University of Puerto Rico
Mayagüez Campus
Department of Electrical and Computer Engineering

Class Project
A PL for Computational Biology
BioL++
Phase 3: Final Report

Angel Rodriguez
Joshua Bonilla
Kelvin Garcia

ICOM 4036 Sec. 096
Prof. W. Rivera

May 18, 2019

Table of Contents

Introduction

Computational biology is a broad and expansive field of study centered mostly on the storage, processing and analysis of biological data pertinent to many fields such as genetics, molecular biology and biophysics, among others. It uses applied mathematics and statistics to aid in the analysis of large amounts of data, often requiring a way to output the results onto graphs and other displays to aid in the visualization of the data and results of the analyses. The field is intricately linked to computer science as it makes heavy use of computers to acquire, store and process all the data, as well as designing algorithms to process it and analyze it.

The main motivation comes from the fact that many of the tasks performed by computational biologists are repetitive and require several steps to achieve the final result. This makes designing a dedicated programming language an ideal solution. The goal of this language is to streamline the workflow and condense many of these tasks into one or two simple instructions which will automatically output the results as desired. Another reason behind creating such a language is that it allows biologists with little to no programming knowledge or experience to easily analyze their own data without having to rely on computational biologists to do it for them. This way they can benefit immensely without having to spend months trying to learn how to program or learn a complex programming language. Using simple and straightforward functions, anyone can learn how to use this programming language in just one afternoon.

Language Tutorial

In order to use BioL++ in your computer, the following requirements must be met:

- Python 3.x – The latest, preferably 64-bit version.
- A Python virtual environment and/or an integrated development environment (i.e. PyCharm)

- Python PIP
- Biopython, scipy, numpy, matplotlib and pylab libraries.

Once these conditions are met, you can clone or download the repository at:

https://github.com/rafo23/biolpp

Then you can import the project into PyCharm or any IDE of your choice and run it from there, or just use the command prompt/terminal to run the program.

To view the tutorial video, please visit the following link:

https://www.youtube.com/watch?v=1ArjKt102Ao

Reference Manual

Grammar

```
<statement> -> VARLIST

<statement> -> ID EQUALS ID | ID EQUALS RESULT

<statement> -> ID

<statement> -> <result>

<result> -> method_one | method_two | method_three

<method_one> -> [function1] LPAR [ID | STRING | INT] RPAR

        | [function2] LPAR [ID | STRING] COMMA [STRING | DTYPE | RTYPE] RPAR

        | [function3] LPAR STRING COMMA STRING COMMA STRING RPAR

<method_two> -> SEQ LPAR STRING RPAR

        | HAMDIS LPAR ID COMMA ID RPAR

        | RECUR LPAR INT COMMA INT RPAR

<method_three> -> DRAW LPAR INT COMMA STRING RPAR

<empty> -> " "

<function1> -> PRINT | COMP | RCOMP | TRANSC | RTRANSC | CTABLE | WRITE

        | GCCON | RNAINF | RNAINF2 | ORF | COMPF | RCOMPF | TRANSCF | RTRANSCF
```

| PROTW | PROTINFER

<function2> -> TRANSL | READ | MOTIF | PUNNETT

<function3> -> WPUNNETT

Language Functions

Make Sequence - seq(string)

seq(string)

Takes as a parameter a string containing the value of the sequence to be denoted.

Example: *myseq = seq('GATGGAACTTGACTACGTAAATT')*

myseq2 = seq('GAUGGAACUUGACUACGUAAAUU')

Denotes a sequence named *myseq* containing the DNA sequence GATGGAACTTGACTACGTAAATT and a sequence named *myseq2* containing the RNA sequence *GAUGGAACUUGACUACGUAAAUU*

***NOTE** - The names of myseq and myseq2 are not fixed, meaning that they are open to the user's discretion*

Print File - print(string)

print(string)

Takes as a parameter a string value corresponding to the path of the desired FASTA file and prints the contents of such file to the console.

Example: *print('IOFiles/file.txt')*

Prints the file named *file.txt* contained in the *IOFiles* directory

Complement - comp(id)

comp(id)

Takes as a parameter an id corresponding to a DNA sequence previously initialized and returns the complement of such sequence.

Example: a = *comp(myseq)*

Assigns *a* the value of the complement of the sequence *myseq*

Reverse Complement - rcomp(id)

rcomp(id)

Takes as a parameter an id corresponding to a DNA sequence previously initialized and returns the reverse complement of such sequence.

Example: a = *rcomp(myseq)*

Assigns a the value of the reverse complement of the sequence *myseq*

DNA Transcription - transc(id)

transc(id)

Takes as a parameter an id corresponding to a DNA sequence previously initialized and returns the transcription of such sequence.

Example: *a = transc(myseq)*

Assigns a the value of the transcription of the sequence *myseq* from DNA to

RNA

RNA Transcription - rtransc(id)

rtransc(id)

Takes as a parameter an id corresponding to a RNA sequence previously initialized and returns the transcription of such sequence.

Example: *a = rtransc(myseq2)*

Assigns *a* the value of the transcription of the sequence *myseq2* from RNA to DNA

Print Codon Table - ctable(int)

ctable(int)

Takes as a parameter an integer (1 or 2) corresponding to the desired codon table and prints it in the console

Example: *ctable(1)*

ctable(2)

Prints a DNA and an RNA codon table respectively

***NOTE** - The integers 1 and 2 are fixed, meaning that providing any other value would result in a system error*

Translate DNA/RNA - transl(id, type)

transl(id, type)

Takes as a parameter an id corresponding to a DNA or RNA sequence previously initialized and a the type (dna or rna) and returns the translation of such sequence

Example*: a = transl(myseq, dna)*

b = transl(myseq2, dna)

Assigns *a* and *b* the value of the translation of the DNA and RNA sequences *myseq* and *myseq2*, respectively.

Read from File - read(string, string)

read(string, string)

Takes as parameters 2 strings representing the name of the sequence to be read and the FASTA file it is read from and returns the sequence with the name provided

Example: *a = read('Seq 1', 'IOFiles/file.txt')*

Assigns a the value of the sequence named *Seq 1* contained in the FASTA file

named *file.txt* contained in the *IOFiles* directory

Write to File - write(id, string)

write(id, string)

Takes as parameter an id and a string corresponding to the value to be written and the file in which it will be written.

Example: *write(myseq, 'IOFiles/myfile'*

Outputs a file named *myfile.txt* containing the value of the sequence *myseq*

GC Content - gccon(string)

gccon(string)

Takes as a parameter a string corresponding to the path of a FASTA file

Example: *gccon('IOFiles/file.txt')*

Prints the value of the GC Content of the sequence contained in the file *file.txt*

RNA Inferring - rnainf(id)

rnainf(id)

Takes as a parameter an id corresponding to a previously initialized RNA sequence and returns the RNA Inference of such sequence

Example: *a = rnainf(myseq2)*

Assigns *a* the value of the RNA inference of the sequence *myseq2*

RNA Inferring File - rnainf2(string)

rnainf2(string)

Takes as a parameter a string corresponding to the path of a file containing an RNA sequence in FASTA format and returns the RNA Inference of such sequence

Example: *rnainf2('IOFiles/file.txt')*

Prints the value of the RNA inference of the sequence contained in the file *file.txt* in the *IOFiles* directory

Open Read Frame - orf(string)

orf(string)

Takes as a parameter a string corresponding to the path of a file containing an DNA sequence in FASTA format and returns the Open Read Frame of such sequence

Example: *orf('IOFiles/file.txt')*

Prints the value of the Open Read Frame of the sequence contained in the file *file.txt* in the *IOFiles* directory

Complement File - compf(string)

compf(string)

Takes as a parameter a string corresponding to the path of a file containing a sequence in FASTA format and returns the complement of such sequence

Example: *compf('IOFiles/file.txt')*

Prints the value of the complement of the sequence contained in the file *file.txt* in the *IOFiles* directory

Reverse Complement File - rcompf(string)

rcompf(string)

Takes as a parameter a string corresponding to the path of a file containing a sequence in FASTA format and returns the reverse complement of such sequence

Example: r*compf('IOFiles/file.txt')*

Prints the value of the reverse complement of the sequence contained in the file *file.txt* in the *IOFiles* directory

DNA Transcription File- transcf(string)

transcf(string)

Takes as a parameter a string corresponding to the path of a file containing a DNA sequence in FASTA format and returns the transcription of such sequence

Example: transc*f('IOFiles/file.txt')*

Prints the value of the transcription of the sequence contained in the file *file.txt* in the *IOFiles* directory

RNA Transcription File - rtranscf(string)

r***transcf(string)***

Takes as a parameter a string corresponding to the path of a file containing a RNA sequence in FASTA format and returns the transcription of such sequence

Example: rtransc*f('IOFiles/file.txt')*

Prints the value of the transcription of the sequence contained in the file *file.txt* in the *IOFiles* directory

Protein Weight - protw(id)

protw(id)

Takes as a parameter an id corresponding to a previously initialized protein sequence and returns the weight of such protein according to a monoisotopic mass table

Example: a = *protw(myprot)*

Assigns *a* the weight of the protein *myprot*

Motif Interval - motif(id, string)

motif(id, string)

Takes as a parameter an id corresponding to a previously initialized sequence and a string corresponding to the splice of the sequence to which calculate the motif interval

Example: *a = motif(myseq, 'GAT')*

Assigns *a* the value of the Motif Interval of the splice GAT in the sequence *myseq*.

Write Punnett - punnett(string, string)

punnett(string, string)

Takes as parameters two strings corresponding to the alleles needed to for a table, separated by spaces, and outputs a table to the console along with the probability of each combination

Example: *punnett('Aa Bb', 'Cc Dd')*

Outputs in console the punnett table for the alleles *Aa Bb* and *Cc Dd*

Write Punnett File - wpunnett(string, string, string)

wpunnett(string, string, string)

Takes as parameters two strings corresponding to the alleles needed to for a table and a third string corresponding to the path of a file to be created and written with the Punnett table

Example: *wpunnet('Aa Bb', 'Cc Dd' ,'IOFiles/myfile')*

Writes the punnett table for the alleles *Aa Bb* and *Cc Dd* in the file *myfile.txt* contained in the *IOFiles* directory

Protein Inference - protinfer(string)

protinfer(string)

Takes as a parameter a string corresponding to the value of the weight of the protein to be inferred

Example: a = *protinfer('3524.8542 3710.9335 3841.974 3970.0326 4057.0646')*

Assigns a the inferred protein

Hamming Distance - hamdis(id, id)

hamdis(id, id)

Takes as a parameter two ids corresponding to two previously initialized sequences and returns the Hamming Distance between them

Example: *a = hamdis(myseq, myseq1)*

Assigns a the value of the hamming distance between *myseq* and *myseq1*

Recurrence - recur(int, int)

recur(int, int)

Takes as a parameter two integers and returns the recurrence interval

Example: a = recur(5, 7)

Assigns a the value of the recurrence interval between 5 and 7

Draw Phylogenetic Tree - drawtree(int, string)

drawtree(int, string)

Takes as parameter an int (1 or 2) corresponding to the method to output the tree contained in the second parameter, which indicates a file in xml format

Example: drawtree(1, 'IOFiles/tree.xml')

drawtree(2, 'IOFiles/tree.xml')

Draws the tree contained in the tree.xml file on the console and on the pylab view

***NOTE** - The integers 1 and 2 are fixed, meaning that providing any other value would result in a system error*

Language Development

Translator Architecture

The language was built using the Python programming language as well as the Python library PLY, which combines both a lexer and a parser. The language runs directly in the terminal where commands can be entered and executed immediately, with no need to write out the code first and then compile and run. This makes the language more versatile and easier to use and test. The following figure shows the basic structure of the language.

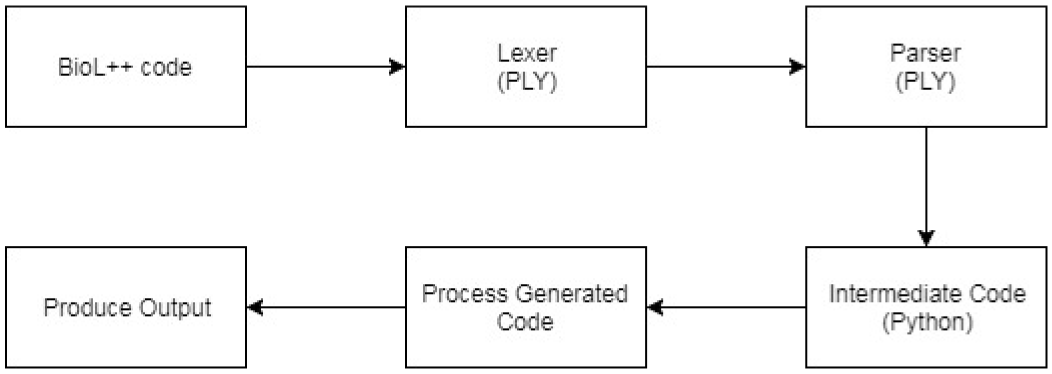

Figure 1 Translator Architecture for BioL++

As seen in Figure 1, the user inputs the BioL++ code into the terminal where it is passed on to the lexer. This generates the different tokens which are then passed on to the parser which matches the correct grammar rule. If no errors are found in the syntax, the underlying intermediate code in Python is executed to produce the requested output, which is then displayed to the user in the terminal.

Module Interfacing

The entire project was developed with modularity in mind. All aspects are separated based on their functionality. The main program is responsible for instantiating the parser and the loop to retrieve user input from the terminal window. The input is sent to the parser for analysis. Inside the parser, the lexer module is referenced and the token list defined there is obtained. Based on the syntax, grammar rules and the specific function that was input, the parser either throws a syntax error to the user, or on a syntactically correct statement calls the respective algorithm in the algorithms module to process the data and produce the desired output, which is then displayed in the terminal or saved to a file on the computer.

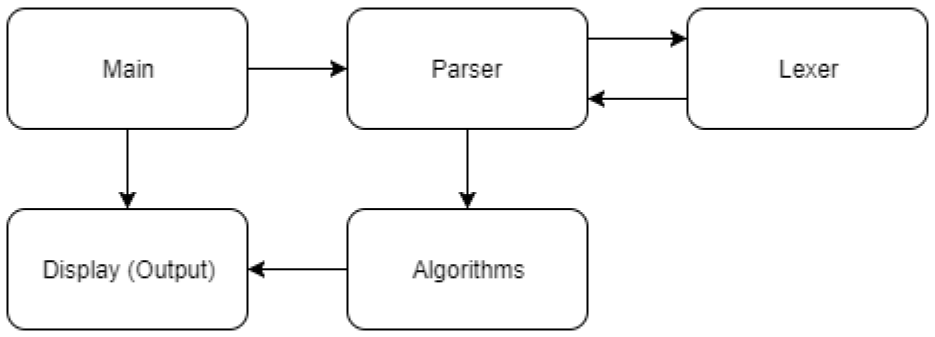

Figure 2 Module interfacing

The algorithms module contains our own implementations as well as external third-party libraries. The main library is *Biopython*, which is used in combination with *pylab* to read and draw the phylogenetic trees. This library allows us to draw an ASCII representation of the tree directly in the terminal as well as an actual image which can be modified and saved to the machine using the *matplotlib* viewer.

Software Development Environment

The entire project development was done with the help of the following programs/applications: JetBrains PyCharm, PIP/virtualenv and GitHub Desktop. PyCharm is a free integrated development environment (IDE) tailored for Python development and contains a plethora of tools and aids to help with coding in Python. It provides tools such as syntax highlighting and code completion, as well as project management and debugging tools. PIP is Python's own package manager that allows one to download and install packages and libraries seamlessly to one's machine through the terminal window. Virtualenv allows one to create a separate Python environment for development which only contains the necessary packages and the required versions to properly run the application. The final component is GitHub Desktop which greatly simplifies teamwork and version control. It allows groups of developers to push their work and retrieve updates done by others and manage the work progress.

Test Methodology

A separate test program was written to make sure all the algorithms worked as they were supposed to before actually getting into the language itself. The language on its own can be easily tested by running the commands directly on the terminal and comparing the output with the expected result obtained manually or from a trusted source. Below is the test program used to verify the algorithms separately before being incorporated into the project.

```
import biolpp_algorithms as bio

def main():
    bio.mendel_table_write('Aa Bb'.split(' '), 'Cc Dd'.split(' '), "out")
    # bio.mendel_table('Aa Bb'.split(' '), 'Cc Dd'.split(' '))
    # print(bio.rna_inferring('MA'))
    # print(bio.rna_inferring(bio.read_seq('HSBGPG Human gene for bone gla
protein (BGP)','file.txt')))
    # print(bio.dna_to_rna('GATGGAACTTGACTACGTAAATT'))
    # print(bio.rna_to_dna('GAUGGAACUUGACUACGUAAAUU'))
    # bio.dna_to_rnaFile('file.txt')
    # bio.rna_to_dnaFile('file.txt')
    # print(bio.complement_dna('GATGGAACTTGACTACGTAAATT'))
    # print(bio.recurrence(10, 12))
    # print(bio.to_protein('AGC', 'dna'))
    #
print(bio.to_protein('AUGGCCAUGGCGCCCAGAACUGAGAUCAAUAGUACCCGUAUUAACGGGUGA',
'rna'))
    # print(bio.hamming_distance('GAGCCTACTAACGGGAT', 'CATCGTAATGACGGCCT'))
    # # print(bio.read_fasta('file.txt'))
    # bio.gc_content('file.txt')
    # print(bio.motif_interval('ACGTACGTACGTACGT', 'GTA'))
    # bio.print_CDT('dna')
    # bio.print_CDT('rna')
    # bio.complement_dna_file('file.txt')
    # bio.open_read_frame('file.txt')
    # print(bio.prot_weight("S"))
    # bio.phylogen('tree.xml', 'console')
    # bio.phylogen('tree.xml', 'pylab')
    # print(bio.prot_infer("""3524.8542 3710.9335 3841.974 3970.0326
4057.0646"""))
    # seq = bio.read_seq('HSGLTH1 Human theta 1-globin gene', 'file.txt')
if __name__ == '__main__':
    main()
```

The lexer was tested by hardcoding different expressions and verifying the console output to make sure the lexer correctly tokenized the expression. Below is the sample test program which is located in the lexer module itself.

```
# Lexer

lexer = lex.lex()

# Tester

# lexer.input("   bseq abc = btree .read()")
#
# while True:
#     tok = lexer.token()
#     if not tok:
#         break
#     print(tok)
```

Once these are tested and proven to work, the next step is to test the entire language in the terminal since there is no way to test the parser separately and at this point the entire structure of the language is already done. As was mentioned previously, the test methodology consisted of running the commands in the terminal for which we already knew the result. We compared the output and made sure it matched. Trying to break the language is the best way to test it and make sure it is robust. Throwing in syntax errors and mismatched data types helps to confirm it is working in order.

Test Programs

The following is a sample program that demonstrates the major functionalities of the language:

```
BIOL++ >>> print('IOFiles/file.txt')
BIOL++ >>> a = seq('GATGGAACTTGACTACGTAAATT')
BIOL++ >>> a
BIOL++ >>> b = comp(a)
BIOL++ >>> b = transc(a)
BIOL++ >>> b = transl(a, dna)
BIOL++ >>> b = read('Seq 1', 'IOFiles/file.txt')
BIOL++ >>> write(a, 'IOFiles/myfile')
```

```
    BIOL++ >>> punnett('Aa Bb', 'Cc Dd')
```

Conclusions

After finishing the language implementation, we can really appreciate how much more simplified and streamlined the workflow is. This was one of the main goals of implementing this language, to make it easier and more accessible for people with limited programming experience as well as advanced users who benefit from the streamlining of tedious tasks to save time. Comparing the lines of code needed in some general-purpose language to obtain the same result as in BioL++, we see a drastic reduction. BioL++ supports the most common operations done on a day-to-day basis by most biologists, most of which are tedious and involve analyzing large datasets. Our language takes care of this by allowing most operations to be tied to a single line of code which then neatly outputs the results as desired. Intermediate results can be saved to variables to allow further operations to be performed. Most functions only require one or two parameters which means the language can be learned quickly and requires basically no memorization to perform even the most complex operations. We can say that we met our goal of making a language that is simple and accessible.