results related to time are presented with an upper and lower bound of 95% confidence for each statistical measures:

Mean execution time is calculated from execution time divided by number of iterations.

Standard deviation measures the amount of variation or dispersion from the average.

All results were generated with samples related to the human genome (hg19):

- Three FastQ files with 500MB, 2GB and 5GB in size. All reads, in all samples, contain exactly 50 BP.

- Three SAM files with 500MB, 2GB and 5GB in size.

- Human genome and respective annotation file (GFF).

In the following sections, a time and memory analysis, as well as other properties that we felt important to benchmark.
3.1 Fastq function

This section presents a time and memory analysis for the FastQ function. We used the three FastQ samples to perform the analysis.

3.1.1 Time

Table 3.1 shows that the mean results for the samples are 5.64, 20.59 and 43.47 seconds. The sample 5GB is, approximately, 10 and 2.5 times bigger, in size, than the samples with 500MB and 2GB, respectively.

<table>
<thead>
<tr>
<th></th>
<th>500MB</th>
<th>2GB</th>
<th>5GB</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mean</td>
<td>5.64 s</td>
<td>22.64 s</td>
<td>54.89 s</td>
</tr>
<tr>
<td>Std Dev</td>
<td>141.13 ms</td>
<td>3.461 s</td>
<td>2.96 s</td>
</tr>
</tbody>
</table>

Table 3.1: Mean and standard deviation for the fastq function execution time

It is easy to see that with the increase on the file size, the time increases in a linear manner. For example, if we multiple the mean of the 500MB by 10, we obtain a value similar to the mean of the sample with 5GB (5.64 × 10 = 56.4 ≈ 54.89). Same process can be applied against the sample with 2GB (5.64 × 4 = 22.56 ≈ 22.64). This, allows to conclude that:

**Time complexity:** $O(n)$, where $n$ is the number of reads.

3.1.2 Memory

The function is designed to be independent in the FastQ file size and, as such, the increase in file size does not affect the memory use (approximately 2MB for all samples). The only possible, small variation in memory usage is with the increase in the amount of base pairs, by a read, but that variation is so small that is considered to be redundant.

**Space complexity:** $O(1)$

3.2 Unique function

For the purpose of the benchmark, the unique function is parametrised to only tolerate two duplicates.

The minimum and maximum file sizes, originated by the scatter step, can be consulted at Table 3.2. All small files have approximately 100 MB, which indicates that the scatter step distributes the files in a balanced way (Section 2.2.1).
### 3.2.1 Time

This function is dependent on the **FastQ size**. Table 3.3 contains the timing results of the unique function. Applying the same calculations as in the **fastq function** (Section 3.1.1), we can conclude that:

**Time complexity:** $O(n)$, where $n$ is the number of reads.

<table>
<thead>
<tr>
<th>Samples</th>
<th>Small Files</th>
<th>Min Size</th>
<th>Max Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>500MB</td>
<td>6</td>
<td>104 MB</td>
<td>97.5 MB</td>
</tr>
<tr>
<td>2GB</td>
<td>21</td>
<td>101 MB</td>
<td>98.0 MB</td>
</tr>
<tr>
<td>5GB</td>
<td>51</td>
<td>102 MB</td>
<td>96.2 MB</td>
</tr>
</tbody>
</table>

Table 3.2: Minimum and maximum (small) file sizes

<table>
<thead>
<tr>
<th></th>
<th>500MB</th>
<th>2GB</th>
<th>5GB</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mean</td>
<td>109.61 s</td>
<td>354.10 s</td>
<td>794.15 s</td>
</tr>
<tr>
<td>Std dev</td>
<td>2.21 s</td>
<td>7.54 s</td>
<td>15.10 s</td>
</tr>
<tr>
<td>Estimate</td>
<td>110.10 s</td>
<td>355.71 s</td>
<td>797.51 s</td>
</tr>
<tr>
<td>Upper bound</td>
<td>110.57 s</td>
<td>357.54 s</td>
<td>803.08 s</td>
</tr>
</tbody>
</table>

Table 3.3: Mean and standard deviation for the unique function execution time

### 3.2.2 Memory

The memory usage is limited by the size of each small file as it is only loaded, into memory, one at a time.

![Memory usage of unique function](image.png)

Figure 3.1: Memory usage of unique function
As can be seen in figure 3.1, independently of which data set (2 or 5 GB), the memory limit reached is roughly the same. We can conclude that as the FastQ size increases, the memory limit is the same. The spikes represent each small file being processed and therefore the 2GB and 5GB samples have 21 and 51 spikes, respectively.

We can now conclude that, independently of the file size, the memory usage is majored by the biggest small size resultant of the scatter process. Therefore, we can conclude that:

**Space complexity:** $O(n)$, where $n$ is the biggest small file created at the scatter step.

### 3.3 Preprocess function

To benchmark the preprocess function we used two operations, substrim ($O(n)$, where $n$ is the read quality sequence size) and indexation ($O(1)$).

The function substrim tolerates a minimum quality of 5 and the indexation removes the last element of the read (sequence and quality).

#### 3.3.1 Time

As both operations are equally applied to all samples the pre-process time should grow in a linear way. Looking at the results in Table 3.4, is possible to see that the mean of the 500MB multiplied by a factor of 10 results in a very similar value to the mean of 5GB, as expected ($94.69 \times 10 = 946.9 \approx 1016.01$).

**Time complexity:** $O(n)$, where $n$ is the number of reads.

<table>
<thead>
<tr>
<th></th>
<th>500MB</th>
<th>2GB</th>
<th>5GB</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lower bound</td>
<td>Mean</td>
<td>Std dev</td>
<td>Mean</td>
</tr>
<tr>
<td></td>
<td>94.43 s</td>
<td>935.92 ms</td>
<td>353.01 s</td>
</tr>
<tr>
<td>Estimate</td>
<td>94.69 s</td>
<td>1.15 s</td>
<td>354.36 s</td>
</tr>
<tr>
<td>Upper bound</td>
<td>94.88 s</td>
<td>1.47 s</td>
<td>356.27 s</td>
</tr>
</tbody>
</table>

Table 3.4: Mean and standard deviation for the preprocess function execution time

#### 3.3.2 Memory

This function is implemented in a way that each read, in a FastQ file, is loaded at a time and as such the memory should grow in a constant way. Table 3.5, allows to see that with an increase in the sample size, the memory is always, approximately, the same. Looking at Table 3.5 we can see that:
Space complexity: $O(1)$

### 3.4 Annotation function

To benchmark this function, we used as annotation file the human genome (hg19) and all three SAM samples. We parametrised the function as:

- **Mode**: Using overlap union mode
- **Features**: Only allow genomic feature gene
- **Keep ambiguous**: Do not allow ambiguous overlaps

#### 3.4.1 Time

The implementation forces the evaluation of the GFF first. As can be seen in Figure 3.2, independently of the SAM size, the first 50 seconds take place loading the GFF. If we subtract that 50 seconds to the mean of the 500MB and 2GB samples and multiply the 500MB by a factor of 4 the result is very similar ($91 - 50 = 41 \times 4 = 164$ and $220 - 50 = 170$). From this we conclude two facts:

- GFF loading is constant as the SAM size increases.
- Function time grows linearly as the SAM size increases.

Time complexity, could be defined as $O(m + n)$ but as the SAM file is, normally, much bigger than the the GFF the complexity tends towards:

**Time complexity**: $O(n)$, where $n$ is the SAM file size.

<table>
<thead>
<tr>
<th>Samples</th>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>500MB</td>
<td>4 MB</td>
</tr>
<tr>
<td>2GB</td>
<td>4 MB</td>
</tr>
<tr>
<td>5GB</td>
<td>4 MB</td>
</tr>
</tbody>
</table>

Table 3.5: Memory usage for the preprocess function

<table>
<thead>
<tr>
<th>500MB</th>
<th>2GB</th>
<th>5GB</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mean</td>
<td>Std dev</td>
<td>Mean</td>
</tr>
<tr>
<td>90.46 s</td>
<td>1.23 s</td>
<td>217.80 s</td>
</tr>
<tr>
<td>Estimate</td>
<td>90.80 s</td>
<td>1.10 s</td>
</tr>
<tr>
<td>Upper bound</td>
<td>91.32 s</td>
<td>1.41 s</td>
</tr>
</tbody>
</table>

Table 3.6: Mean and standard deviation for the annotate function execution time
Time comparison with similar tool

The features provided by the annotation are very similar to the ones provided by the HTSEQ-count. As such, we can compare the times for the same SAM sample, see Table 3.7. It is clear from the results that using this common parametrisation we are, at least, 4 times faster to produce exactly the same result.

<table>
<thead>
<tr>
<th>SAM Samples</th>
<th>NGLess</th>
<th>HTSEQ-count</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>500MB</td>
<td>91</td>
<td>445</td>
<td>4.8</td>
</tr>
<tr>
<td>2GB</td>
<td>220</td>
<td>1071</td>
<td>4.8</td>
</tr>
<tr>
<td>5GB</td>
<td>490</td>
<td>2383</td>
<td>4.8</td>
</tr>
</tbody>
</table>

Table 3.7: Annotation speedup of NGLess against HTSEQ-count

3.4.2 Memory

Figure 3.2 shows the memory flow for two samples with completely different sizes. As can be seen, the memory grows at the same rate independently of the SAM file size and in the first 50 seconds is when the GFF file is loaded. Afterwards, the memory stabilises as it does not depend on the SAM file size.

Figure 3.2: Memory usage for the function annotate

Space complexity: $O(n)$, where $n$ is the GFF file size.
3.5 Map function

As mentioned before, we provide the index required by BWA already calculated, and therefore transmitted through the internet. This section performs an evaluation to see if its worth doing it. We also provide an analysis on the map execution time, both in a serial and parallel manner.

<table>
<thead>
<tr>
<th>Organism</th>
<th>No preprocess</th>
<th>Preprocessed</th>
<th>Difference</th>
</tr>
</thead>
<tbody>
<tr>
<td>Saccharomyces Cerevisiae</td>
<td>8 MB</td>
<td>32 MB</td>
<td>24 MB</td>
</tr>
<tr>
<td>Caenorhabditis Elegans</td>
<td>64 MB</td>
<td>256 MB</td>
<td>192 MB</td>
</tr>
<tr>
<td>Drosophila Melanogaster</td>
<td>64 MB</td>
<td>256 MB</td>
<td>192 MB</td>
</tr>
<tr>
<td>Canis Familiaris</td>
<td>768 MB</td>
<td>4100 MB</td>
<td>3332 MB</td>
</tr>
<tr>
<td>Rattus Norvegicus</td>
<td>1024 MB</td>
<td>4200 MB</td>
<td>3176 MB</td>
</tr>
<tr>
<td>Bos Taurus</td>
<td>1024 MB</td>
<td>4300 MB</td>
<td>3276 MB</td>
</tr>
<tr>
<td>Mus Musculus</td>
<td>1300 MB</td>
<td>4500 MB</td>
<td>3200 MB</td>
</tr>
<tr>
<td>Homo Sapiens</td>
<td>1024 MB</td>
<td>25600 MB</td>
<td>24576 MB</td>
</tr>
</tbody>
</table>

Table 3.8: Tarball, organism, sizes with and without preprocess

Table 3.8 shows the size of the tarball with and without the index calculated and the respective disk space increment. As can be seen, the processing greatly increases the size of the tarball, mainly by a factor of 4 except for the human that increases by a factor of 25.

To see if the transmission of the index is worth it, we present a histogram that compares the calculation time of the index versus the transmission, for all NGLess provided organisms, see Figure 3.3.

![Figure 3.3: BWA Index calculation vs transmission. Assuming an average bandwidth of 1MB/s.](image)

As can be seen, for smaller organisms the index transmission is not worth it, as we would spend more time transmitting than calculating. But as the organism size increases its clear that is worth the transmission.

For example, the human genome takes 38 hours to calculate the index required for the alignment, and the transmission (assuming a bandwidth of 1MB/s) takes only 7 hours, which is 5.5
times faster. It is actually worth it to transmit for an average bandwidth higher than 180KB/s.

Table 3.9 presents the benchmark results, associated with the map operation, for the different samples. Table 3.10 provides the speedup of the map operation for a sample SAM with 5GB.

<table>
<thead>
<tr>
<th>500MB</th>
<th>2GB</th>
<th>5GB</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mean</td>
<td>Std dev</td>
<td>Mean</td>
</tr>
<tr>
<td>Lower bound</td>
<td>630.20 s</td>
<td>17.7</td>
</tr>
<tr>
<td>Estimate</td>
<td>680.85 s</td>
<td>19.5</td>
</tr>
<tr>
<td>Upper bound</td>
<td>720.52 s</td>
<td>21.6</td>
</tr>
</tbody>
</table>

Table 3.9: Mean and standard deviation for the map function execution time

<table>
<thead>
<tr>
<th># threads</th>
<th>time</th>
<th>speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>5964</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>1518</td>
<td>3.9</td>
</tr>
<tr>
<td>8</td>
<td>927</td>
<td>6.4</td>
</tr>
</tbody>
</table>

Table 3.10: Speed up of map operation

3.6 Overall execution

From the previous sections, it is possible to infer that the map function is the operation that takes most time to execute. Figure 3.4 (a) shows that for a sample with 5GB, 72% of the total execution time takes place at the map operation. This percentage can even increase if it is required to calculate the index for the alignment. As for all the other operations, the sum is only 28% of the execution time.

Figure 3.4: Functions percentage on total execution time

It is important to mention that the map function can execute in parallel and if we allow NGLess to run with 8 threads, the map execution percentage reduces from 72 to 28. Figure 3.4 (a) and
(b) have total execution times of 8321 and 3284 seconds, respectively. This allows to infer that with 8 threads, there is a speedup of approximately 2.5.
Chapter 4

Conclusions

As could be seen in the state of the art, there are a lot of different approaches to create pipelines for NGS data analysis. The current solutions are more concerned with high performance than with the usability provided to the researcher. Also, all the approaches depend upon the current Unix/Linux tools as they are merely an interface to abstract the complexity of using a terminal to make the analysis, especially the workflow engine systems.

Libraries and packages solutions, are a good approach as they try to provide the researcher with a familiar domain (this case NGS analysis) but the programming languages that they require to make the analysis are just too generic, and the user either knows how to work with the programming language or will waste a lot of time learning it.

Tools that use the Cloud or Hadoop to improve the performance are just too complex for a researcher outside major research centers. They are good solutions for big and advanced laboratories that have informatic experts to help on configuring all dependencies and mounting systems.

The different kinds of approaches are still error prone, due to the fact that they are too generic to be able to validate pipelines at a biological level.

NGLess fills a gap in the market by using a domain specific language that has, in fact, biological knowledge. This context awareness allows us for a rich type validation. The fact that we limit our domain specific language to a restricted set of operations allows us to provide everything (e.g., data types, operations, etc.) in a much more concise way and completely oriented to NGS data analysis.

NGLess improves researchers productivity as they do not have to lose time on things as non standard formats, external tools dependencies, etc.. Productivity is also increased as the development of a pipeline for NGS data analysis, is now, quite simple.

Reproducing an analysis can be easily made, since we keep versions of everything. This allows researchers to share with the scientific community, in a simple way, the pipeline used that generated the results.
4.1 Future Work

From feedback of researchers using NGLess, a feature that is commonly asked is the possibility of using different alignment tools. In a future version of NGLess should be added a parametrised argument to choose between a list of possible aligners. The infrastructure is already in place for the BWA aligner and as such adding a new aligner should not be hard.

Even though it is not of key importance, a future feature is to make NGLess operations (excluding map) in parallel as it is now embarrassingly parallel.
Bibliography


Appendix A

Specification

NGLess is a, imperative, domain specific language that infers variable types from context and is presented in a precise manner in the following document.

1 Basic Characteristics

1.1 Data Types

There are 4 (four) basic data types that can be instantiated in NGLess: Integers, chains of characters (String), Booleans (Bool) and Symbols.

- **Integer** can hold any number, no matter how big.
- **String** is a chain of Unicode characters.
- **Bool** is an enumeration that can be either True or False.
- **Symbol** represents a tag that may have different meaning in other contexts.

There are other data types, that are related to the biological context, but they can not be directly created. They are:

- **ReadSet** keeps information about a data set.
- **ShortRead** is a read of a given data set.
- **MappedReadSet** keeps information about the map.
- **AnnotatedSet** stores annotation results.
- **Void** keeps no information.

In addition, is supported the composite type **List of X** where X is a basic type. All elements in **List** must have the same type. Lists are built with square brackets (e.g., \([1,2,3]\), a list of three integers).

The data types supported by each operator are indicated in the expression definition (§ 6).

1.2 Name manipulation

The names (§ 2.6) correspond to constants and variables. In the following topics the term entity is used to designate them.
1.2.1 Name space and identifiers visibility

The name space is global and unique and for that reason a name used to designate an entity, on a given context, cannot be used to designate any other.

Identifiers are always visible. In particular, a redefined identifier on a lower context will modify it at a upper context.

1.2.2 Validate variables

The global entities (at the higher context), exist during the whole execution of the program. The variables, local to a block, exist only during his execution.

1.3 Auto comprehension

A function that receives a the data type A and returns B (A $\rightarrow$ * $\rightarrow$ B) can be automatically used as receiving a List of A and returning a List of B (i.e $[A] \rightarrow * \rightarrow [B]$).

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

In the previous example, function fastq receives a List of String and returns a List of ReadSet.

2 Lexical Conventions

For each of the lexical elements (tokens), is considered the biggest sequence of characters existent that constitutes a valid lexical element.

2.1 Line structure

A program is divided into logical lines and one instruction cannot occupy more than one line, except if the instruction explicitly allows for a line change, for example the if condition.

2.1.1 Physical lines

A Physical line may have any size. The termination of a line (eol) does not depend on the operating system and must equally work in UNIX, MAC OS and Windows. The line feed (U+000A, \n) and carriage return (U+000D, \r) are both accepted as Physical line (eol) terminators.

2.1.2 White characters

Are considered white characters those that, even though are used to separate lexical elements, do not represent one.

Is considered white characters the Unicode character space U+0020 (White Space, ' '). The character tabulation, U+0009 (HT, \t), cannot be used in NGLess as a white character.

The mentioned character, even though is white, has special meaning if appears at the beginning of a logical line. Similarly, the characters of line change (eol) also have special meaning in indentation (§ 2.1.4) and comments (§ 2.2).

2.1.3 White lines

A logical line that only contains white characters and comments is ignored, not being generated any lexical element neither changing the indentation.
2.1.4 Indentation

As previously mentioned in § 2.1.2, the character space is not considered white when in the beginning of a logical line.

The total number of white spaces until the first non-white character defines the level of indentation of the block. Each block must have the same level of indentation and it should be a multiple of 4.

Even though the levels of indentation might be different, the meaning can be the same:

```
1. ngless "0.0"
2. in = fastq('sample.fq')
3. preprocess(in) using |read|
4. read = read[3:]
5. if len(read) < 20:
6. discard
7. m = map(in,reference='ce10')
```

a. Correct indentation.

```
1. ngless "0.0"
2. in = fastq('sample.fq')
3. preprocess(in) using |read|
4. read = read[3:]
5. if len(read) < 20:
6. discard
7. m = map(in,reference='ce10')
```

b. Correct indentation.

```
1. ngless "0.0"
2. in = fastq('sample.fq')
3. preprocess(in) using |read|
4. read = read[3:]
5. if len(read) < 20:
6. discard
7. m = map(in,reference='ce10')
```

c. Incorrect indentation.

Examples a) and b) are correct as both have an increasing indentation level and all instructions that are at the same level are consistent with the context. At example c), the instruction with the 'preprocess' function (Line 3) has two indentation errors:

- Indentation level higher than the one at the current context.
- The provided block has lower indentation level and should have higher.

2.2 Comments

The comments work as separators of lexical elements. There are two kinds of comments:

**Single-line** Start with # or // (as long as the sequence does not belong to a chain of characters) and end at the line termination.

**Multi-line** Start with /* and terminates with */ (if the sequence does not belong to a chain of characters). Cannot be nested.

2.3 Key Words

The following words are reserved and do not constitute identifiers:

```
if, else, ngless, len, discard, continue, using
```

2.4 Operators

Are considered operators the following lexical elements:

```
= += -= [] ! == >= < >=
```

2.5 Delimiters and Terminators

The following lexical elements are considered delimiters/terminators:

```
, : ( )
```
2.6 Identifiers (names)

Are initiated by a letter (uppercase or lowercase) or by a '_' (underscore). The first character can be followed by 0 (zero) or more letters, digits and underscores. The number of characters that constitute an identifier is unlimited and two names are distinct if there is a transformation of uppercase to lowercase, or vice versa, of at least one character.

2.7 Literals

Are notations for constant values of data types provided by NGLess.

2.7.1 Integers

An integer literal is a non rational number non negative (can be negative by the application of the unary operator (-) to a positive literal).

An integer literal in decimal is constituted by a sequence of 1 (one) or more digits (from 0 to 9).

An integer literal in hexadecimal starts with the sequence 0x, followed by one or more digits from 0 to 9, a to f or A to F.

There is an unlimited size representation to a integer.

2.7.2 Chain of characters (string)

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

Special sequences are initiated by a \. They can be represented by the characters LF and CR (\n and \r respectively), quotation marks ('') or slash (\).  

2.7.3 Boolean

A boolean literal can be represented as word True or False, with the first letter in upper or lower case.

2.7.4 Symbol

The representation is a token involved in curly brackets ({}). (e.g., {CDS} or {gene}). Symbols have specific meaning to functions (§ 7) as they allow to parametrise them.

2.8 Complex data type format

This data types result always from function invocations and cannot be created directly.

2.8.1 ReadSet

A ReadSet is a file in the FastQ format. It has 0 or more ShortRead.

2.8.2 ShortRead

A ShortRead is each read of a ReadSet with the following structure:

(read) ::= '@'(seqname)'\n'(seq)'\n'+\n'(qual)'\n'

(seqname) ::= String

(seq) ::= String
The `<qual>` represents the quality of the read and has a range of values from 33 to 126.

### 2.8.3 MappedReadSet

MappedReadSet follows the Sequence Alignment/Map(SAM) format. The SAM Format allows to store sequence data in a series of tab delimited columns.

The SAM file is divided into two sections, the header and the alignment. The first contains information of the entire file and additional alignment information. The latter contains the information for each sequence about the alignment.

Each line in the alignment section has 11 mandatory fields. In the following table is represented each field and his type.

<table>
<thead>
<tr>
<th>Column</th>
<th>Field</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>QNAME</td>
<td>String</td>
</tr>
<tr>
<td>2</td>
<td>FLAG</td>
<td>Integer</td>
</tr>
<tr>
<td>3</td>
<td>RNAME</td>
<td>String</td>
</tr>
<tr>
<td>4</td>
<td>POS</td>
<td>Integer</td>
</tr>
<tr>
<td>5</td>
<td>MAPQ</td>
<td>Integer</td>
</tr>
<tr>
<td>6</td>
<td>CIGAR</td>
<td>String</td>
</tr>
<tr>
<td>7</td>
<td>RNEXT</td>
<td>String</td>
</tr>
<tr>
<td>8</td>
<td>PNEXT</td>
<td>Integer</td>
</tr>
<tr>
<td>9</td>
<td>TLEN</td>
<td>Integer</td>
</tr>
<tr>
<td>10</td>
<td>SEQ</td>
<td>String</td>
</tr>
<tr>
<td>11</td>
<td>QUAL</td>
<td>String</td>
</tr>
</tbody>
</table>

### 2.8.4 AnnotatedSet

A AnnotatedSet stores the result of annotating a given MappedReadSet. It follows a tab delimited structure and is represented next:

<table>
<thead>
<tr>
<th>Column</th>
<th>Field</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Id</td>
<td>String</td>
</tr>
<tr>
<td>2</td>
<td>Features</td>
<td>List of Symbol</td>
</tr>
<tr>
<td>3</td>
<td>Counts</td>
<td>Integer</td>
</tr>
<tr>
<td>4</td>
<td>Strand</td>
<td>String</td>
</tr>
</tbody>
</table>

### 3 Grammar

The language grammar can be resumed by the rules described next. Consider that:

- elements in fix type are called literals.
- optional items are enclosed in square brackets (i.e `[^item-x>]]`).
- alternative elements are separated by a vertical bar (`'|`).
- items repeating 0 or more times are suffixed with an asterisk (`'*`).
- items repeating 1 or more times are suffixed with a plus (`'+`).

\[
\langle \text{script} \rangle := \langle \text{version} \rangle \langle \text{body} \rangle
\]
\[\text{version} ::= 'ngless' \text{(literal-string)}\]
\[\text{body} ::= \text{instructions}\]
\[\text{block} ::= \text{indentation} \text{instructions} \text{block}\]
\[\text{instructions} ::= \ast\text{expr}\]
\[\text{expr} ::= 'continue' \mid 'discard'
| \text{conditional}
| \text{iteration}
| \text{assignment}
| \text{funccall}
| \text{inner-expr}\]
\[\text{inner-expr} ::= \text{binary-op}
| \text{base-expri}
| \text{index-expri}
| \text{list-expri}\]
\[\text{base-expri} ::= \text{pexpr}
| \text{literal}
| \text{unary-expri}
| \text{variable}\]
\[\text{variable} ::= \text{word-req} \ast\text{word-opt}\]
\[\text{word-req} ::= '_' \mid \text{letter}\]
\[\text{word-opt} ::= \text{word-req} \mid \text{digit}\]
\[\text{pexpr} ::= '(' \text{expr} ')'
| \text{conditional}
| \text{iteration}
| \text{assignment}
| \text{funccall}\]
\[\text{conditional} ::= 'if' \text{expr} ':' \text{block} [ 'else' ':' \text{block} ]
| \text{iteration} ::= 'preprocess' '(' \text{expr} ')' \text{using}'1' \text{variable} '1' ':' \text{block}
| \text{assignment} ::= \text{variable} '=' \text{expr}
| \text{funccall} ::= \text{func-name} '(' \text{expr} \ast\text{opt-args} ')'
| \text{opt-args} ::= ',' \text{opt-arg}\]
\[\text{opt-arg} ::= \text{variable} '=' \text{expr}\]
\[\text{list-expri} ::= '+' '1' | '+' \text{inner-expr} \ast\text{inner-expr-opt} '1'
| \text{inner-expr-opt} ::= ',' \text{inner-expr}\]
\[\text{index-expri} ::= \text{base-expri} \text{index-one}
| \text{base-expri} \text{index-two}\]
\[\text{index-two} ::= '+' ['\text{expr}'] ':' ['\text{expr}'] '1'\]
The precedence of binary and unary operators are described in detail at section § 6. Also, the values that the literals can take are defined in § 2.7.

### 3.1 Left value

The elements of an expression (operators) that can be used as a left-value are individually identified in section § 6.

### 3.2 Script

Is designated by script the file that contain all the code to run on NGLess. All scripts must be in UTF-8 format.

### 3.3 Variables and constants

#### 3.3.1 Initialization

Is performed with a value that follows the operator $=$ ("equal"): integer (an Integer expression), string (a String expression), boolean (a Bool expression) and a symbol (a Symbol expression). Examples:

<table>
<thead>
<tr>
<th>Type</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Integer</td>
<td>i = 2</td>
</tr>
<tr>
<td>String</td>
<td>s = 'hey'</td>
</tr>
<tr>
<td>Boolean</td>
<td>b = True</td>
</tr>
<tr>
<td>Symbol</td>
<td>s = {gene}</td>
</tr>
</tbody>
</table>

A7
To associate a variable with an array of expressions, it’s required to start with the [ operator and terminate with ]. The expression should have the same type. Examples:

- List of symbols  \[ \Rightarrow \text{ls} = \{ \{\text{gene}\}, \{\text{CDS}\} \} \]
- List of strings  \[ \Rightarrow \text{ls} = \{\text{‘fp1’}, \text{‘fp2’}\} \]

### 3.3.2 Constants

The language allows for the definition of constant identifiers, preventing it of being used in operations that modify it’s value. All characters in the identifier must be in upper case. Examples:

- Constant integer  \[ \Rightarrow \text{CI} = 2 \]
- Constant string  \[ \Rightarrow \text{CS} = \text{‘hey’} \]
- Constant boolean  \[ \Rightarrow \text{CB} = \text{True} \]
- Constant symbol  \[ \Rightarrow \text{CS} = \{\text{gene}\} \]

### 4 Functions

A function allows to execute predefined code to a given set of parameters, that are provided by argument.

With NGLess is not possible to define new functions. Nonetheless, a big variety of functions is provided and are described in § 7.

#### 4.1 Invocation

A function can only be invoked by the use of an identifier that refers to one of the provided functions. After the identifier, the delimiter ( is opened to refer to the start of the arguments and ended with the delimiter ).

Functions have a single positional parameter and all other must be provided by name. Example:

\[
\text{result} = \text{f(arg, arg1=2)}
\]

The argument \text{arg} can be any data type (§ 1.1) as for \text{arg1} it is an expression that evaluates either to a string, integer, bool or a symbol. The variable \text{result} will store a new data type that is consequence of executing the function \text{f}.

In most cases, all arguments are passed by value and no modification is made by the execution of a given function. In particular, there is one exception (pre-process) and is detailed in § 4.4.

#### 4.2 Parametrize functions

To parametrize functions, arguments must be passed by name. They are optional as all functions have default values, in case one is not provided. It works as a variable assignment but reflects only to the function.

It is not possible to pass arguments by name to a function that has no use to them. The names and possible values must be followed and are detailed in § 7.

#### 4.3 Pure functions

The result of calling a pure function must be assigned to another variable. At § 7 is indicated which function are pure.
4.4 Special invocation

Instead of multiple parameters, there is the case where a function takes a single positional parameter and a block. The block is a closure that is passed to the function and the parameter is passed by reference, which means that the variable provided will be modified.

The block is passed with the using keyword. All instructions in the block must have the same indentation, using white space (U+0020, ' '). Example:

```plaintext
f(all) using |i|:
    block

    It works as a for each, the contents of the variable all is traversed and kept in the variable i for each iteration.
    At the end the result of the function is assigned to the variable passed as argument, in this example all. The flow of the execution must continue even if the result is an empty data set.
```

5 Instructions

All instructions are executed in sequence.

5.1 Conditional instruction

If the expression evaluates to a true boolean then the block that follows the operator ':' is executed.

If the expression evaluates to a false boolean and is present the reserved word else and delimiter ':' , the else block is executed. If evaluates to false and the reserved word else is not present, nothing happens.

Case both the reserved word if and else are present, exactly one of the two blocks will be executed.

5.2 Iteration instruction

It is only possible to iterate the data type ReadSet. A ReadSet has zero or more ShortRead.

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

    The variable read (ShortRead) contains each read of the variable reads (ReadSet).
    The variable passed to the 'preprocess', in this case reads, is passed by reference and so every modification to the variable read in the block modifies the variable reads.
```

5.3 Discard instruction

Discard instruction can only be used inside an iteration instruction (§ 5.2).

Indicated by the reserved word discard (when executed, is the last instruction in the block) it removes the current ShortRead (being iterated) from the ReadSet and jumps for the next iteration.

5.4 Continue instruction

Discard instruction can only be used inside an iteration instruction (§ 5.2).

Indicated by the reserved word continue (when executed, is the last instruction in the block) it stops executing the block to an ShortRead (being iterated) from the ReadSet and jumps to the next iteration.
6 Expressions

The expression are always evaluated from the left to the right, independent of the operator associativity.

The operators precedence is the same when in the same section, knowing that following sections represent less priority.

The following table is a resume of the possible operators and is grouped by decreasing precedence.

<table>
<thead>
<tr>
<th>Type</th>
<th>Symbol</th>
<th>Precedence</th>
</tr>
</thead>
<tbody>
<tr>
<td>primary</td>
<td>() []</td>
<td>non associative</td>
</tr>
<tr>
<td>unary</td>
<td>- len</td>
<td>non associative</td>
</tr>
<tr>
<td>arithmetic</td>
<td>* +</td>
<td>left to right</td>
</tr>
<tr>
<td>comparative</td>
<td>&lt; &gt; &gt;= &lt;=</td>
<td>left to right</td>
</tr>
<tr>
<td>equality</td>
<td>== !=</td>
<td>left to right</td>
</tr>
<tr>
<td>attribution</td>
<td>=</td>
<td>right to left</td>
</tr>
</tbody>
</table>

For example the following expressions would result in different values, due to precedence:

- $5 + 5 * 2 = 15$
- $((5 + 5) * 2) = 20$

6.1 Primary expressions

6.1.1 Identifiers

One identifier can denote a variable or a constant.

A identifier is the most simple case of a left-value, this is an entity that can be used in the left of an attribution.

6.1.2 Parentheses

Expressions between parentheses, "(" and ")", has the same value as without them. This property allows for nested parentheses.

One expression between parentheses can not be a left-value.

6.1.3 Indexation

Indexation expressions return the same data type. They can not be used as a left-value.

Can be used to access one element or a range of elements. To access one element, is required a 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 until length of variable x
- $x[:10]$ returns from position 0 until 10

Indexation can be applied to a List or a ShortRead.

6.1.4 Invocation

A function can only be invoked (§ 4.1) through the use of a identifier that is specified at § 7.
6.1.5 Read
The operation to look up for a given variable value can be performed by simply using its name.

6.2 Unary expressions
The operator \((-\) returns the symmetric of its Integer argument.
The operator \(\text{len}\) returns the length of a List or a ShortRead.

6.3 Multiplicative expressions
This operations are only applicable to Integer values, returning the value of the corresponding algebraic operation.

6.4 Additive expressions
This operations are only applicable to Integer values, as the previous.

6.5 Comparative expressions
This operations are only applicable to Integer values and returns a Bool. Therefore the return can either be true or false.

6.6 Equality expressions
This operations are applicable to any NGLessObject and the return is a Bool.

6.7 Attribution expressions
The value of the expression in the right side of the operator \(\text{=}\) is saved in the variable, indicated by the left-value, at the left side of the attribution operator.

To the same left-value can not be assigned right values with different types.

7 Function Definition
Here is provided the definition of all the available functions. To denote a List of a given data type is used \([X]\) where \(X\) is the data type. (i.e \([\text{Symbol}]\))

7.1 Fastq
Function to load, one or more, FastQ files. An example:

\(\text{in} = \text{fastq('input.fq')}\)

**Argument:**
String

**Return:**
ReadSet
Arguments by value:
none

The String argument must be a valid file path and exist. An error is reported otherwise.

The only compression method supported for the data sets is gzip (.gz).

The encoding prediction is based on the lowest quality character of the fastQ file.

When loading a data set, quality control is performed and statistics can be visualised in a graphical user interface (GUI).

The simple statistics calculated are percentage of guanine and cytosine (%GC), encoding prediction, number of sequences and minimum/maximum sequence length. The more complex statistics calculated are the mean, median, lower quartile and upper quartile for each position of the base pairs.

7.2 Unique
Function that receives a set of reads and returns a equal or smaller set of reads. Only retains a given number of copies for each read (if there are any duplicates). An example:

\[
\text{input} = \text{unique}(\text{input}, \text{max\_copies}=3)
\]

Argument:
ReadSet

Return:
ReadSet

Arguments by value:

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Required</th>
</tr>
</thead>
<tbody>
<tr>
<td>max_copies</td>
<td>Integer</td>
<td>X</td>
</tr>
</tbody>
</table>

The optional argument max\_copies allows to define the number of tolerated copies (default: 1).
Is considered a copy: ShortReads with the same sequence regardless of quality and identifier.

It’s a pure function § 4.3.

7.3 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. An example:

\[
\text{preprocess}(\text{inputs}) \text{ using } |\text{read}|:
    \text{read} = \text{read}[3:]
\]
Argument:
ReadSet

Return:
Void

Arguments by value:
none

This function also performs quality control on its output.

7.4 Map
The function map allows for a ReadSet to be mapped against a reference. An example:

\[
\text{mapped} = \text{map(}\text{input, reference]='sacCer3')}\]

Argument:
ReadSet

Return:
MappedReadSet

Arguments by value:

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Required</th>
</tr>
</thead>
<tbody>
<tr>
<td>reference</td>
<td>String</td>
<td>✓</td>
</tr>
</tbody>
</table>

The argument reference must be either a path to a data set or the name of a provided data set by NGLess. The provided data sets of NGLess are:

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
<th>Assembly</th>
</tr>
</thead>
<tbody>
<tr>
<td>sacCer3</td>
<td>saccharomyces_cerevisiae</td>
<td>R64-1-1</td>
</tr>
<tr>
<td>ce10</td>
<td>caenorhabditis_elegans</td>
<td>WBcel235</td>
</tr>
<tr>
<td>dm3</td>
<td>drosophila_melanogaster</td>
<td>BDGP5</td>
</tr>
<tr>
<td>canFam2</td>
<td>canis_familiaris</td>
<td>CanFam3.1</td>
</tr>
<tr>
<td>rn4</td>
<td>rattus_norvegicus</td>
<td>Rnor_.5.0</td>
</tr>
<tr>
<td>bosTau4</td>
<td>bos_taurus</td>
<td>UMD3.1</td>
</tr>
<tr>
<td>mm10</td>
<td>mus_musculus</td>
<td>GRChm38</td>
</tr>
<tr>
<td>hg19</td>
<td>homo_sapiens</td>
<td>GRCh37</td>
</tr>
</tbody>
</table>

If one of the previous data sets are chosen, the data sets will be downloaded (if they are not already locally). This data set contains the BWA index files and a gene annotation file.

It’s a pure function § 4.3.
7.5 Annotate

Given a file with aligned sequencing reads (ReadSet) and a list of genomic features (gff file), the function allows to annotate reads to each feature. An example:

```python
annotated = annotate(mapped, strand=false, mode={union}, keep_ambiguous=False)
```

**Argument**:

MappedReadSet

**Return**:

AnnotatedSet

**Arguments by value**:

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Required</th>
</tr>
</thead>
<tbody>
<tr>
<td>gff</td>
<td>String</td>
<td>✓</td>
</tr>
<tr>
<td>features</td>
<td>[ Symbol ]</td>
<td>x</td>
</tr>
<tr>
<td>mode</td>
<td>Symbol</td>
<td>x</td>
</tr>
<tr>
<td>keep_ambiguous</td>
<td>Bool</td>
<td>x</td>
</tr>
<tr>
<td>strand</td>
<td>Bool</td>
<td>x</td>
</tr>
</tbody>
</table>

The `gff` argument is required, unless it was used a data set provided by NGLess on the map (map section). It must be a valid path to a annotation file.

The argument `features` represents which features to keep, discarding everything else. If nothing is provided everything is considered to be important. The possible symbols are `{gene}`, `{exon}` and `{cds}`.

**Mode** is a string that represents the mode to handle reads overlapping more than one feature. The possible values for `mode` are `{union}`, `{intersection-strict}` and `{intersection-nonempty}` (default: `{union}`). For each read position is obtained features that intersect it, which are called 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.

The `keep_ambiguous` argument allows to decide whether to count reads that overlap with more than one feature. The possible values are False and True (default: False).

Argument `strand` represents whether the data is from a strand-specific and the possible values can be True or False (default: False). For False, a read is always overlapping with a feature independently of whether maps to the same or the opposite strand. For True, the read has to be mapped to the same strand as the feature.

7.6 Count

Function that allows to filter the counts of features. Example:

```python
counts = count(annotated, min=2)
```
Argument:
AnnotatedSet

Return:
AnnotatedSet

Arguments by value:

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Required</th>
</tr>
</thead>
<tbody>
<tr>
<td>counts</td>
<td>Symbol</td>
<td></td>
</tr>
<tr>
<td>min</td>
<td>Integer</td>
<td></td>
</tr>
</tbody>
</table>

The argument **counts** represents which features to keep, discarding everything else. The possible symbols are gene, exon and cds. If nothing is provided everything is considered to be important.

**Min** defines the minimum amount of overlaps a given feature must have, at least, to be kept (default: 0).

It’s a pure function § 4.3.

### 7.7 Subtrim

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

```python
read = substrim(read, min_quality=5)
```

Argument:
ShortRead

Return:
ShortRead

Arguments by value:

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Required</th>
</tr>
</thead>
<tbody>
<tr>
<td>min_quality</td>
<td>Integer</td>
<td></td>
</tr>
</tbody>
</table>

**Min_quality** parameter defines the minimum quality accepted for the sub-sequence (default: 0).

It’s a pure function § 4.3.

### 7.8 Write

Write function allows to write a NGLessObject to Disk. Different Types of NGLessObject are manipulated in different manners.
7.8.1 ReadSet

Argument:
ReadSet

Return:
Void

Arguments by value:

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Required</th>
</tr>
</thead>
<tbody>
<tr>
<td>ofile</td>
<td>String</td>
<td>✓</td>
</tr>
</tbody>
</table>

The argument ofile is a file path to where the content is written.

7.8.2 MappedReadSet

Argument:
MappedReadSet

Return:
Void

Arguments by value:

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Required</th>
</tr>
</thead>
<tbody>
<tr>
<td>ofile</td>
<td>String</td>
<td>✓</td>
</tr>
<tr>
<td>format</td>
<td>String</td>
<td>x</td>
</tr>
</tbody>
</table>

Format can have value \{bam\} or \{sam\} (default: \{sam\}).

7.8.3 AnnotatedSet

Argument:
AnnotatedSet

Return:
Void

Arguments by value:

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Required</th>
</tr>
</thead>
<tbody>
<tr>
<td>ofile</td>
<td>String</td>
<td>✓</td>
</tr>
<tr>
<td>format</td>
<td>String</td>
<td>x</td>
</tr>
<tr>
<td>verbose</td>
<td>Symbol</td>
<td>x</td>
</tr>
</tbody>
</table>
Format can have value `csv` or `tsv` (default: `tsv`).

Verbose allows to choose between writing a simplified or a verbose version of the results. Possible values are `yes` or `no` (default: `no`).

If a list of any of the previous mentioned data types is provided, the `ofile` argument must use an `{index}` in the template name to differentiate between the files in the list. For example for a list with two elements:

```python
ofile = "../samples/CountsResult{index}.txt"
```

would result in,

```
"../samples/CountsResult1.txt", "../samples/CountsResult2.txt"
```

7.9 Print

Print function allows to print a NGLessObject to IO.

**Argument:**

NGLessObject

**Return:**

Void

**Arguments by value:**

none

8 Omissions and Errors

Omissions and errors will be fixed in future versions of NGLess specification.
Appendix B

Documentation
CHAPTER
ONE

INSTALLATION

1.1 From source

1.1.1 Requirements

Before starting, make sure you have an Internet connection!

These are the required programs you must have installed

- GHC (version 7.6.3 or higher)
- Cabal (1.8.0.3 or higher)
- Git

Both GHC and Cabal can be installed via the haskell-platform package.

1.1.2 Update cabal

Cabal by default comes with an older version installed. Start by running:

$ cabal --version

If the version is equal or higher than 1.8.0.3 you are ready to install NGLess and no more steps are required! Otherwise continue this installation process which will update your cabal version. Start by running the following commands:

$ cabal update
$ sudo cabal install cabal-install --prefix=/usr

These commands require super user privilege and install the correct version of cabal in ‘/usr’ which is by default in $PATH. In case, you only have user privileges you can choose a --prefix anywhere else (that you have permissions) but make sure that is in your $PATH environment. After the installation, you should be ready to go. To be sure, run again:

$ cabal --version

Check if the new version is higher than 1.8.0.3.

If it is, you are ready to install NGLess.

If it isn’t, one of the following problems might be occurring

1. The path used as --prefix is not in your $PATH.
2. An older version of cabal is installed in some directory which comes first than ‘/usr/local’ in your $PATH variable.
1.1.3 Steps
Start by download latest NGLess version from GitHub:

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

Then download and configure all NGLess dependencies by running the following command:

```
$ make
```

This will take a while, so go ahead and make some tea! After the previous command is completed (without errors) you are ready to install ngless wherever you want it to be.

```
$ sudo make install prefix=dir (default dir is /usr/local)
```

(Note: `sudo` is only required when installing on a directory with super user privileges)

After this ngless is ready to use!

1.1.4 Options:
The following are options to the Makefile:

- `all` - builds NGLess
- `nglessconf` - downloads and configures all NGLess external dependencies
- `clean` - remove local files generated by compilation
- `distclean` - remove all local files generated by the compilation and all external downloaded dependencies
- `uninstall` - remove installed files. By default assumes installation in /usr/local, but prefix=dir can be passed
- `dist` - Create a distribution tar file for NGLess
- `check` - Execute NGLess tests

1.2 Binary
A self-contained NGLess distribution is also available for:

- OS X Mavericks (x86_64): `ngless-0.0.0-Darwin-x86_64.tar.gz`
- Linux (x86_64): `ngless-0.0.0-Linux-x86_64.tar.gz`

In this case, NGLess executable can be used directly and it’s located at `ngless-*/bin`.

1.3 FAQ
During 'make' an error was reported saying : 'curses.h: No such file or directory.'. How do I fix it?

You need to install the curses library which include routines for a terminal-independent method of updating character screens with reasonable optimisation. The fix depends on the Operating System you are currently in.

Ubuntu:

```
$ sudo apt-get install libncurses5-dev libncursesw5-dev
```

Fedora / RHEL / CentOS Linux:

```
$ yum install ncurses-devel ncurses
```

1.2 Binary
During ‘make’ an error was reported saying ‘Error: SSE2 instruction set not enabled’. How do I fix it?

This is a known problem when compiling the Burrows-Wheeler Aligner (BWA) tool when under a 32 bits operating system. To fix it, you have to change the Makefile inside the directory `bwa-0.7.7/` in

Line 3) CFLAGS= -g -Wall -O2 -msse -mmmx -msse2

Line 6) DFLAGS= -DHAVE_PTHREAD
NGLess provides two modes to generate scripts. One targets (programming) skilled users as every single character of the code as to be typed and the other approach uses a wizard to generate the script.

To access either one of this modes you have to do:

```bash
$ NGLess visualizemode -
```

**Note 1:** If port 8000 is already in use, you can change the destiny port by using `-p X` where `X` can be any port you wish.

**Note 2:** Argument `-` is used as this is the first run and there isn’t any NGLess results yet. It is a mode to open the web server without any data.

Now you can open your browser at `http://localhost:8000`. This will open the NGLess web server at a given port.

After the web server loads the page, you should see something similar to the next image.

![NGLess welcome page](image)

### 2.1 Example Description

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](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](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.
2.2 Create your script

Since the web server is now open you can start to create your script. Start by clicking in the tab Create Script from the navigation menu, as in the following image:

After clicking, a page as the following should be displayed.

As you can see, there are two ways to create your scripts. At the left there is a text editor that allows to edit and create your scripts and at the right is a wizard that generates a script with little to no effort.

Choose the mode that will make you feel more comfortable with.
2.3 Text Editor

2.3.1 Load FastQ file

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

Using the text editor, type:

```bash
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`.

```bash
$ ngless test.ngl
$ ngless visualizemode test -p 8000
```

Note: Make sure you don't have NGLess already running in that port.

Using the web server you can visualise key information about a data set. 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 analysing the plot we can see that the first 3 base pairs, on average, have the lowest quality (31.0). So, a good pre-process starts by removing the first 3 base pairs.

Feel free to explore all the available statistics.
### 2.3.2 Preprocess

For the preprocess we will:

- Remove the first 3 base pairs.
- Subtrim 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 in the Text Editor.

```python
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
```

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

### 2.3.3 Map

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

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

### 2.3.4 Annotate

We are only interested in the human genes so let’s annotate the map results with the only feature being genes. Since we used a genome provided by NGLess, we will also use the annotation provided by default:

```python
/* annotation features */
feats = [{gene}]
annotated = annotate(mapped, strand=false, mode={union}, ambiguity=false, features=feats)
```

### 2.3.5 Count & Write

Annotation will annotate the results but won’t store them. In order to count and save them you have to write the counts of the annotation to somewhere in your disk:

```python
/* write counts to disk */
counts = count(annotated)
write(counts, verbose={yes}, ofile="samples/CountsResult.txt")
```

### 2.3.6 Final Script

At the end, your Text Editor should have the following code:
Jump to section **Execute** to run the script and see the results.

## 2.4 Wizard

### 2.4.1 Load fastQ file

To load the file ‘SRR867735.fastq.gz’ you should click ‘add file’ which will open a box that you can type the file name.

Important to notice in the image also that the script is created in real time. This means that you can see the modifications while making them. You can now click in ‘Next Section’ to go to Pre-Process.

### 2.4.2 Preprocess

At the preprocess we are going to:

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

To do a), since we want to remove the first 3 base pairs, we need to make a left trim of 3. You can see the script changing while making modifications.
For b) we want to make a substrim and for that you should update the respective field value to 15.

For c) we want to make a discard and since this is a common operation our wizard as a special field for that:

Now, after filling all the values, the script displayed at the bottom should look as follows:

```bash
ngless '0.0'
in = fasta(['SRR867735.fasta.gz'])
preprocess(in) using (read):
    read = read[3:]
    read = substrim(read, min_quality=15)
if len(read) < 20:
    discard

n = map(in, reference='hg19')
```

Since the unique operation is not applied, click twice in ‘Next Section’ to jump to Map.

### 2.4.3 Map

Since the human genome is provided by default, you can simply fill the reference field with ‘hg19’:

Your script should look exactly like the script at the end of the previous image.

You can now click in ‘Next Section’ to go to Annotate.
2.4.4 Annotate

We are only interested in the human genes so let's annotate the map results with only the genes as features.

**Features to filter**

Choose between the possible genomic features the ones you're interested in.

- [ ] cds
- [ ] exon
- [x] gene

**Do not allow** ambiguity when deciding a feature.

**Keep ambiguity**

Do you want to count with the ambiguous results?

- [ ] yes
- [ ] no

Allow the matches to be in any strand, positive or negative.

**Strand Specific**

The genomic feature must have the same strand as the strand of the alignment?

- [ ] yes
- [ ] no

Your script should look exactly like this:

```python
ngless '0.8'

in = fastq(['SRR867735.fastq.gz'])
preprocess(in) using |read|
  read = read[3:]
  read = substring(read,min_quality=15)
  if len(read) < 28:
    discard
n = map(in,reference='hg19')
a = annotate(n,keep_ambiguous=False,strand=False,features=[[gene]])
```

You can now click in ‘Next Section’ to go to Count.

2.4.5 Count

We are only interested in counting genes. It's quite redundant choosing 'gene' again but while using the wizard this field is mandatory.

**Features to filter**

Choose between the possible genomic features the ones you're interested in for the result of the counts.

- [ ] cds
- [ ] exon
- [x] gene
You can now click in ‘Next Section’ to go to Write.

### 2.4.6 Write

In the write you should fill the file destiny as ‘CountResults.txt’ and your final script should look like the following image:

```
File Destiny
Where you want to store your count results?
Since you are using an array of data sets, the keyword '(index)' must be used on the filepath:

CountResults.txt
```

You can now click in ‘Next Section’ to complete the script generation.

### 2.4.7 Final Script

At the end you should see the final script:

```
ngless '0.0'
in - fastq(['SRR867735.fastq.gz'])
preprocess(in) using fread:
    read = read[3:]
    read = substrim(read,min_quality=15)
    if len(read) < 28: discard
m = map(in,reference='hg39')
a = annotate(m,keep_ambiguous=false,strand=false,features=\[\{gene\}\])
c = count(a,counts=\{\{gene\}\})
write(c,ofile='CountRes.txt')
```

Jump to section Execute to run the script and see the results.

### 2.5 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.

All other results can be accessed through the web server by doing. As you might already be running a web server from the previous execution, open a new web server at port 8080 or close the one used before:

```
$ ngless visualizemode test -p 8080
```

The results are in the following (sub)sections.
2.5.1 After quality control

As the function preprocess was used, quality control is generated.

It can be visualised at the tab ‘After QC’ as shown next:

As can be seen the quality has increased and the minimum quality is now 34. Also, the minimum and maximum ([min,max]) sequence decreased from [50,50] to [20,47] which implies a decrease in both the sequence length minimum and maximum, 30 and 3 respectively.

2.5.2 Counts

In order to access the top gene counts, you can use the ‘Visualize’ tab in the navigation menu.

You should be able to see a table with all results.

You should be able to see a list of all files at the column on the left. Click on the one named ‘CountResults’ that is representative of the annotation results of the script.

By clicking on the counts column you will be able to sort the counts in descending order. By default are shown 20 genes at a time, but you can define the amount to either 10, 25, 50 or 100.

If you sort in descending order and select to be displayed 10 results, you should be able to see the top 10 results with most counts. If everything went well they should be:

<table>
<thead>
<tr>
<th>Gene name</th>
<th>Counts</th>
</tr>
</thead>
<tbody>
<tr>
<td>ENSG00000210082</td>
<td>2901346</td>
</tr>
<tr>
<td>ENSG00000265150</td>
<td>182390</td>
</tr>
<tr>
<td>ENSG00000269900</td>
<td>179083</td>
</tr>
<tr>
<td>ENSG00000202198</td>
<td>175199</td>
</tr>
<tr>
<td>ENSG00000211459</td>
<td>165836</td>
</tr>
<tr>
<td>ENSG00000259001</td>
<td>116589</td>
</tr>
<tr>
<td>ENSG00000269028</td>
<td>98050</td>
</tr>
<tr>
<td>ENSG00000187608</td>
<td>95884</td>
</tr>
<tr>
<td>ENSG00000126709</td>
<td>94874</td>
</tr>
<tr>
<td>ENSG00000067225</td>
<td>82878</td>
</tr>
</tbody>
</table>

Also if you want to edit the file directly you can by opening the file ‘CountResults.txt’ with your preferred text editor.

2.5. Execute
3.1 Genome references available

NGLess provides archives containing data sets of organisms. It’s 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:

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
<th>Assembly</th>
</tr>
</thead>
<tbody>
<tr>
<td>sacCer3</td>
<td>saccharomyces_cerevisiae</td>
<td>R64-1-1</td>
</tr>
<tr>
<td>ce10</td>
<td>caenorhabditis_elegans</td>
<td>WBcel235</td>
</tr>
<tr>
<td>dm3</td>
<td>drosophila_melanogaster</td>
<td>BDGP5</td>
</tr>
<tr>
<td>canFam2</td>
<td>canis_familiaris</td>
<td>CanFam3.1</td>
</tr>
<tr>
<td>rn4</td>
<td>rattus_norvegicus</td>
<td>Rnor_5.0</td>
</tr>
<tr>
<td>bosTau4</td>
<td>bos_taurus</td>
<td>UMD3.1</td>
</tr>
<tr>
<td>mm10</td>
<td>mus_musculus</td>
<td>GRCm38</td>
</tr>
<tr>
<td>hg19</td>
<td>homo_sapiens</td>
<td>GRCh38</td>
</tr>
</tbody>
</table>

3.2 Data Set Structure

The archives provided by NGLess contain BWA index files, the genome reference file and a gene annotation file. These archives are all created using version 75 of Ensembl.

```
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).
3.3 Automatic installation

It is possible to use the provided data sets directly on NGLess by simply typing its name. If it isn’t already available locally (either on user or on root mode), then the archive is downloaded and the script execution is paused until the download is complete and the archive installed.

The archive is installed on User mode and so at home/ngless/organisms.

3.4 Manual installation

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

To install locally (organism bos taurus), is as simple as:

$ ngless --install-reference-data bosTau4

And to install globally is:

$ 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.

Note: Can be used flag –i instead of –install-reference-data.
4.1 Launch

In order to visualize the script results a web server is launched with the following command.

```
$ ngless visualizemode <script_location.ngless_output>
> Launching WebServer at: 8000
> You can access it at: http://localhost:8000
```

To change the port from the default 8000, as to be used the flag `-p [int]` or `--port=[Int]`.

If everything went correctly, you should see something like this.

4.2 Visualize Runs

All prior executions of NGLess, for a script, can be consulted and are organised in a drop down list fashion.
Each box has information outside as the data set name that we used and the date and time of the execution.

If we open one box, there can be seen three main things:

- The executed script
- The quality control at the beginning
- Quality controls after the pre-processing (this one being optional).

Is crucial to save the script that lead to some results. For that reason, scripts are always associated with the results that they generate, allowing this way to easily reproduce an experiment. The script can be consulted has shown in the next figure.

![Script Example](image)

### 4.3 Before Quality Control

This quality control is always present and it allows to visualise the quality and basic info about a given data set.

The basic information provided is:

- The original file path
- The percentage of guanine and cytosine in the whole data set.
- Encoding prediction.
- The number of sequences.
- The minimum and maximum sequence length of the entire data set.

Not only basic information is supplied, but also statistical calculations are made. These statistics are made in relation to each base pair, and are presented in an interactive plot that allows show/hide these metrics.

The used statistical measures are in relation to each base pair and are:

- Mean
- Median
- Lower Quartile (25%)
- Upper Quartile (75%)

The Quality control can be accessed in the web interface through the tab **before QC** and a vertical menu allows to choose which dataset in question. Can be seen an example next:
The plot can be adjusted to show one statistic at a time, and the plot limits adapt to the presented values. An image of only the upper quartile shown looks like the following:

### 4.4 After Quality Control

The quality control, in terms of statistics, is very similar to the represented at section Before Quality Control. The main different is a correlation in the results before vs after.
At the table, the column summary represents the difference in %GC, number of sequences and minimum/maximum sequence length from the before QC to after QC. Encoding can’t change and for that reason remains the same.

From the results can be seen that:

- Decreased 0.005 in %GC.
- 802204 sequences were removed
- the minimum and maximum sequences decreased 30 and 3 respectively.

### 4.5 Visualise results

This option allows to visualise the results from the annotation. It’s possible to filter by a given gene id and sort in ascending or descending order by the number of counts. Is also possible to create a top 10,25,50,100 by choosing one of the possible values from the menu at the bottom right.

Can be seen, in the previous figure, a top 10 example, named ‘CountResults’, where the features are only genes with the column counts sorted descending.

### 4.5. Visualise results

B20
5.1 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.

5.2 Comments

5.2.1 Explicative

Start with `#` or `//` and end at the end of the line.

```
i = 10 // This variable is used to explain Explicative comments
```

5.2.2 Operational

Start with `/*` and end with `*/`. Can’t be nested.

```
/*
  This comment is used to explain operational comments.
*/
i = 10
```

5.3 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.

### 5.3.1 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 (\).

### 5.3.2 Integer
Integers are specified as decimals [0-9]+ or as hexadecimals 0x[0-9a-fA-F]+. They are non negative, but can be negative through the use of the operator (-).

### 5.3.3 Boolean
Booleans are denoted as the word true or false, with the first letter in upper or lower case.

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

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

Assignment is performed with = operator:

```plaintext
variable = value
```

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

### 5.5 Operators

#### 5.5.1 Unary
The operator (-) returns the symmetric of its integer argument.

The operator len returns the length of a List or a ShortRead.

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

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

#### 5.5.3 Indexation
Can be used to access only one element or a range of elements in a List or 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:

<table>
<thead>
<tr>
<th>x[: ]</th>
<th>returns from position 0 until length of variable x</th>
</tr>
</thead>
<tbody>
<tr>
<td>x[10:]</td>
<td>returns from position 10 until length of variable x</td>
</tr>
<tr>
<td>x[:10]</td>
<td>returns from position 0 until 10</td>
</tr>
</tbody>
</table>

### 5.4. Variables
5.6 Conditional

If the expression, following the word *if*, is *true* then the block that follows the ‘:’ is executed.

```python
if true:
    val = 10  # will be executed
```

If the expression returns *false*, is present the reserved word *else* and delimiter ‘:’, the else block is executed.

```python
if 5 > 10:
    val = 10
else:
    val = 20  # will be execute
```

If is returned *false* and the word *else* is not present, nothing happens.

5.7 Functions

Functions are called with parentheses:

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

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

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

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

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

There is no possibility of defining new functions. Only the built-in functions are available.

5.7.1 Pure functions

Functions that their result must be assigned to some variable are called pure functions. They are comprised of:

- unique
- substrim
- map
- count

5.8 Auto-comprehension

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

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

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

Function to load, one or more, FastQ files. An example:

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

**Argument:**
String

**Return:**
ReadSet

**Arguments by value:**
none

The only compression method supported for the data sets is `gzip` (.gz).
The encoding prediction is built on the lowest ASCII character of the fastQ file.
When loading a data set, quality control is carried out and statistics can be visualised in a graphical user interface (GUI).

Simple statistics calculated are percentage of guanine and cytosine (%GC), encoding, number of sequences and minimum maximum sequence length. The more complex statistics calculated are the mean, median, lower quartile and upper quartile for each position of the base pairs.

**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:

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

**Argument:**
ReadSet

**Return:**
ReadSet
Arguments by value:

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Required</th>
<th>Default Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>max_copies</td>
<td>Integer</td>
<td>no</td>
<td>2</td>
</tr>
</tbody>
</table>

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

Is considered a copy: ShortReads with the same sequence regardless of quality and identifier.

**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. An example:

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

**Argument:**

ReadSet

**Return:**

Void

**Arguments by value:**

none

This function also performs quality control on its output.

**Map**

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

```python
mapped = map(input, reference='sacCer3')
```

**Argument:**

ReadSet

**Return:**

MappedReadSet

**Arguments by value:**

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Required</th>
<th>Default Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>reference</td>
<td>String</td>
<td>yes</td>
<td>-</td>
</tr>
</tbody>
</table>

The argument `reference` can either be a path to a data set or the `name` of a provided data set by NGLess. The provided data sets of NGLess are:
The argument `reference` can either be a path to a data set or the name of a NGLess provided data set. Provided data sets of NGLess are:

### Annotate

Given a file with aligned sequencing reads (ReadSet) and a list of genomic features (GFF file), the function allows to annotate reads to each feature. An example:

```python
annotated = annotate(mapped, strand=false, mode={union}, keep_ambiguous=false)
```

#### Argument:

**MappedReadSet**

#### Return:

**AnnotatedSet**

#### Arguments by value:

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Required</th>
<th>Default value</th>
</tr>
</thead>
<tbody>
<tr>
<td>gff</td>
<td>String</td>
<td>yes*</td>
<td>-</td>
</tr>
<tr>
<td>features</td>
<td>[ Symbol ]</td>
<td>no</td>
<td>{gene}</td>
</tr>
<tr>
<td>mode</td>
<td>Symbol</td>
<td>no</td>
<td>{union}</td>
</tr>
<tr>
<td>keep_ambiguous</td>
<td>Bool</td>
<td>no</td>
<td>false</td>
</tr>
<tr>
<td>strand</td>
<td>Bool</td>
<td>no</td>
<td>false</td>
</tr>
</tbody>
</table>

The `gff` argument is required, unless a known reference was used for mapping.

`features` represents which features to keep, discarding everything else. If nothing is provided, everything is considered to be significant. Possible symbols are `{gene}`, `{exon}`, and `{cds}`.

`Mode` is a symbol which dictates how to handle reads overlapping more than one feature. 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.

The `keep_ambiguous` argument is an opportunity to decide whether to annotate reads that overlap with more than one feature.

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.
Count
Function that allows to filter the counts of features. Example:

```python
counts = count(annotated, min=2)
```

**Argument:**
AnnotatedSet

**Return:**
AnnotatedSet

**Arguments by value:**

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Required</th>
<th>Default Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>counts</td>
<td>[ Symbol ]</td>
<td>no</td>
<td>-</td>
</tr>
<tr>
<td>min</td>
<td>Integer</td>
<td>no</td>
<td>0</td>
</tr>
</tbody>
</table>

The argument `counts` represents which features to keep, discarding everything else. Possible symbols are gene, exon and cds. If nothing is provided everything is considered to be important.

**Min** defines the minimum amount of overlaps a given feature must have, at least, to be kept (default: 0).

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

```python
read = substrim(read, min_quality=5)
```

**Argument:**
ShortRead

**Return:**
ShortRead

**Arguments by value:**

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Required</th>
<th>Default Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>min_quality</td>
<td>Integer</td>
<td>no</td>
<td>0</td>
</tr>
</tbody>
</table>

**Min_quality** parameter defines the minimum quality accepted for the sub-sequence (default: 0).

Write
Write function allows to write a NGLessObject to Disk. Different Types of NGLessObject are manipulated in different manners.

**ReadSet**

**Argument:**
ReadSet
Return:
Void

Arguments by value:

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Required</th>
<th>Default Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>ofile</td>
<td>String</td>
<td>yes</td>
<td></td>
</tr>
</tbody>
</table>

The argument `ofile` is a file path to where the content is written.

**MappedReadSet**

**Argument:**

MappedReadSet

**Return:**

Void

Arguments by value:

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Required</th>
<th>Default Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>ofile</td>
<td>String</td>
<td>yes</td>
<td></td>
</tr>
<tr>
<td>format</td>
<td>String</td>
<td>no</td>
<td>{sam}</td>
</tr>
</tbody>
</table>

Format can have value `{bam}` or `{sam}` (default: `{sam}`).

**AnnotatedSet**

**Argument:**

AnnotatedSet

**Return:**

Void

Arguments by value:

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Required</th>
<th>Default Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>ofile</td>
<td>String</td>
<td>yes</td>
<td></td>
</tr>
<tr>
<td>format</td>
<td>String</td>
<td>no</td>
<td>{tsv}</td>
</tr>
</tbody>
</table>

Format can have value `{csv}` or `{tsv}` (default: `{tsv}`).

If a list of any of the previously mentioned data types is provided, the `ofile` argument must use an `{index}` in the template name to differentiate between the files in the list. For example for a list with two elements:

```
ofile = "../samples/CountsResult{index}.txt"
```

would result in,

```
../samples/CountsResult1.txt", "./samples/CountsResult2.txt"
```

**Print**

Print function allows to print a NGLessObject to IO.
**Argument:**
NGLessObject

**Return:**
Void

**Arguments by value:**
none
These are the required programs you must have installed to test the ngless tool

- GHC (version 7.6.3 or higher)
- Cabal (1.8.0.3 or higher)
- Git
- Python (with numpy and matplotlib)

**Install Python libraries**

Depending on the operating system, there are multiple ways to install the required packages. **Make sure you have them installed before doing anything else!**

In case they aren’t, there are a few steps depending on the operating system that you can use to install them.

**7.0.1 Ubuntu**

```
$ sudo apt-get install build-essential python2.7-dev python-numpy python-matplotlib
```

**7.0.2 RedHat, Fedora, CentOS**

```
$ sudo yum groupinstall "Development Tools"
$ sudo yum install python-devel numpy python-matplotlib
```

**7.0.3 Mac**

Using **macports**, can be installed by running the following command:

```
$ sudo port install py27-numpy py27-matplotlib
```

**Steps**

Start by download latest NGLess version from GitHub:

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

Then compile and run nglesstest by executing the following command:

```
$ make check
```