National University of Singapore
School of Computing
CS1010S: Programming Methodology
Semester I, 2022/2023

Side Quest 11.1
DNA using Data-Directed Programming

Release date: 13th October 2022
Due: 20th October 2022, 23:59

Required Files

- codon_mapping.csv
- sidequest11.1-template.py

Information

In this mission, we continue our investigation into the micro world of Molecular Biology. We realise that DNA, RNA and protein all encode related information, and store that information as strings. Yet, we need a way to distinguish a string that represents DNA from one that represents RNA or protein.

Since a string is used to represent the different types of data, we develop a tagging system to differentiate the types. Next, we use data-directed programming to examine the tags and retrieve correct functions from an operation table to operate on the data.

This mission consists of **three** tasks.

Overview of Molecule Types and Tagging

We wish to use data-directed programming to easily convert between DNA, RNA and protein types.

Data-directed programming involves tagging each data according to their types, and examining the tag of the data to find the correct operation from a look-up table when applying operations on tagged-data. We will be using the dictionary `OPERATION_TABLE` to store all our operations.

We introduce the concept of a Tagged-Data, where our data is tagged with a tag-type. You are to decide the underlying representation of a Tagged-Data in **Task 1**. We will be working with THREE types of tagged data, as follows:

Tagged-DNA A tagged DNA is a DNA strand in the 5' to 3' direction, tagged with the string tag-type `"dna"`.

Tagged-RNA A tagged RNA is an RNA strand in the 5' to 3' direction, tagged with the string tag-type `"rna"`.

Tagged-Protein A tagged protein is a string of amino acids (consisting of the 20 unique amino acids characters + 1 stop character) tagged with the string tag-type `"protein"`.

Task 1: Tagged-Data ADT (4 marks)

Before populating our `OPERATION_TABLE` with the various operations, we will first need to prepare our data for tagging. In this task, we create our Tagged-Data abstract data type.

(a) Define the function `tag` that accepts as inputs: the tag-type and the data. The function should return a Tagged-Data abstract data type. You are to decide the implementation. **(2 marks)**

(b) Define the function `get_tag_type` that accepts as input: a Tagged-Data. The function should return the tag-type of the given Tagged-Data. **(1 mark)**

(c) Define the function `get_data` that accepts as input: a Tagged-Data. The function should return the data of the given Tagged-Data. **(1 mark)**

Note: Once you are done with **Task 1**, you should use your constructors and accessors `tag`, `get_tag_type` and `get_data` to work with Tagged-Data.

```
### TAGGING DATA ###
dna            = "TTACTCCATATATCGCCGTGCCAT"
rna            = transcribe(dna)
protein        = translate(rna)
tagged_dna     = tag("dna", dna)
tagged_rna     = tag("rna", rna)
tagged_protein = tag("protein", protein)

get_tag_type(tagged_dna)     # 'dna'
get_tag_type(tagged_rna)     # 'rna'
get_tag_type(tagged_protein) # 'protein'

get_data(tagged_dna) == dna         # True
get_data(tagged_rna) == rna         # True
get_data(tagged_protein) == protein # True
```

Operation Table ADT

At this juncture, we are ready to populate our operation table. The table will hold various functions that can operate on different types of data. We wish to support operations that will allow for conversion between **Tagged-DNA**, **Tagged-RNA** and **Tagged-Protein**. We will also include functions to check if two Tagged-Datas are of same information flow (DNA → RNA → protein).

You are provided with the functions `put_op` and `get_op` to modify and access `OPERATION_TABLE`. You do not need to be concerned about how they are implemented, but you should understand how to use them. You should only interact with `OPERATION_TABLE` via the given modifier and accessor.

`OPERATION_TABLE` a dictionary of dictionaries, storing functions that will operate on various types of data.

`put_op` takes 3 inputs: an operation name, a tuple of tag-types, and a function to put into `OPERATION_TABLE`. It returns `None`.

`get_op` takes 2 inputs: the operation name and the tuple of tag-types. It returns the associated function retrieved from `OPERATION_TABLE`.

The following code demonstrates how functions are put into `OPERATION_TABLE`:

```
put_op("to_dna", ("dna",), lambda x: x)
put_op("to_dna", ("rna",), reverse_transcribe)
put_op("to_rna", ("dna",), transcribe)
put_op("to_rna", ("rna",), lambda x: x)
put_op("is_same_dogma", ("dna","dna"), lambda x, y: x == y)
put_op("is_same_dogma", ("dna","rna"), lambda x, y: transcribe(x) == y)
put_op("is_same_dogma", ("rna","dna"), lambda x, y: x == transcribe(y))
put_op("is_same_dogma", ("rna","rna"), lambda x, y: x == y)
```

This is one way of visualising the modified `OPERATION_TABLE`:

op name / tag-type tuple	`"to_dna"`	`"to_rna"`	`"to_protein"`	`"is_same_dogma"`
`("dna",)`	`lambda x:x`	`transcribe`	<Task 3b>	×
`("rna",)`	`reverse_transcribe`	`lambda x:x`	<Task 3b>	×
`("protein",)`	<Task 3e?>	<Task 3e?>	<Task 3b>	×
`("dna","dna")`	×	×	×	`lambda x,y:...`
`("rna","dna")`	×	×	×	`lambda x,y:...`
`("dna","rna")`	×	×	×	`lambda x,y:...`
`("rna","rna")`	×	×	×	`lambda x,y:...`
`("dna","protein")`	×	×	×	<Task 3c>
...	...	...	...	...
`("protein","protein")`	×	×	×	<Task 3c>

We can then retrieve the `transcribe` function by calling `get_op`.

```
some_op = get_op("to_rna", ("dna",))
some_op == transcribe  # True

another_op = get_op("to_dna", ("dna",))
another_op(dna) == dna # True
```

Task 2: Data-Directed Programming (6 marks)

In data-directed programming, any operation performed on the data is driven by looking-up the correct function to dispatch from the operation table. The program retrieves the correct function based on the attached tags to the given inputs.

The function `to_dna` takes in a generic Tagged-Data and converts it to a **Tagged-DNA**.

Similarly, the function `to_rna` converts a Tagged-Data to **Tagged-RNA**.

The function `is_same_dogma` takes in two Tagged-Datas and returns if `True` they both are part of the same information flow DNA → RNA → protein, otherwise it returns `False`.

`to_dna(Tagged-Data)`	→	**`Tagged-DNA`**
`to_rna(Tagged-Data)`	→	**`Tagged-RNA`**
`is_same_dogma(Tagged-Data, Tagged-Data)`	→	`boolean`

The operations `to_dna`, `to_rna` and `is_same_dogma` act on generic Tagged-Datas as arguments, and do not specify if the given Tagged-Data is a **Tagged-DNA**, **Tagged-RNA** or **Tagged-Protein**. Hence, we have to first extract the tag-types, then use the tag-types to look for the correct function in `OPERATION_TABLE` which matches the given types of data.

The function `to_dna` has been implemented for you.
Implement the other two functions `to_rna` and `is_same_dogma`. (3 marks each)

```
def to_dna(tagged_data):
    tag  = get_tag_type(tagged_data)
    data = get_data(tagged_data)
    op   = get_op("to_dna", (tag,))
    return tag("dna", op(data))

def to_rna(your_args_here):
    """Your code here"""

def is_same_dogma(your_args_here):
    """Your code here"""

tagged_dna1 = tag("dna", "AAATGC")
tagged_rna1 = tag("rna", "GCAUUU")
tagged_dna2 = tag("dna", "TTACAT")
tagged_rna2 = tag("rna", "AUGUAA")

get_data(to_rna(tagged_dna1)) # 'GCAUUU'
get_data(to_rna(tagged_rna1)) # 'GCAUUU'

is_same_dogma(tagged_dna1, tagged_rna1) # True
is_same_dogma(tagged_rna1, tagged_rna1) # True
is_same_dogma(tagged_rna1, tagged_dna2) # False
```

Task 3: Extending the Operation Table (15 marks)

We can now operate freely between **Tagged-DNA** and **Tagged-RNA** types.
Let us try extending `OPERATION_TABLE` to also support **Tagged-Protein** operations.

(a) Implement the function `to_protein`, that takes as input a Tagged-Data.
It returns a **Tagged-Protein**. **(3 marks)**

(b) To support the newly-defined `to_protein`. Write 3 `put_op` function calls to put the appropriate functions into `OPERATION_TABLE`: **(3 marks)**

(i) a function that converts **Tagged-DNA** into **Tagged-Protein**,

(ii) a function that converts **Tagged-RNA** into **Tagged-Protein**,

(iii) a function that converts **Tagged-Protein** into **Tagged-Protein**.

(c) Write 5 additional `put_op` function calls to allow `is_same_dogma` to accept and check if the new Tagged-Data pairs are from the **same information flow** (DNA → RNA → protein): **(5 marks)**

(i) a function that checks **Tagged-Protein** and **Tagged-Protein**,

(ii) a function that checks **Tagged-Protein** and **Tagged-DNA**,

(iii) a function that checks **Tagged-DNA** and **Tagged-Protein**,

(iv) a function that checks **Tagged-Protein** and **Tagged-RNA**,

(v) a function that checks **Tagged-RNA** and **Tagged-Protein**.

(d) Some of the code lines below may eventually cause an error when executed. Comment those lines and uncomment the remaining lines that will never cause an error. **(1 mark)**

```
dna_1 = to_dna(protein)
dna_2 = to_dna(to_protein(tagged_rna))
dna_3 = to_dna(tagged_rna)
dna_4 = to_dna(tagged_protein)
dna_5 = to_dna(to_protein(tagged_dna))
dna_6 = to_dna(to_rna(tagged_dna))
dna_7 = to_dna(to_rna(tagged_protein))
dna_8 = to_dna(to_protein(to_rna(tagged_dna)))
dna_9 = to_dna(to_rna(to_protein(dna_1)))
dna_10 = ('dna', get_data(dna_6))
```

(e) Resident biologist Linus says that we do not need to add in the reverse conversion from **Tagged-Protein** to **Tagged-RNA** into `OPERATION_TABLE`, because of degeneracy. This means that multiple **Tagged-RNA**s can give rise to a single **Tagged-Protein**, making it impossible to determine which **Tagged-RNA** the **Tagged-Protein** originated from.

Give 6 different examples of **Tagged-RNA**s that will give rise to the **Tagged-Protein** with data equals to `"MYVHAN_"`. Put them in a list called `my_rna_list`. **(2 marks)**

(f) Yuhan, Linus' twin brother, wonders if there are more than 6 **Tagged-RNA**s that satisfies the problem above. How many different **shortest** possible **Tagged-RNA**s that will give rise to the **Tagged-Protein** with data equals to `"MYVHAN_"`? **(1 mark)**