OCaml

Pattern Matching and Data Types - 0

Summary: The main theme of this module is the pattern matching usage and the manipulation of the various constructed types available in OCaml.

Version: 1.00

Contents

Chapter I

Foreword

Here is what Wikipedia has to say about DNA:

Deoxyribonucleic acid is a molecule that encodes the genetic instructions used in the development and functioning of all known living organisms and many viruses. DNA is a nucleic acid; alongside proteins and carbohydrates, nucleic acids compose the three major macromolecules essential for all known forms of life. Most DNA molecules consist of two biopolymer strands coiled around each other to form a double helix. The two DNA strands are known as polynucleotides since they are composed of simpler units called nucleotides. Each nucleotide is composed of a nitrogen-containing nucleobase—either guanine (G), adenine (A), thymine (T), or cytosine (C)—as well as a monosaccharide sugar called deoxyribose and a phosphate group. The nucleotides are joined to one another in a chain by covalent bonds between the sugar of one nucleotide and the phosphate of the next, resulting in an alternating sugar-phosphate backbone. According to base pairing rules (A with T and C with G), hydrogen bonds bind the nitrogenous bases of the two separate polynucleotide strands to make double-stranded DNA. DNA was first discovered by **James Watson** and **Francis Crick**, using experimental data collected by Rosalind Franklin and Maurice Wilkins. The structure of DNA of all species comprises two helical chains each coiled round the same axis, and each with a pitch of 34 ångströms (3.4 nanometres) and a radius of 10 ångströms (1.0 nanometres).

Chapter II

General rules

- Your project must be realized in a virtual machine.
- Your virtual machine must have all the necessary software to complete your project. These softwares must be configured and installed.
- You can choose the operating system to use for your virtual machine.
- You must be able to use your virtual machine from a cluster computer.
- You must use a shared folder between your virtual machine and your host machine.
- During your evaluations you will use this folder to share with your repository.
- Your functions should not quit unexpectedly (segmentation fault, bus error, double free, etc) apart from undefined behaviors. If this happens, your project will be considered non functional and will receive a 0 during the evaluation.
- We encourage you to create test programs for your project even though this work **won't have to be submitted and won't be graded**. It will give you a chance to easily test your work and your peers' work. You will find those tests especially useful during your defence. Indeed, during defence, you are free to use your tests and/or the tests of the peer you are evaluating.
- Submit your work to your assigned git repository. Only the work in the git repository will be graded. If Deepthought is assigned to grade your work, it will be done after your peer-evaluations. If an error happens in any section of your work during Deepthought's grading, the evaluation will stop.

Chapter III

Ocaml piscine, general rules

- Every output goes to the standard output, and will be ended by a newline, unless specified otherwise.
- The imposed filenames must be followed to the letter, as well as class names, function names and method names, etc.
- Unless otherwise explicitly stated, the keywords `open`, `for` and `while` are forbidden. Their use will be flagged as cheating, no questions asked.
- Turn-in directories are `ex00/`, `ex01/`, . . . , `exn/`.
- You must read the examples thoroughly. They can contain requirements that are not obvious in the exercise's description.
- Since you are allowed to use the `OCaml` syntaxes you learned about since the beginning of the piscine, you are not allowed to use any additional syntaxes, modules and libraries unless explicitly stated otherwise.
- The exercices must be done in order. The graduation will stop at the first failed exercice. Yes, the old school way.
- Read each exercise FULLY before starting it! Really, do it.
- The compiler to use is `ocamlopt`. When you are required to turn in a function, you must also include anything necessary to compile a full executable. That executable should display some tests that prove that you've done the exercise correctly.
- Remember that the special token `";;"` is only used to end an expression in the interpreter. Thus, it must never appear in any file you turn in. Regardless, the interpreter is a powerfull ally, learn to use it at its best as soon as possible!
- The subject can be modified up to 4 hours before the final turn-in time.
- In case you're wondering, no coding style is enforced during the `OCaml` piscine. You can use any style you like, no restrictions. But remember that a code your peer-evaluator can't read is a code he or she can't grade. As usual, big functions are a weak style.
- You will NOT be graded by a program, unless explictly stated in the subject. Therefore, you are given a certain amount of freedom in how you choose to do the

exercises. However, some piscine day might explicitly cancel this rule, and you will have to respect directions and outputs perfectly.

- Only the requested files must be turned in and thus present on the repository during the peer-evaluation.
- Even if the subject of an exercise is short, it's worth spending some time on it to be absolutely sure you understand what's expected of you, and that you did it in the best possible way.
- By Odin, by Thor! Use your brain!!!

Chapter IV

Day-specific rules

- Some themes of this module can be hard to understand. Feel free to practice as much as you can. They will all be used wisely and frequently during the Piscine. The part about basic genetics is not as hard as it seems. Calm down and take a deep breath. Really.
- You are in a functional programming Piscine, so your coding style MUST be functional (except for the side effects for the input/output). We insist, your code MUST be functional, otherwise you'll have a tedious defense session.
- For **EVERY** exercise, you `MUST` provide a full program that runs enough tests to prove that your work is done.
- From exercise 05, you must embed the code of each previous exercise into the next one, i.e., ex06 embeds ex05, ex07 embeds ex05 and ex06, and so on.

Chapter V

Exercise 00: Do you even compress?

	Exercise 00
Exercise 00: Do you even compress?	
Turn-in directory : *ex*00/	
Files to turn in : `encode.ml`	
Allowed functions : `None`	

Run-length encoding is a very simple form of data compression algorithm. Consecutive elements are stored as a single data element and the number of times it repeats. For instance, the string "aaabbb" can be stored as "3a3b".

Write a function *encode* that encodes a list of elements to a list of tuples containing the element and the number of times it repeats. The function must be typed as:

```
val encode : 'a list -> (int * 'a) list
```

In case of an empty list as a parameter, the function should return an empty list as well.

Chapter VI

Exercise 01: Crossover

	Exercise 01
Exercise 01: Crossover	
Turn-in directory : *ex*01/	
Files to turn in : `crossover.ml`	
Allowed functions : `None`	

Write a function `crossover` that takes two lists as parameters and returns a list of all the common elements between the two lists. The function must be typed as:

```
val crossover : 'a list -> 'a list -> 'a list
```

In case of an empty list as one of the parameters, the function should return an empty list too. But it's obvious, isn't it? We don't have to handle duplicates in lists.

Chapter VII

Exercise 02: Fifty Strings of Gray

	Exercise 02
Exercise 02: Fifty Strings of Gray	
Turn-in directory : *ex*02/	
Files to turn in : `gray.ml`	
Allowed functions : `String module and not the @ operator`	

The sequence of Gray is a sequence of possible combinations of bits ordered so that when you want to go from one element to the following, you only have to shift one bit. It's a way of having a constant time of computing when changing values so that there's no intermediate state that can crash a program. If you have a 2-bit standard set sequence, for example: `00 01 10 11`

Assume you are in the state `01`. If you want to switch to the next state, you have to change the last bit to 0 and the first one to 1. There could be an intermediate state where the set of bits is `00` before becoming `10`. And that's wrong.

The Gray sequence of a set of two bits is as follows: `00 01 11 10`. That way, when you pass from `01` to `11`, you only have to shift one bit.

Write a function that takes an integer n as a parameter and writes all the strings of the Gray sequence of size n, in the correct order on the standard output, finished by a newline.

```
# gray 1
0 1
- : unit = ()
# gray 2
00 01 11 10
- : unit = ()
# gray 3
000 001 011 010 110 111 101 100
- : unit = ()
```

Chapter VIII

Exercise 03: One and one and one is three

	Exercise 03
Exercise 03: One and one and one is three	
Turn-in directory : *ex*03/	
Files to turn in : `sequence.ml`	
Allowed functions : `None`	

Assume the following sequence:

```
1
11
21
1211
111221
312211
13112221
...
```

Just like in Exercise 00, this sequence generates the element *n* from the element *n-1* and consists of enumerating the count of the numbers found in *n-1*.

1. The first element is 1, so there is one 1; thus, the second element is 11.
2. The second element is 11, so there are two 1s; thus, the third element is 21.
3. The third element is 21, so there is one 2 and one 1; thus, the fourth element is 1211.
4. And so on. . .

Write a function *sequence* that takes an integer *n* as parameter and returns the n^{th} element of that sequence as a string. The function must be typed as: `val sequence :`

`int -> string`. In case of an invalid parameter, the function should return an empty string.

Chapter IX

Exercise 04: DNA -> Nucleotides

	Exercise 04
Exercise 04: DNA -> Nucleotides	
Turn-in directory : *ex*04/	
Files to turn in : `nucleotides.ml`	
Allowed functions : `None`	

The very beginning of DNA takes place in a structure consisting of a phosphate group linked to a deoxyribose sugar, which is itself linked with a nucleobase. A list of many structures is called a helix, and two of them make a DNA sample. Helix: P - D - Base, P - D - Base, . . .

- Create the type `phosphate`, which is an alias for the string type.
- Create the type `deoxyribose`, which is also an alias for the string type.
- Create the variant type `nucleobase`. Its constructors are `A`, `T`, `C`, `G`, and `None`.
- Write the *nucleotide* type that contains three elements: one `phosphate`, one `deoxyribose`, and one `nucleobase`. The structure of the type *nucleotide* is up to you; a record or a tuple will do the trick.
- Write a function `generate_nucleotide` that returns a nucleotide from a given nucleobase passed as a `char`. The function must be typed as `val generate_nucleotide : char -> nucleotide`. Set the `phosphate` value to `"phosphate"` and the `deoxyribose` value to `"deoxyribose"`.

Chapter X

Exercise 05: DNA -> Helix

	Exercise 05
Exercise 05: DNA -> Helix	
Turn-in directory : *ex*05/	
Files to turn in : `helix.ml`	
Allowed functions : `String concatenation operator and Random module`	

As seen previously, two helices can combine to create a DNA structure. As you will see in this exercise, rules are applied when there is a combination. The link of that combination occurs where the bases are located: P - D - Base <=> Base - D - P, P - D - Base <=> Base - D - P, . . .

- Write an *helix* type that is a list of elements of type *nucleotide*.
- Write a function `generate_helix` that takes an int *n* as a parameter and construct a random sequence of nucleotides as a list of size *n*. The function must be typed a : `val generate_helix : int -> helix`.
- Write a function helix_to_string that convert a list of nucleotides as helix type resulting from the previous function to a string of nucleobases. The function must be typed as: `val helix_to_string : helix -> string`.
- Write a function complementary_helix that takes an helix as a parameter and generate the corresponding helix according of the Nucleobase pairing rules that follows :
    - A (Adenine) can be associated with T.
    - T (Thymine) can be associated with A.
    - C (Cytosine) can be associated with G.
    - G (Guanine) can be associated with C.

  The function must be typed as: `val complementary_helix : helix -> helix`

This exercise is not mandatory.

Chapter XI

Exercise 06: DNA -> Messenger RNA

	Exercise 06
Exercise 06: DNA -> Messenger RNA	
Turn-in directory : *ex*06/	
Files to turn in : `rna.ml`	
Allowed functions : `None`	

A Messenger RNA is a molecule involved in the process of synthesizing proteins. The main aim of the RNA is to create a complementary working copy of a DNA Helix. It's really clever since it prevents the DNA of being altered and allows multiple copies so that the process can be really fast. It was first introduced by scientists Jacques Monod and Francois Jacob.

- Write a type `rna` as a list of elements of type `nucleobase`.
- Write a function `generate_rna` that creates an element of type `rna` from an element of type `helix` according to the following rules :
  - During the creation, the rna is just like a complementary helix except that the `T` nucleobase is switched to `U` nucleobase (Uracil). Modify your type `nucleobase` accordingly. (I told you to read the whole subject before starting...)
  - The list of nucleobases of the rna is the list of nucleobases that are complementary with the original helix's nucleobase (except for the first rule).

  For instance, the sequence of `nucleobase` "ATCGA" will produce a [U;A;G;C;U] rna. The function must be typed as: `val generate_rna: helix -> rna`.

`This exercise is not mandatory.`

Chapter XII

Exercise 07: DNA -> Ribosome

	Exercise 07
Exercise 07: DNA -> Ribosome	
Turn-in directory : *ex*07/	
Files to turn in : `ribosome.ml`	
Allowed functions : `None`	

The ribosome is a large and complex molecular machine found within all living cells. Its main purpose is to synthesize proteins from messenger RNA (mRNA) by combining amino acids together. Proteins are essential to all living organisms, including humans.

- Write a function *generate_bases_triplets* that creates a list of triplets of elements of type *nucleobase* from an element of type *rna* according to the following rule: if the number of nucleobases of the list is not a multiple of 3, it ignores the last incomplete triplet. The function must be typed as : `generate_bases_triplets : rna -> (nucleobase * nucleobase * nucleobase) list`.

- Write a `protein` type that consists of a list of `aminoacid`, and of function `string_of_protein` of type `protein -> string`.

- Write a function *decode_arn* of type `rna -> protein` that creates a list of the variant type *aminoacid* from an element of type *rna* according to the following rules:

  - The decode process begins with the first triplet and ends with the first Stop triplet encountered. Obvious isn't it?

  - Here is the matching table of the nucleobases triplet, the corresponding amino acid and the constructor of type *aminoacid* :

    - UAA, UAG, UGA : End of translation -> `Stop`
    - GCA, GCC, GCG, GCU : Alanine -> `Ala`
    - AGA, AGG, CGA, CGC, CGG, CGU : Arginine -> `Arg`
    - AAC, AAU : Asparagine -> `Asn`

* GAC, GAU : Aspartique -> `Asp`
* UGC, UGU : Cysteine -> `Cys`
* CAA, CAG : Glutamine -> `Gln`
* GAA, GAG : Glutamique -> `Glu`
* GGA, GGC, GGG, GGU : Glycine -> `Gly`
* CAC, CAU : Histidine -> `His`
* AUA, AUC, AUU : Isoleucine -> `Ile`
* CUA, CUC, CUG, CUU, UUA, UUG : Leucine -> `Leu`
* AAA, AAG : Lysine -> `Lys`
* AUG : Methionine -> `Met`
* UUC, UUU : Phenylalanine -> `Phe`
* CCC, CCA, CCG, CCU : Proline -> `Pro`
* UCA, UCC, UCG, UCU, AGU, AGC : Serine -> `Ser`
* ACA, ACC, ACG, ACU : Threonine -> `Thr`
* UGG : Tryptophane -> `Trp`
* UAC, UAU : Tyrosine -> `Tyr`
* GUA, GUC, GUG, GUU : Valine -> `Val`

`This exercise is not mandatory.`

Chapter XIII

Exercise 08: DNA -> The Complete Process of Protein Creation

	Exercise 08
Exercise 08: DNA -> The Complete Process of Protein Creation	
Turn-in directory : *ex*08/	
Files to turn in : `life.ml`	
Allowed functions : `String module`	

- Write a function that goes from the generation of a helix of a reasonable length to the creation of the corresponding `protein`. Each step must be displayed clearly on the standard output. This function takes a string as a parameter.

`This exercise is not mandatory.`

Chapter XIV

Submission and Peer Evaluation

Turn in your assignment in your `Git` repository as usual. Only the work inside your repository will be evaluated during the defense. Don't hesitate to double-check the names of your folders and files to ensure they are correct.

```
The evaluation process will take place on the computer of the
evaluated group.
```