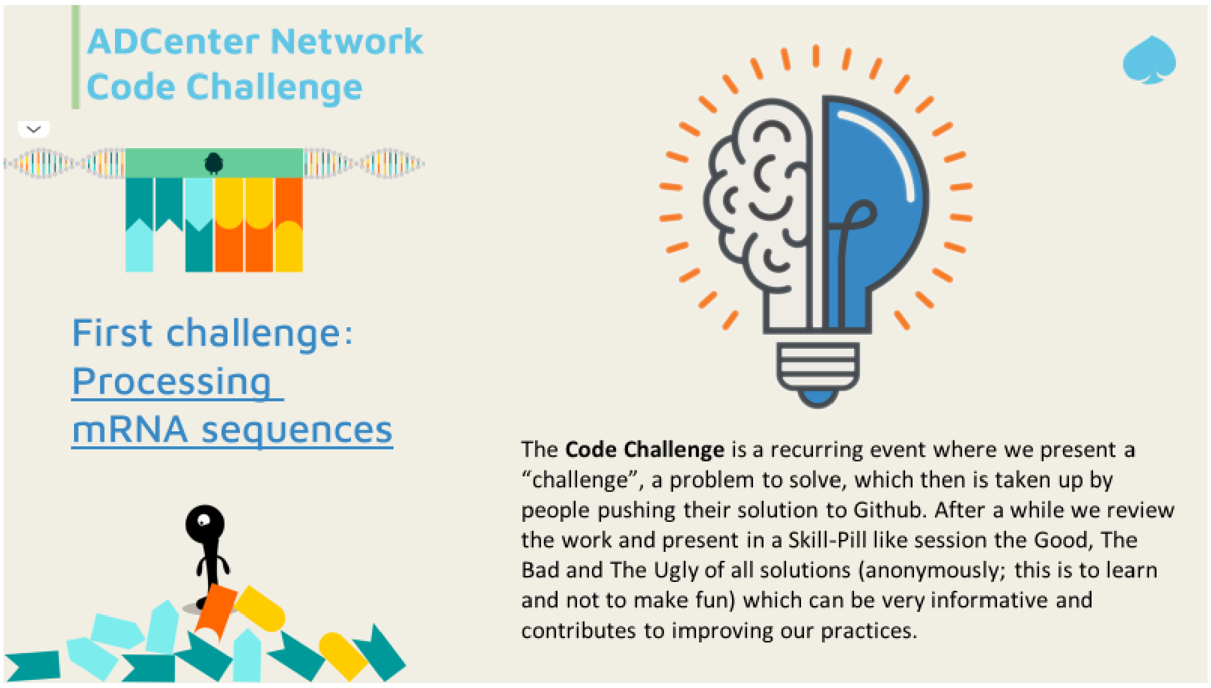
ADCenter Network Code Challenge

Processing mRNA Sequences

Introduction



ADCenter Network Code Challenge

First challenge: Processing mRNA sequences

The **Code Challenge** is a recurring event where we present a "challenge", a problem to solve, which then is taken up by people pushing their solution to Github. After a while we review the work and present in a Skill-Pill like session the Good, The Bad and The Ugly of all solutions (anonymously; this is to learn and not to make fun) which can be very informative and contributes to improving our practices.

Description

The sequence of bases in DNA carries the genetic code of all living things on earth. Scattered along the DNA molecule are particularly important sequences of bases known as *genes*. Each gene is a coded description for making a particular protein. Getting from the code in DNA to the final protein is a very complicated process which will be naively simplified in the following (for a proper deep dive see: https://en.wikipedia.org/wiki/Protein_biosynthesis)

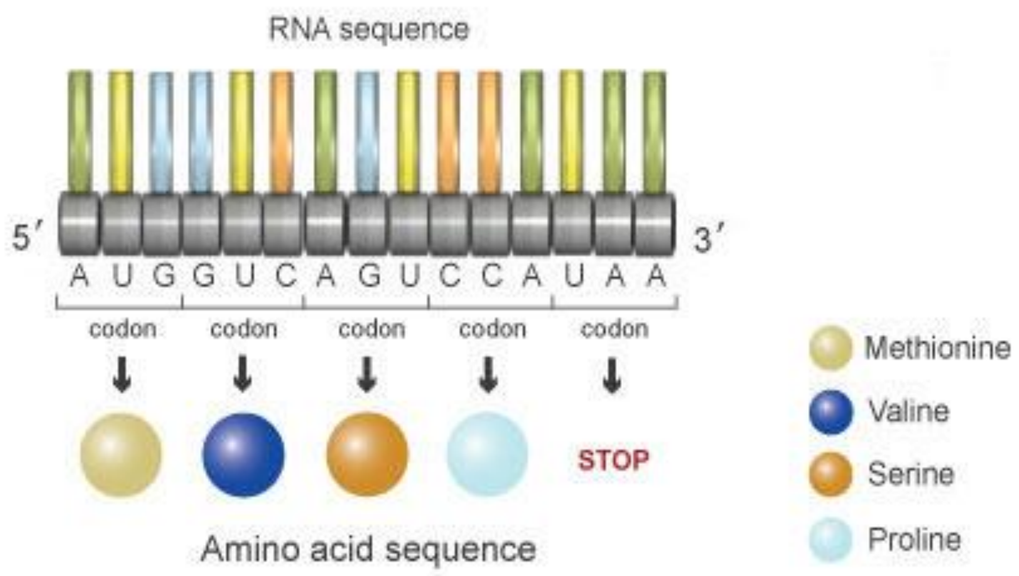
The genetic code in DNA is first transcribed ("copied") to *messenger* RNA (mRNA). That then travels out of the nucleus of the cell (where the DNA is found) into the cytoplasm of the cell. The cytoplasm contains essentially everything else in the cell apart from the nucleus. Here the code is read, and the protein is synthesised in the ribosome.

This sequence of mRNA is broken into a series of three-nucleotide units known as "*codons*" (see down). The three-letter nature of codons means that the four nucleotides found in mRNA — called A, U, G, and C — can produce a total of 64 different combinations.

So, a sequence of these three letters, the codons, are typically depicted like the following example:

... CUG UCU AUG GGA AAA UGC UGA UUA AGU UUU AUG UCC UCC ...

Genes are defined by a sequence of what could be many hundreds of codons. Each gene contains a 'start codon' which indicates where it begins and a 'stop codon' which defines where the gene ends. Start codons can vary according to the species but there are commonly three possible stop codons: UAG, UGA and UAA.



Challenge

Write a function (any programming language allowed but keep it reasonable: esoteric languages like brainf*ck are not helpful) which accepts a single string containing the letters of the genetic code.

The input string is typically generated by automatic sequencing of mRNA. In the industry there is a standard to include metadata and comments to these files by adding lines prefixed with “>” (greater then) character. All data after that character until the end of the line can be ignored.

The remaining data in the input string may contain whitespace characters (space, tab, linefeed, Unicode whitespace) which have no semantic meaning and should be ignored.

Apart from whitespace the string should only contain the four characters ´A´, ´U´, ´G´ and ´C´ which may appear in upper or lowercase representation. The function should process the string from the first position. The string contains none, one or more “genes” (sequences of codons terminated by a stop codon) and possible “noise”, a sequence of stop codons. The function should return the genes, each consisting of a sequence of codons and terminated by a single “stop codon”. More stop codons should be treated as noise and be discarded.

The function should return:

- An array (or equivalent, i.e. a sequence) of genes as contained in the string . Note that:
    - A gene should be an array (or equivalent, i.e. a sequence) where each item represents a possible codon. Encoding of the coding can be in a byte, char, or a higher-level representation.
- Alternatively: an error state, indicating invalid input. Error states may be:
  - invalid characters or

- invalid string length (remaining string length mod 3 != 0; NOT the full string as the string may contain whitespace and comments)
- Unexpected end of gene, I.e., no stop codon at the end of the string)

- All error state should incorporate the character position on which the error occurred and a human readable – user friendly – error message.

The function should be accompanied by a unit test with the standard unit testing framework (i.e. JUnit for Java) covering all possible input/output states.

The function should be safe (not throw exceptions, not “show undefined behaviour” (null pointer exceptions etc) and be immutable (i.e. not change the original input data)

Expressed as Types in trivial, pseudo-code:

```
type Seq<T> = <... array or equivalent sequence of type T ...>
type Codon = <... Scalar denoting one of 64 combinations of the 4 nucleotides
A,U,G,C ...> or otherwise
type StopCodon = Codon where instance of Codon = UAG | UGA | UAA
type Gene = Seq<Codon... (1...n) + StopCodon>
type Genes = Seq<Gene>
type Error = InvalidChars | InvalidLength | UnexpectedEnd

type processGenes =  (string) -> Error | Genes
```

Challenge 2

As automatic gene sequencing can generate very large datasets, into the hundreds of megabytes, a function processing the entire data set is not scalable. Therefore, the intention is to create a mechanism (optionally but not necessarily using the first function) which processes the input data as a "stream" instead of a string (type of the stream up to the implementation). The following requirements are defined:

- This mechanism, from here on “processor”, consists of a function or object which must obtain a single, complete gene from the data stream (not individual characters nor codons) consecutively (i.e. in steps, one after the other)
- So, the return type of method/function/call back should be `Error| Gene` rather than `Error |Genes.`
- Therefore, the processor must be called n times to obtain n genes from the stream
- The processor may be stateful but must treat the underlying data stream as immutable
- The processor should only use the memory necessary to process a single gene
- The processor could be treated as a file (“read” operations), an iterator (“next” method or “iterator” interface) or even a reactive mechanism (call-backs). But each operation should only return one gene at a time
- The call to the processor can be blocking in case of an incomplete gene (InvalidLength | UnexpectedEnd errors) rather than generating an error in case future data could negate the error
- If the underlying stream terminates and an error condition occurs than in all cases the error should be generated

- A compound datatype with associated functions (i.e. class/object with methods or record with functions, etc) should be used
- An abstraction on top of distinct implementation offering processing different types of data streams (i.e. an interface with two implementations, etc) should be used
- An implementation for a string data type (for testing) and one for a text file must be implemented
- The processor must provide a sentinel value denoting if no more genres will be returned (the underlying data stream has terminated/closed) (this may be through a *hasNext* method/function or a special, unique return value of the next method/function etc)

Possible usages of a processor in trivial, pseudo code:

```
val proc = new GeneProcessorFileInput(input_data_url)
for gene in proc {
   print(gene)  //print successful result or error
}
```

Alternatively:

```
val proc =  { file: input_data_url, state: null}
while has_next_gene(proc) {
      print (get_gene(proc)) //print successful result or error
}
```

Note that the shown forms are for illustrative purposes only. They serve as examples. Implementation is up to the developer as long as all requirements are met.

Example string

```
uuucaugug cccaaaauc cucucaggc auggucaag cccauccuu uuccacaac
acagccuag
>NM_001293063 1
augugcgag gacugcugu gcugcaacu guuuuccgu ccuuucuuu cacuaa
```

Data File

See the file `refMrna.fa.txt` for a large(r) data set.