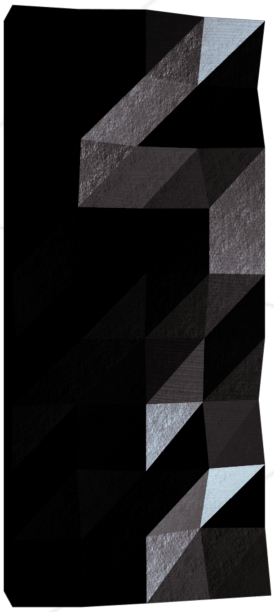

OCaml project

ft_ality

nate alafouas@student.42.fr
42 Staff pedago@42.fr

Summary: *There's hardly anything sexier than a parser.*

Contents

I Foreword **2**

II Introduction **3**

III Goals **4**

IV General instructions **5**

V Mandatory part **7**
- V.1 Formal definition . . . 7
- V.2 Automaton training . . . 8
- V.3 Automaton running and language recognition . . . 8

VI Bonus part **9**
- VI.1 Code optimization . . . 9
- VI.2 Functional programming philosophy . . . 9
- VI.3 Graphical interface . . . 10
- VI.4 Gamepads . . . 10
- VI.5 Debug mode . . . 10

VII Turn-in and peer-evaluation **12**

Chapter I

Foreword

A SQUAT grey building of only thirty-four stories. Over the main entrance the words, CENTRAL LONDON HATCHERY AND CONDITIONING CENTRE, and, in a shield, the World State's motto, COMMUNITY, IDENTITY, STABILITY.

The enormous room on the ground floor faced towards the north. Cold for all the summer beyond the panes, for all the tropical heat of the room itself, a harsh thin light glared through the windows, hungrily seeking some draped lay figure, some pallid shape of academic goose-flesh, but finding only the glass and nickel and bleakly shining porcelain of a laboratory. Wintriness responded to wintriness. The overalls of the workers were white, their hands gloved with a pale corpse-coloured rubber. The light was frozen, dead, a ghost. Only from the yellow barrels of the microscopes did it borrow a certain rich and living substance, lying along the polished tubes like butter, streak after luscious streak in long recession down the work tables.

(...)

Major instruments of social stability.

Standard men and women; in uniform batches. The whole of a small factory staffed with the products of a single bokanovskified egg.

"Ninety-six identical twins working ninety-six identical machines!" The voice was almost tremulous with enthusiasm. "You really know where you are. For the first time in history." He quoted the planetary motto. "Community, Identity, Stability." Grand words. "If we could bo- kanovskify indefinitely the whole problem would be solved."

Solved by standard Gammas, unvarying Deltas, uniform Epsilons. Millions of identical twins. The principle of mass production at last applied to biology.

By `Aldous Huxley`, in `Brave New World`.

Chapter II

Introduction

You may have heard the word “automaton” or “machine” in many different contexts ; for example, they can be used to render an AI’s behaviour, and they can also recognise some particular languages named regular languages. Regular languages are famous in the software engineering world for being describable through regular expressions — or *regexps*.

In this project you will have an insight into a particular kind of regular languages: moves and combos from fighting games, such as `Mortal Kombat`. You will implement, train and run an automaton to recognise such moves from the list of key combinations it is composed of; and you will understand that in the same way words are made with symbols, combos are made with keys.

Your task is then simple: you will recreate a fighting game’s training mode.

Chapter III

Goals

This project is the first step of a series, aimed at making you acquire the skills to perform syntactic analysis (a.k.a. parsing) on formal languages. Implementing it should make you familiar with the following notions:

- Formal grammars
- The Chomsky hierarchy
- Regular languages (or type 3 languages)
- Finite-State Automata (Finite-State Machines)

This is also, incidentally, one of your first long and complex projects in OCaml; you should take this opportunity to think carefully about your program structure in terms of modules, and set yourself up with an efficient workflow with external libraries.

Chapter IV

General instructions

- The inputs and outputs presented in this subject are only given as guidelines. You are free to format those as you wish. As such, you may index your automaton's states whichever way you see fit. If you have a doubt, just play it safe and do it like in the subject.
- This project is in `OCaml`, and the compiler must report no errors or warnings when building your project. One single warning means your project does not build and is worth 0. Yes, yes, I know. But still.
- The `open` keyword is forbidden and considered cheating. Yes, yes, I know. But still.
- Your code will be compiled, which means you should never have the `;;` token in your source files.
- You are allowed to use the following modules:
  - Pervasives
  - List
  - String
  - Sys
  - Sdl
  - Sdlevent
  - Sdlkey
- If you want to use a module and it's not in this list, well, you can't.

- No coding style is enforced for this project; but remember that small functions and wisely designed modules are elements to writing well thought code in OCaml. Your graders will have the possibility to give you bonus points if your code is elegant and robust.

- You must provide a `Makefile` which will build your entire project.

Chapter V

Mandatory part

The runtime of this project mainly consists in two steps: training the automaton, and running it.

V.1 Formal definition

A finite-state automaton A is a tuple containing the following elements:

$$A = \langle Q, \Sigma, Q_0, F, \delta \rangle$$

- Σ is the automaton's input alphabet.
- Q is the set of states in the automaton.
- Q_0 is the starting state, with $Q_0 \in Q$ of course.
- F is the set of recognition states, with $F \subseteq Q$.
- δ is a function that assigns transitions to the automaton's states; a transition is a state assigned to a couple of a state and a symbol from the alphabet, which makes the function's type $Q \times \Sigma \to Q$.

Of course, feel free to do your research on the Internet if you need more information. But the main idea is that an automaton reads the input (which is a word), symbol by symbol, and goes from one state to the other at each symbol using the transition function δ. At the end of the input, if the automaton is in a recognition state, the automaton recognizes the word. If it's not in a recognition state, then it, well, it...doesn't recognize the word.

Actually, the transition function could be a set. But then again, functions and sets are actually the same thing. You do know that, right?

V.2 Automaton training

Your automaton will be built at runtime, using grammar files that contain the moves to be learnt by the automaton. The file path of one grammar will be given in command line arguments. Since moves are usually very simple, all you have to do is split (i.e. `tokenize`) your rule to get a list of tokens, and then give it to the automaton which will generate transitions for each token in succession.

This operation should be very quick (as in, near-instant).

V.3 Automaton running and language recognition

Once you have trained your automaton, go ahead and run it! Your program will wait for input from the keyboard, just like the training mode of a fighting game. The user must be able to press keys on his keyboard, using a key mapping displayed on the screen, and move names should be displayed when their key combinations are executed.

The key mappings must be automatically computed from the grammar. If they're hardcoded, I will personally come and break your bones.

Example:

```
% ./ft_ality grammars/mk9.gmr
Key mappings:
q -> Block
down -> Down
w -> Flip Stance
left -> Left
right -> Right
e -> Tag
a -> Throw
up -> Up
s -> [BK]
d -> [BP]
z -> [FK]
x -> [FP]
----------------------

[BP]
Claw Slam (Freddy Krueger) !!
Knockdown (Sonya) !!
Fist of Death (Liu-Kang) !!

[BP], [FP]
Saibot Blast (Noob Saibot) !!
Active Duty (Jax) !!
```

Chapter VI

Bonus part

If you have a running automaton and reasonable error handling, that's great! Here are some ideas to make an even better project and earn some bonus points.

VI.1 Code optimization

Your code will also be evaluated for performance outside of formal algorithmic criteria. Here are a few guidelines to follow:

- OCaml relies on recursion to loop through calculations. As such, your functions should be written responsibly, and be tail-recursive if they can be computationally heavy.

- The program should be reasonably scalable – input processing should be online. For example, the inputs should always be processed line by line without the program storing everything in memory; if it does, it will crash if given sufficiently large input.

VI.2 Functional programming philosophy

This project is in OCaml (whether you want it or not), which means your code should follow functional programming principles as much as possible. Here are some guidelines to help you (but really, this should be obvious to everyone).

- Your functions should be short (preferably less than 20 lines), and should use nested definitions when needed. "Utility" functions left at the module's top level shows a messy code.

- With the same idea in mind, your code should be split into well-thought modules; these modules should not export unnecessary functions (e.g. previously mentioned "utility" functions).

- Your code should avoid exceptions as much as possible. Exceptions break the mechanism of stack frames, resulting in awkward constructions relying on implicitly potential errors. You can't do much to avoid system-induced exceptions such as `Sys_error`, but if one of your functions can fail in one way or another, you can use OCaml's type system to signify that. The `option` type or writing a `try` monad should be better ideas.

- You should avoid mutable constructions as much as possible. Constant definitions promote code scalability and behaviour validation at compile time. To be more precise, you should be avoiding:
  - the `ref` type
  - mutable records
  - arrays
  - mutable attributes in objects

If your code respects all those guidelines, then congratulations! You're in for some bonus points :)

VI.3 Graphical interface

Your output could be implemented using some graphical interface to make it more visually appealing (in case OCaml wasn't, you know, visually appealing enough). You could draw the automaton and show it recognizing your input step by step, for example. You can use any library you want to implement your graphical interface. Unleash your imagination!

VI.4 Gamepads

Don't blame me, I'm all with you on the PC master race idea. But let's face it, keyboards are a horrible way to play fighting games. You can use the `Sdljoystick` module and implement game pads handling. And when I say game pads, it could be actual game pads or arcade sticks...You are free to do what you want, just make sure you do something awesome.

VI.5 Debug mode

Your program can include a `debug` mode. You can trigger it whatever way you want, but it must be reachable at runtime (i.e. you must run the program in normal and debug modes without having to rebuild the project to switch between modes). The debug mode must at least show your automaton going from state to state and eventually finding (or not) and end state with the rule(s) associated.

```
Butt slam (Ermac)
% ./ft_ality grammars/mk9.gmr
Key mappings:
q -> Block
down -> Down
w -> Flip Stance
left -> Left
right -> Right
e -> Tag
a -> Throw
up -> Up
s -> [BK]
d -> [BP]
z -> [FK]
x -> [FP]
----------------------

[BP]
State 1, "[BP]" -> State 11
Found end state for "Claw Slam (Freddy Krueger)" at: 11
Found end state for "Knockdown (Sonya)" at: 11
Found end state for "Fist of Death (Liu-Kang)" at: 11
Claw Slam (Freddy Krueger) !!
Knockdown (Sonya) !!
Fist of Death (Liu-Kang) !!

[BP], [FP]
State 11, "[FP]" -> State 162
Found end state for "Saibot Blast (Noob Saibot)" at: 162
Found end state for "Active Duty (Jax)" at: 162
Saibot Blast (Noob Saibot) !!
Active Duty (Jax) !!
```

Chapter VII

Turn-in and peer-evaluation

- As usual, only the code on your Git repository will be evaluated.
- This project will only be graded by humans. As such, you are afforded some liberty to infer and interpret details from the subject, whichever way you see fit. But remember that every choice you make may have to be justified and defended to your graders.
- You must also include grammar files with your work; as previously stated you are free to format them the way you want, which means you have to provide them yourself.