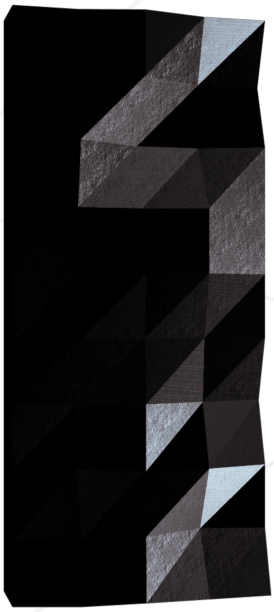

Ft_turing

The origin of programming

Pedago pedago@staff.42.fr

Abstract: Ever heard of Turing's machine ?

Contents

Chapter I

Forewords

Here is what Wikipedia has to say about the man who created your job:

Alan Mathison Turing was a British pioneering computer scientist, mathematician, logician, cryptanalyst, philosopher, mathematical biologist, and marathon and ultra distance runner. He was highly influential in the development of computer science, providing a formalisation of the concepts of algorithm and computation with the Turing machine, which can be considered a model of a general purpose computer. Turing is widely considered to be the father of theoretical computer science and artificial intelligence.

During the Second World War, Turing worked for the Government Code and Cypher School (GC&CS) at Bletchley Park, Britain's codebreaking centre. For a time he led Hut 8, the section responsible for German naval cryptanalysis. He devised a number of techniques for breaking German ciphers, including improvements to the pre-war Polish bombe method, an electromechanical machine that could find settings for the Enigma machine. Turing played a pivotal role in cracking intercepted coded messages that enabled the Allies to defeat the Nazis in many crucial engagements, including the Battle of the Atlantic; it has been estimated that this work shortened the war in Europe by as many as two to four years.

After the war, he worked at the National Physical Laboratory, where he designed the ACE, among the first designs for a stored-program computer. In 1948 Turing joined Max Newman's Computing Laboratory at the University of Manchester, where he helped develop the Manchester computers and became interested in mathematical biology. He wrote a paper on the chemical basis of morphogenesis, and predicted oscillating chemical reactions such as the Belousov–Zhabotinsky reaction, first observed in the 1960s.

Turing was prosecuted in 1952 for homosexual acts, when such behaviour was still a criminal act in the UK. He accepted treatment with oestrogen injections (chemical castration) as an alternative to prison. Turing died in 1954, 16 days before his 42nd birthday, from cyanide poisoning. An inquest determined his death a suicide, but it has been noted that the known evidence is equally consistent with accidental poisoning. In 2009, following an Internet campaign, British Prime Minister Gordon Brown made an official public apology on behalf of the British government for "the appalling way he was treated". Queen Elizabeth II granted him a posthumous pardon in 2013.

Chapter II

Introduction

The Turing machine is a mathematical model fairly easy to understand and to implement. A formal definition is available here, or here for instance. The `ft_turing` project is an `OCaml` implementation of a single infinite tape Turing machine.

Figure II.1: Alan Turing

Chapter III

Objectives

The goal of this project is to write a program able to simulate a single headed, single tape Turing machine from a machine description provided in `json`.

This project will also let you write `OCaml` code in the context of a real program. It will be a good occasion to experiment clever type designs and a smart functionnal approach to your program.

As stated below, you will be free to use any `OCaml` libraries you like. The project is also a good opportunity to discover famous and widely used libraries like `Core` or `Batteries` to name a few.

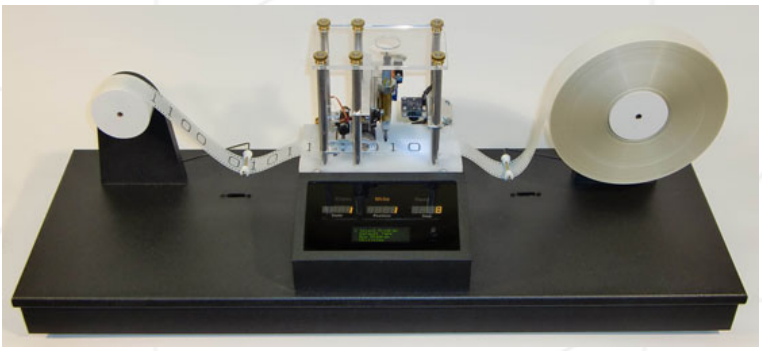

Figure III.1: A Turing machine

Chapter IV

Generic rules

- If you are more of a `Haskell` kind of person, the use of `Haskell` instead of `OCaml` is allowed for this project. In this case, adapt the rules stated in this chapter to `Haskell`. This project is in `OCaml`, `Haskell` is accepted, any other language is forbidden.

- You are free to use any version of `OCaml`, any syntax extension, any version of any library, and any `OCaml` tool you want.

- You must turn-in a `Makefile` able to compile your code with `ocamlopt` and `ocamlc`. Also, your `Makefile` must detect any missing libraries and/or tools needed by your program and install them via `OPAM`. Your peer-evaluator is never required to install anything himself prior to the defense.

- 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. Anyway, the interpreter is a powerfull ally, learn to use it at its best as soon as possible !

- In case you're wondering, no coding style is enforced in `OCaml`. You can use any style you like, no restrictions. But remember that a code your peer-evaluator can't read is a code she or he can't grade. As usual, big fonctions is a weak style. Also, taking time to design smart modules and interfaces is the best way to success.

- Please, remember to not rely on imperative style just because you feel confortable with it. Iterators like `iter`, `map` or `fold` are your best friends, along with anonymous functions, and constants.

Chapter V

Mandatory part

V.1 The Turing machine

You must write a program able to simulate a single headed and single tape Turing machine from a `json` machine description given as a parameter to your program. The `json` machine description is sligthly simplier than a formal description of the same machine.

This is a valid exemple of a `json` machine description for this project:

```
$>cat unary_sub.json
{
    "name"    : "unary_sub",
    "alphabet": [ "1", ".", "-", "=" ],
    "blank"   : ".",
    "states"  : [ "scanright", "eraseone", "subone", "skip", "HALT" ],
    "initial" : "scanright",
    "finals"  : [ "HALT" ],

    "transitions" : {

        "scanright": [
            { "read" : ".", "to_state": "scanright", "write": ".", "action": "RIGHT"},
            { "read" : "1", "to_state": "scanright", "write": "1", "action": "RIGHT"},
            { "read" : "-", "to_state": "scanright", "write": "-", "action": "RIGHT"},
            { "read" : "=", "to_state": "eraseone" , "write": ".", "action": "LEFT" }
        ],

        "eraseone": [
            { "read" : "1", "to_state": "subone", "write": "=", "action": "LEFT"},
            { "read" : "-", "to_state": "HALT"  , "write": ".", "action": "LEFT"}
        ],

        "subone": [
            { "read" : "1", "to_state": "subone", "write": "1", "action": "LEFT"},
            { "read" : "-", "to_state": "skip"  , "write": "-", "action": "LEFT"}
        ],

        "skip": [
            { "read" : ".", "to_state": "skip"     , "write": ".", "action": "LEFT"},
            { "read" : "1", "to_state": "scanright", "write": ".", "action": "RIGHT"}
        ]
    }
}
$>
```

The `json` fields are defined as follows:

name: The name of the described machine

alphabet: Both input and work alphabet of the machine merged into a single alphabet for simplicity's sake, including the blank character. Each character of the alphabet must be a string of length strictly equal to 1.

blank: The blank character, must be part of the alphabet.

states: The exhaustive list of the machine's states names.

initial: The initial state of the machine, must be part of the states list.

finals: The exhaustive list of the machine's final states. This list must be a sub-list of the `states` list.

transitions: A dictionnary of the machine's transitions indexed by state name. Each transition is a list of dictionnaries, and each dictionnary describes the transition for a given character under the head of the machine. A transition is defined as follows:

- **read:** The character of the machine's alphabet on the tape under the machine's head.
- **to_state:** The new state of the machine after the transition is done.
- **write:** The character of the machine's alphabet to write on the tape before moving the head.
- **action:** Movement of the head for this transition, either `LEFT`, or `RIGHT`.

- Your program will have the following usage:

```
$./ft_turing --help
usage: ft_turing [-h] jsonfile input

positional arguments:
  jsonfile              json description of the machine

  input                 input of the machine

optional arguments:
  -h, --help            show this help message and exit
```

- Your program must detect and reject ill formated or invalid machine descriptions and inputs, with a relevant error message. This means that your program must never crash for any reason.

- Your program must output at least the state of the tape with a visible representation of the head at each transition. For instance, something like:

```
$>./ft_turing res/unary_sub.json "111-11="
********************************************************************************
*                                                                              *
*                                unary_sub                                     *
*                                                                              *
********************************************************************************
Alphabet: [ 1, ., -, = ]
States  : [ scanright, eraseone, subone, skip, HALT ]
Initial : scanright
Finals  : [ HALT ]
(scanright, .) -> (scanright, ., RIGHT)
(scanright, 1) -> (scanright, 1, RIGHT)
(scanright, -) -> (scanright, -, RIGHT)
(scanright, =) -> (eraseone, ., LEFT)
(eraseone, 1) -> (subone, =, LEFT)
(eraseone, -) -> (HALT, ., LEFT)
(subone, 1) -> (subone, 1, LEFT)
(subone, -) -> (skip, -, LEFT)
(skip, .) -> (skip, ., LEFT)
(skip, 1) -> (scanright, ., RIGHT)
********************************************************************************
[<1>11-11=.............] (scanright, 1) -> (scanright, 1, RIGHT)
[1<1>1-11=.............] (scanright, 1) -> (scanright, 1, RIGHT)
[11<1>-11=.............] (scanright, 1) -> (scanright, 1, RIGHT)
[111<->11=.............] (scanright, -) -> (scanright, -, RIGHT)
[111-<1>1=.............] (scanright, 1) -> (scanright, 1, RIGHT)
[111-1<1>=.............] (scanright, 1) -> (scanright, 1, RIGHT)
[111-11<=>.............] (scanright, =) -> (eraseone, ., LEFT)
[111-1<1>..............] (eraseone, 1) -> (subone, =, LEFT)
[111-<1>=..............] (subone, 1) -> (subone, 1, LEFT)
[111<->1=..............] (subone, -) -> (skip, -, LEFT)
[11<1>-1=..............] (skip, 1) -> (scanright, ., RIGHT)
[11.<->1=..............] (scanright, -) -> (scanright, -, RIGHT)
[11.-<1>=..............] (scanright, 1) -> (scanright, 1, RIGHT)
[11.-1<=>..............] (scanright, =) -> (eraseone, ., LEFT)
[11.-<1>...............] (eraseone, 1) -> (subone, =, LEFT)
[11.<->=...............] (subone, -) -> (skip, -, LEFT)
[11<.>-=...............] (skip, .) -> (skip, ., LEFT)
[1<1>.-=...............] (skip, 1) -> (scanright, ., RIGHT)
[1.<.>-=...............] (scanright, .) -> (scanright, ., RIGHT)
[1..<->=...............] (scanright, -) -> (scanright, -, RIGHT)
[1..-<=>...............] (scanright, =) -> (eraseone, ., LEFT)
[1..<->................] (eraseone, -) -> (HALT, ., LEFT)
```

- If for some reason the machine is blocked, you must detect it and inform the user of your program of what happened.

V.2 Machine descriptions

Writing a `Turing machine` is a lot of fun. But using it to compute actual stuff is even more fun. In the previous section you had a machine description of a machine able to compute unary substraction. In that exemple, the alphabet was given to you, but choosing the alphabet is not always as straightforward as one could expect because the working alphabet can be larger than the input alphabet.

You must write 5 machine descriptions for your program:

1. A machine able to compute an unary addition.
2. A machine able to decide whether its input is a palindrome or not. Before halting, write the result on the tape as a `'n'` or a `'y'` at the right of the rightmost character of the tape.
3. A machine able to decide if the input is a word of the language 0^n1^n, for instance the words `000111` or `0000011111`. Before halting, write the result on the tape as a `'n'` or a `'y'` at the right of the rightmost character of the tape.
4. A machine able to decide if the input is a word of the language 0^{2n}, for instance the words `00` or `0000`, but **not** the words `000` or `00000`. Before halting, write the result on the tape as a `'n'` or a `'y'` at the right of the rightmost character of the tape.
5. A machine able to run the first machine of this list, the one computing an unary addition. The machine alphabet, states, transitions and input ARE the input of the machine you are writing, encoded as you see fit.

Figure V.1: Advice for your 5th machine

Chapter VI

Bonus part

One of the most interesting thing about `Turing machines` is computing algorithm complexity in time or space. The bonus of this project is to modify anything you deem relevant in order to allow your program to compute the `time` complexity of the executed algorithm. Don't loose time trying to create some lame bonuses, `time` complexity is what you want to do here.

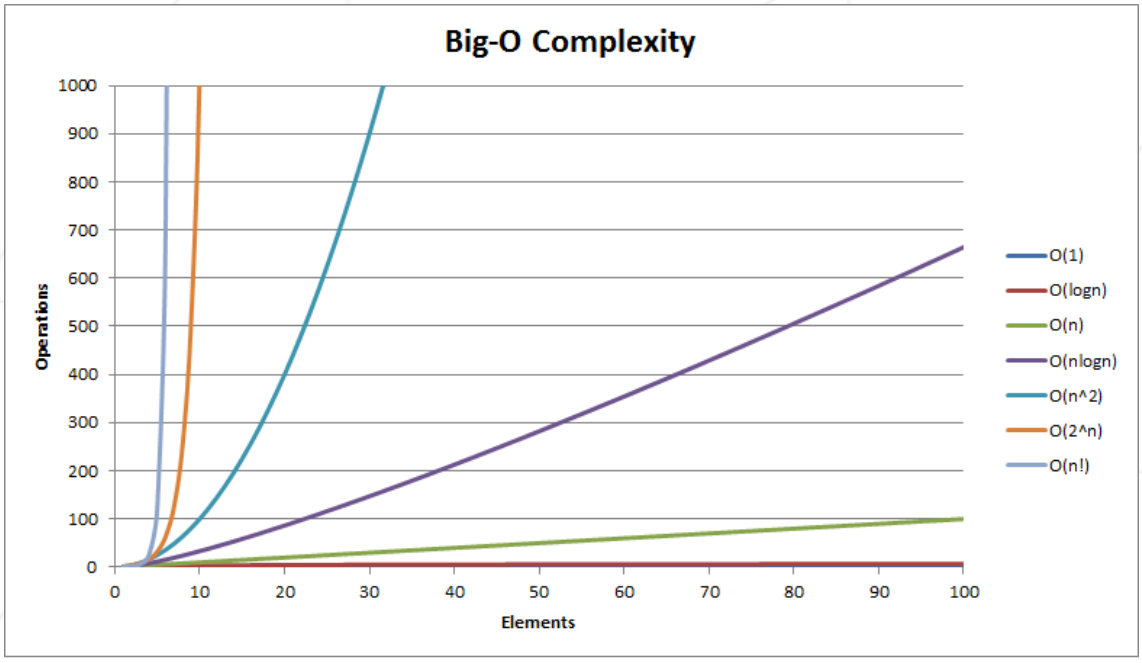

Figure VI.1: Complexities

Chapter VII

Turn-in and peer-evaluation

Turn your work in using your `GiT` repository, as usual. Only work present on your repository will be graded in defense and bonus part will be accessible if and only if your mendatory part is complete and perfect.

May Turing be with you.