type regexp =

|Empty

|Epsilon

|Lettre of char

|Union of regexp \* regexp

|Concat of regexp \* regexp

|Kleene of regexp

(\*Etape 1 \*)

(\*Pour linearliser l'expression, On peut utiliser un tableau de caracteres tels que les indices signifient les numerotations \*)

let list\_to\_tab l1 =

let n = List.length l1 in

let tab = Array.make n ' ' in

let l = ref l1 in

for i = 0 to n-1 do

match l1 with

|[] -> ()

|x::xs ->

begin

tab.(i) <- x;

l := xs

end

done;

tab

;;

(\*C'est plus facile de transformer expression reguiliers a liste de caracteres avant de mettre au tableau\*)

let linearliser e =

let rec regexp\_to\_list e =

match e with

|Empty |Epsilon -> []

|Lettre a -> [a]

|Union(e1,e2)|Concat(e1,e2) ->

(regexp\_to\_list e1)@(regexp\_to\_list e2)

|Kleene e ->

regexp\_to\_list e

in

list\_to\_tab (regexp\_to\_list e)

;;

(\*Etape 2\*)

(\*Pour representer les prefixes, suffixes, facteurs, en utilisant les numerotations de lettres dans le tableau, on cree liste des indices de prefixes,suffixes,facteurs \*)

(\*On change le type de regexp, mais explin car ce qu'on interesse c'est la numeration plutot que le lettre\*)

type explin =

|Empty

|Epsilon

|Number of int

|Union of explin \* explin

|Concat of explin \* explin

|Kleene of explin

let rec regexp\_to\_explin (e:regexp): explin =

let id = ref 0 in

match e with

|Empty -> Empty

|Epsilon -> Epsilon

|Lettre a ->

incr id;

Number(!id)

|Union(e1,e2) ->

Union(regexp\_to\_explin e1, regexp\_to\_explin e2)

|Concat(e1,e2) ->

Concat(regexp\_to\_explin e1, regexp\_to\_explin e2)

|Kleene e ->

Kleene(regexp\_to\_explin e)

;;

let rec empty\_word e =

match e with

|Empty -> true

|Epsilon |Number \_ -> false

|Union (e1,e2) -> empty\_word e1 || empty\_word e2

|Concat (e1,e2) -> empty\_word e1 && empty\_word e2

|Kleene e -> false

;;

let rec have\_eps e =

match e with

|Epsilon |Kleene \_ -> true

|Union (e1,e2) -> have\_eps e1 || have\_eps e2

|Concat(e1,e2) -> have\_eps e1 && have\_eps e2

|Number \_ |Empty -> false

;;

let rec union l1 l2 =

match l1 with

|[] -> l2

|x::xs ->

if List.mem x l2 then

union xs l2

else

x::union xs l2

;;

let rec suffixe e =

match e with

|Empty |Epsilon -> []

|Number a -> [a]

|Union(e1,e2) ->

union (suffixe e1) (suffixe e2)

|Concat(e1,e2) ->

if have\_eps e2 then

union (suffixe e1) (suffixe e2)

else

suffixe e2

|Kleene e -> suffixe e

;;

let rec prefixe e =

match e with

|Empty |Epsilon -> []

|Number a -> [a]

|Union(e1,e2) ->

union (prefixe e1) (prefixe e2)

|Concat(e1,e2)->

if have\_eps e1 then

union (prefixe e1) (prefixe e2)

else

prefixe e1

|Kleene e -> prefixe e

;;

let rec facteurs e =

let rec produit l1 l2 =

match l1 with

|[] -> []

|x::xs ->

List.map (fun y -> (x,y)) l2

@ produit xs l2

in

match e with

|Empty -> []

|Epsilon -> []

|Number \_ -> []

|Union(e1,e2) ->

union (facteurs e1) (facteurs e2)

|Concat(e1,e2) ->

if empty\_word e1 || empty\_word e2 then

union (facteurs e1) (facteurs e2)

else

union (union (facteurs e1) (facteurs e2)) (produit (suffixe e1) (prefixe e2))

|Kleene e ->

union (facteurs e) (produit (prefixe e) (suffixe e))

;;

(\*Etape 3\*)

(\*On doit construire automate par rapport aux prefixes, suffixes, facteurs\*)

type afd = {

nb\_etats : int;

initiaux: bool array;

terminaux : bool array;

transitions: int array array }

;;

(\*on peut avoir plusieurs etats initiaux car plusieurs prefixes donc on le represente comme les terminaux bool array\*)

let create\_automate (e:regexp)(p:int list)(s:int list)(f:(int\*int)list): afd =

let tab = linearliser e in

let n = Array.length tab in

let tab\_init = Array.make n false in

let tab\_term = Array.make n false in

let tab\_transitions = Array.make\_matrix n n (-1) in

let rec prefixe p =

match p with

|[] -> ()

|x::xs ->

begin

tab\_init.(x) <- true;

prefixe xs

end

in

let rec suffixe s =

match s with

|[] -> ()

|y::ys ->

begin

tab\_term.(y) <- true;

suffixe ys

end

in

let rec facteurs f =

match f with

|[] -> ()

|(x,y)::xs ->

begin

tab\_transitions.(x).(y) <- y;

facteurs xs

end

in

prefixe p;

suffixe s;

facteurs f;

{nb\_etats = n;

initiaux = tab\_init;

terminaux = tab\_term;

transitions = tab\_transitions}

;;

(\*Etape 4\*)

(\*Pour effacer des marques, on a besoins de remplacer les numerotations par la lettre dans le tableau de lineralisation\*)

type afnd = {

nb\_etats : int;

initiaux: bool array;

terminaux : bool array;

transitions: char array array }

;;

let erase\_marks\_afd (a:afd) (e:regexp) =

let tab = linearliser e in

let n = Array.length tab in

let afnd\_transitions = Array.make\_matrix n n ' ' in

for i = 0 to n-1 do

for j = 0 to n-1 do

if a.transitions.(i).(j) <> -1 then

afnd\_transitions.(i).(j) <- tab.(j);

done

done;

{nb\_etats = a.nb\_etats ;

initiaux = a.initiaux;

terminaux = a.terminaux;

transitions = afnd\_transitions}