

Homework 2

COSE312, Spring 2023

Hakjoo Oh

Due: 04/09, 23:59

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

1. Template code:

`https://github.com/kupl-courses/COSE312-2023spring/tree/main/homework/hw2/regex2dfa`

2. 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.

3. 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 $\Sigma = \{A, B\}$.

4. 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
false

```

Problem 2 수업시간에 다른 하향식 파서(top-down parser)를 구현해 봅시다. 아래 템플릿 코드 가운데 `parse.ml`파일을 완성해서 제출하세요:

<https://github.com/kupl-courses/COSE312-2023spring/tree/main/homework/hw2/parser>

OCaml 자료형으로 문맥 자유 문법(context-free grammar)을 다음과 같이 정의할 수 있습니다.

```

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

```

예를 들어, 다음의 문법

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

은 아래와 같이 나타낼 수 있습니다:

```

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"])
  ]
)

```

문자열은 기호(symbol)의 리스트로 나타냅니다. 예를 들어, 문자열 `id + id * id`는 다음과 같이 표현합니다(문자열의 가장 마지막 기호는 `End`이어야 합니다):

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

1. 주어진 문법이 LL(1)에 속하는지 확인하는 함수 `check_LL1`을 작성해 봅시다.

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

예를 들어, `check_LL1 cfg1`은 `true`를 반환해야 합니다. 아래와 같은 문법들에 대해서

```

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 cfg2는 true를, check_LL1 cfg3는 false를, check_LL1 cfg4는 false
를 반환해야 합니다.

```

- 문법과 문자열을 받아서 하향식 파싱을 수행하는 함수 `parse`를 작성해 봅시다.

```

parse : cfg -> symbol list -> bool

```

`parse`의 첫번째 입력으로 주어진 문법이 LL(1)이라 가정할 때, 두번째 입력으로 주어진 문자열이 해당 문법이 규정하는 언어에 속하면 `true`, 아니면 `false`를 반환합니다. 예를 들어, 아래 문자열들에 대해서

```
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 s2는 각각 true,
false, true, true를 반환해야 합니다.
```