EC3204: Programming Languages and Compilers (Fall 2024)
Homework 1: Regex Compiler

Due: 11/8, 23:59 (submit on GIST LMS)

Instructor: Sunbeom So

Important Notes

- **Evaluation criteria**
  The correctness of your implementation will be evaluated using testcases:

  $$\frac{\#\text{Passed}}{\#\text{Total}} \times 100$$

  - "Total": testcases prepared by the instructor (not disclosed before the evaluation).
  - "Passed": testcases where expected outputs match your outputs.

- **Compilable**
  Make sure that your submission is successfully compiled. If your code cannot be compiled, you will get 0 points for the corresponding HW.

- **Academic Integrity**
  Violating academic integrity will result in an F, even after the end of the semester.

  Click here to check the rules related to academic integrity.

  Be aware that proving your academic integrity is entirely your responsibility.

- **No Changes on Templates, File Names, and File Extensions**
  Your job is to complete `(* TODO *)` parts in provided code templates. You should not modify the other templates. Do not change the file names. The submitted files should have `.ml` extensions, not the others (e.g., `.pdf`, `.zip`, `.tar`).

- **Regarding Using LLM-based Tools**
  You can use LLM-based tools to complete your HW, as long as (1) you use LLMs by yourself, (2) you can reproduce all the steps assisted by LLMs, and (3) you can clearly explain all the details of your implementation (including LLM-generated parts).

  Even though you claim that you have completed HW with the help of LLMs, if your submission is highly similar to other students' code and you cannot reproduce certain steps by any reason (including the nondeterminism of LLMs), I will consider that you cheated on your assignments.

1 Goal

Your goal is to implement a compiler that translates a regular expression into an equivalent nondeterministic finite automaton (NFA) and deterministic finite automaton (DFA).

2 Structure of the Project

You can find the following files in the `hw1/code` directory.

- `main.ml`: contains the driver code with testcases where you can add your own. At the top-level, `main.ml` executes the following code:

```
let _ =
  List.iter (fun (regex, str) ->
    let (b1,b2) = match_regex regex str in
    print_endline (string_of_bool b1 ^ ", " ^ string_of_bool b2)
  ) testcases
```

  where `b1` and `b2` are *true* iff an NFA and DFA, converted from a given regex (`regex`), recognize the string (`str`).

- `regex.ml`: contains the definition of our regular expressions.

```
type t =
  | Empty
  | Epsilon
  | Symbol of symbol
  | SymbolSet of symbol * symbol
  | OR of t * t
  | CONCAT of t * t
  | STAR of t
  | POS of t

and symbol = char
```

  `Symbol` and `SymbolSet` are constructors for input symbols and character (symbol) classes, respectively. `STAR` and `POS` are constructors for Kleene closures and Positive closures, respectively. We assume an alphabet $\Sigma = \{a, \cdots, z, A, \cdots, Z, 0, \cdots, 9, \_\}$.

  For example, a regular expression $([a - z] \mid x)^{+}(\_)^{*}$ can be represented as follows:

  `CONCAT (POS (OR (SymbolSet ('a', 'z'), Symbol 'x')), STAR (Symbol '_'))`

- `nfa.ml`: is a module that contains utilities for implementing NFAs.

- `nfa.mli`: is the interface file for `nfa.ml`, which lists the types and functions accessible from external modules (external `.ml` files).

- `dfa.ml`: is a module that contains utilities for implementing DFAs.

- `dfa.mli`: is the interface file for `dfa.ml`, which lists the types and functions accessible from external modules (external `.ml` files).
- `hw.ml`: contains four unimplemented functions: `regex2nfa`, `nfa2dfa`, `run_nfa`, `run_dfa`. **Your job is to complete these four functions and submit this file.**

3 How to Build

(1) Install the dependencies using the following commands.

```
$ sudo apt-get update
$ sudo apt-get install -y opam ocamlbuild ocaml-findlib
$ opam init
$ eval $(opam env)
$ opam update
$ opam install -y batteries
```

(2) Activate the build script by running the below command in the `hw1/code` directory.

```
$ chmod +x build
```

(3) After completing `hw.ml`, run the following command to compile the project.

```
$ ./build
```

Then, the executable `main.native` will be generated. You can run it as follows.

```
$ ./main.native
```

4 Running Example

If you correctly implemented `hw.ml`, you must obtain the following results by running the command `./main.native`. The comments (* $\cdots$ *) are intended to explain the expected results and will not show up during the actual run.

```
false, false   (* $\epsilon \notin L(\emptyset)$ *)
true, true     (* $\epsilon \in L(\epsilon)$ *)
true, true     (* $a \in L(a)$ *)
true, true     (* $b \in L(b)$ *)
true, true     (* $b \in L(a \mid b)$ *)
true, true     (* $b \in L(a^*b)$ *)
true, true     (* $ab \in L(a^*b)$ *)
true, true     (* $aab \in L(a^*b)$ *)
false, false   (* $abb \notin L(a^*b)$ *)
true, true     (* $b \in L((aa)^*b)$ *)
true, true     (* $aab \in L((aa)^*b)$ *)
false, false   (* $aaab \notin L((aa)^*b)$ *)
true, true     (* $aaaab \in L((aa)^*b)$ *)
```

```
false, false  (* b ∉ L((aa)^+ b) *)
true, true    (* aab ∈ L((aa)^+ b) *)
false, false  (* aaab ∉ L((aa)^+ b) *)
true, true    (* aaaab ∈ L((aa)^+ b) *)
true, true    (* gist ∈ L(([a-z] | [A-Z])^+ (_)^*) *)
true, true    (* GiSt ∈ L(([a-z] | [A-Z])^+ (_)^*) *)
false, false  (* Gi_st ∈ L(([a-z] | [A-Z])^+ (_)^*) *)
```