Homework 2
COSE312, Spring 2026

Hakjoo Oh

Due: 4/2, 23:59

Problem 1 Write a compiler that translates regular expressions to deterministic finite automata (DFAs).

1. In the provided template, you can find the following files:

   - `main.ml`: Driver code with some test cases. You can add your own test cases here.
   - `regex.ml`: The definition of regular expressions (the "source language" of our compiler).
   - `nfa.ml`: NFA implementation (the "intermediate representation" of our compiler). Read `nfa.mli` to see how to use the NFA module.
   - `dfa.ml`: DFA implementation (the "target language" of our compiler). Read `dfa.mli` to see how to use the DFA module.
   - `trans.ml`: Complete and submit this file.

2. In `regex.ml`, regular expression is defined as follows:

   ```
   type alphabet = A | B
   type t =
     | Empty
     | Epsilon
     | Alpha of alphabet
     | OR of t * t
     | CONCAT of t * t
     | STAR of t
   ```

   where we assume $\Sigma = \{\texttt{A}, \texttt{B}\}$.

3. In `trans.ml`, you can find code below:

```
open Regex

exception Not_implemented

let regex2nfa : Regex.t -> Nfa.t
=fun _ -> raise Not_implemented (* TODO *)

let nfa2dfa : Nfa.t -> Dfa.t
=fun _ -> raise Not_implemented (* TODO *)

(* Do not modify this function *)
let regex2dfa : Regex.t -> Dfa.t
=fun regex ->
  let nfa = regex2nfa regex in
  let dfa = nfa2dfa nfa in
    dfa

let run_dfa : Dfa.t -> alphabet list -> bool
=fun _ _ -> raise Not_implemented (* TODO *)
```

Your job is to implement the functions:

- `regex2nfa`, which converts a regular expression into an equivalent NFA,
- `nfa2dfa`, which converts an NFA into an equivalent DFA, and
- `run_dfa`, which takes a DFA and a string (i.e., a sequence of input symbols) and returns true (i.e., accept) or false (i.e., reject).

Once you complete the implementation, you can build and run the program as follows:

```
$ dune build main.exe; _build/default/main.exe
```

For the test cases in `main.ml`:

```
let testcases : (Regex.t * alphabet list) list =
  [
    (Empty, []);
    (Epsilon, []);
    (Alpha A, [A]);
    (Alpha A, [B]);
    (OR (Alpha A, Alpha B), [B]);
    (CONCAT (STAR (Alpha A), Alpha B), [B]);
    (CONCAT (STAR (Alpha A), Alpha B), [A;B]);
```

```
    (CONCAT (STAR (Alpha A), Alpha B), [A;A;B]);
    (CONCAT (STAR (Alpha A), Alpha B), [A;B;B]);
    (CONCAT (CONCAT (STAR (CONCAT (Alpha A, Alpha A)),
                     STAR (CONCAT (Alpha B, Alpha B))), Alpha B), [B]);
    (CONCAT (CONCAT (STAR (CONCAT (Alpha A, Alpha A)),
                     STAR (CONCAT (Alpha B, Alpha B))), Alpha B), [A;A;B]);
    (CONCAT (CONCAT (STAR (CONCAT (Alpha A, Alpha A)),
                     STAR (CONCAT (Alpha B, Alpha B))), Alpha B), [B;B;B]);
    (CONCAT (CONCAT (STAR (CONCAT (Alpha A, Alpha A)),
                     STAR (CONCAT (Alpha B, Alpha B))), Alpha B), [A;A;A;A;B;B;B]);
    (CONCAT (CONCAT (STAR (CONCAT (Alpha A, Alpha A)),
                     STAR (CONCAT (Alpha B, Alpha B))), Alpha B), [A;A;A;B;B;B])
  ]
```

the correct output is the following:

```
false
true
true
false
true
true
true
true
false
true
true
true
true
true
false
```

Problem 2 Implement a top-down parser. Complete and submit the `parse.ml` file.

Context-free grammars can be defined in OCaml as follows.

```
type symbol =
  | T of string  (* terminal symbol *)
  | N of string  (* nonterminal symbol *)
  | Epsilon      (* empty string *)
  | End          (* end marker *)
type production = symbol * symbol list
type cfg = symbol list * symbol list * symbol * production list
```

For instance, the grammar

$$(\{E, E', T, T', F\}, \{+, *, (,), \mathbf{id}\}, E, \left\{ \begin{array}{lcl} E & \rightarrow & T\ E' \\ E' & \rightarrow & +\ T\ E' \mid \epsilon \\ T & \rightarrow & F\ T' \\ T' & \rightarrow & *\ F\ T' \mid \epsilon \\ F & \rightarrow & (E) \mid \mathbf{id} \end{array} \right\})$$

can be written as follows:

```
let cfg1 = (
  [N "E"; N "E'"; N "T"; N "T'"; N "F"],
  [T "+"; T "*"; T "("; T ")"; T "id"],
  N "E",
  [
    (N "E", [N "T"; N "E'"]);
    (N "E'", [T "+"; N "T"; N "E'"]);
    (N "E'", []);
    (N "T", [N "F"; N "T'"]);
    (N "T'", [T "*"; N "F"; N "T'"]);
    (N "T'", []);
    (N "F", [T "("; N "E"; T ")"]);
    (N "F", [T "id"])
  ])
```

A string is represented by a list of symbols. For example, string $\mathbf{id} + \mathbf{id} * \mathbf{id}$ can be represented by the following (the last symbol in a string must be `End`):

```
let s1 = [T "id"; T "+"; T "id"; T "*"; T "id"; End]
```

1. Write function `check_LL1` that checks whether the given grammar belongs to LL(1) or not.

   ```
   check_LL1 : cfg -> bool
   ```

   For example, `check_LL1 cfg1` should evaluate to `true`. Consider the following grammars.

   ```
   let cfg2 = (
     [N "S"; N "E"; N "E'"; N "T"; N "T'"; N "F"],
     [T "+"; T "-"; T "*"; T "/"; T "id"; T "num"; T "("; T ")"],
     N "S",
     [
         (N "S", [N "E"]);
         (N "E", [N "T"; N "E'"]);
         (N "E'", [T "+"; N "T"; N "E'"]);
         (N "E'", [T "-"; N "T"; N "E'"]);
         (N "E'", []);
         (N "T", [N "F"; N "T'"]);
         (N "T'", [T "*"; N "F"; N "T'"]);
         (N "T'", [T "/"; N "F"; N "T'"]);
         (N "T'", []);
         (N "F", [T "id"]);
         (N "F", [T "num"]);
         (N "F", [T "("; N "E" ;T ")"]);
     ]
   )
   ```

```
let cfg3 = (
  [N "X"; N "Y"; N "Z"],
  [T "a"; T"c"; T"d"],
  N "X",
  [
    (N "X", [N "Y"]);
    (N "X", [T "a"]);
    (N "Y", [T "c"]);
    (N "Y", []);
    (N "Z", [T "d"]);
    (N "Z", [N "X"; N "Y"; N "Z"])
  ]
)

let cfg4 = (
  [N "S"; N "S'"; N "E"],
  [T "a"; T "b"; T "e"; T "i"; T "t"],
  N "S",
  [
   (N "S", [T "i"; N "E"; T "t"; N "S"; N "S'"]);
   (N "S", [T "a"]);
   (N "S'", [T "e"; N "S"]);
   (N "S'", []);
   (N "E", [T "b"])
  ]
)
```

`check_LL1 cfg2` evaluates to `true`. `check_LL1 cfg3` evaluates to `false`. `check_LL1 cfg4` evaluates to `false`.

2. Write function `parse` that takes a grammar and a string, and performs a top-down parsing.

```
parse : cfg -> symbol list -> bool
```

Assuming that the first argument of `parse` is LL(1), `parse` should return `true` if the string given as the second argument belongs to the grammar. Otherwise, it should evaluates to `false`. Examples:

```
let s1 = [T "id"; T "+"; T "id"; T "*"; T "id"; End]
let s2 = [T "id"; T "/"; T "("; T "num"; T "+"; T "id"; T ")"; End]
```

`parse cfg1 s1`, `parse cfg1 s2`, `parse cfg2 s1`, `parse cfg2 s2` evaluate to `true`, `false`, `true`, and `true`, respectively.