Calcul formel

DURAND Ulysse

Les expressions ou morceaux de code OCaml seront `sous cette forme`.
Les chaînes de caractères seront *sous cette forme*.

Le but est de faire ce qui peut s'apparenter à un logiciel de calcul formel en OCaml, il doit pouvoir évaluer des expressions, les dériver, et pouvoir les afficher en LaTeX.
Un objectif est aussi de pouvoir renseigner au programme les expressions des fonctions à traiter en LaTeX.
En complément, le programme peut aussi partiellement simplifier des expressions. Ansi on peut simplifier $(0+1) * x_0$ par x_0
On aura à la fin, créé ces fonctions :

```
val evalue : expression -> float array -> float = <fun>
val affiche : expression -> string = <fun>
val derive : int -> expression -> expression = <fun>
val latex_vers_expression : string -> expression = <fun>
val simplifie : expression -> bool * expression = <fun>
```

(`simplifie` retourne aussi un booléen, il sera à `true` si l'expression donnée est simplifiable et `false` sinon.)

Voci un exemple de ce dont le programme est capable :
Le code suivant :

```
let exprlatexa = "x_0^{x_1}+x_1^{x_0}";;

let expra = latex_vers_expression exprlatexa;;
(*parse le latex*)

let exprb = (derive 0 expra);;
(*derive l'expression par rapport a x_0*)

let exprbsimpl = snd (simplifie exprb);;
(*simplifie l'expression obtenue*)

let exprlatexb = affiche exprbsimpl;;
(*transforme l'expression en latex*)

print_string (exprlatex b);;
```

Retournera :
(frac{{x_{1}}}{{x_{0}}})({x_{0}} {x_{1}})+(ln({x_{1}}))*({x_{1}}{x_{0}})*
qui s'affiche ainsi en latex :
$\left(\frac{x_1}{x_0}\right) * ({x_0}^{x_1}) + (ln(x_1)) * ({x_1}^{x_0})$
Et il s'agit bien de la dérivée par rapport à x_0 de $x_0^{x_1} + x_1^{x_0}$.

Remarque : Sans la simplification de l'expression dérivée, on aurait ce résultat, juste mais peu accueillant.
$((0.) * (\ln(x_0)) + (x_1) * ((1.) * (\frac{1}{x_0}))) * (exp((x_1) * (\ln(x_0)))) + ((1.) * (\ln(x_1)) + (x_0) * ((0.) * (\frac{1}{x_1}))) * (exp((x_0) * (\ln(x_1))))$

1 Représentation des fonctions

Nous traiterons les expressions algébriques en les représentant par des arbres d'expression.

l'arbre Figure1 représente l'expression algébrique $\dfrac{x+3}{-y}$:

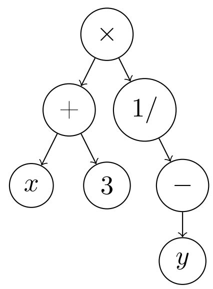


Figure 1: Exemple d'arbre d'expression

On discernera :

- Les feuilles qui sont des variables ou des constantes
- Les nœuds qui sont de la forme (une opération, des fils)

d'où une telle implémentation en OCaml :

```
type operation = {nbvar : int ;affichage : string ; evaluation : float array->float ; derive
     : deriv}
and expression =
    C of float |
    V of int |
    F of operation * (expression array)
and deriv = NonDeriv | Deriv of ((expression array)->(expression -> expression) ->
    expression);;
```

Les variables sont caractérisées par un entier $(x_i, i \in [|0, n-1|])$ où n est nbvar, le nombre de variable en entrée de l'opération (son nombre de fils).
Nous allons détailler les caractéristiques `affichage` , `evaluation` et `derive` des opérations et le type `deriv` .

2 affichage

Construisons une fonction `affiche` : `expression -> string` , telle que si a est l'arbre d'expression Figure1, alors `affiche a` retourne *`frac{x+3}{-y}`*
Pour expliciter comment afficher une expression, nous allons, à chaque opération, associer un code dans un language inventé (nous allons l'appeler LanguInv), ce code étant sous forme d'une chaîne de caractères, il décrit la forme de l'affichage d'une opération.

2.1 le language LanguInv

Ce language inventé a pour but d'exprimer des motifs. C'est à dire, une forme de chaîne de caractère, avec ce qui s'appelle des placeholder, c'est à dire des emplacements libres. Par la suite, soit on replacera ces placeholder par des chaînes de caractères, par exemple pour afficher du LaTeX(dans cette partie), soit on reconnaîtra le motif et alors on retournera les chaînes de caractères à la place des placeholders (dans le parser).

L'usage le plus courant qu'on aura, c'est celui où on renseigne un seul placeholder à la fois. Alors la

syntaxe pour désigner le placeholder i sera `%|i;%`.
On a déjà de quoi construire le motif latex de l'opération d'addition : `%|0;% + %|1;%`.
Alors on veut par exemple que notre fonction `affiche`, appliquée à `(F(plus,[|a ; b|]))` où `a` et `b` sont du type `expression`, retourne `(affiche a)` `+` `(affiche b)`.
Notre fonction aura remplacé dans le motif du plus, le placeholder 0 par l'affichage du fils 0, soit `a`, et le placeholder 1 par l'affichage du fils 1, soit `b`.

Une fonctionnalité plus avancée du language est celle de pouvoir renseigner plusieurs placeholder à la fois, séparés par une chaîne de caractères appelée délimiteur. La syntaxe est la suivante : `%delimiteur|i-j;%` pour désigner les placeholder de i à j-1, séparés par le delimiteur.
Ainsi, les deux expressions en LanguInv suivantes correspondent au même motif :
`determinant[%;,;|0-4;%]`
`determinant[%|0;%;,;%|1;%;,;%|2;%;,;%|3;%]`

Une toute dernière fonctionnalitée : si j n'est pas renseigné, alors il a pour valeur de base `ope.nbvar`, et si i n'est pas renseigné, alors il a pour valeur de base 0.

2.2 Interpreteur de ce language

Comment faire comprend à OCaml ce language ?
Nous allons utiliser un type d'automate particulier qui nous permettra de faire l'automate `(autospecial n rendph s)` qui reconnaît un tel language, ce type particulier d'automate sera appelé ici automate déterministe qui écrit avec mémoire.
Il écrit, c'est à dire qu'il reconnaît des mots mais renvoit aussi une chaîne de caractères.
On peut donc associer à un tel automate une fonction (ce sera $\eta^\star((i, mem_i), \cdot)$, explications dans la section 3).
On fera la correspondance par une fonction `fonction_d_auto : automate_quiecrit -> (char list -> char list)`.
(On confond `string` et `char list`)

L'automate `autospecial n rendph s` sera associé à une fonction `evaluelatex = fonction_d_auto autospecial`.
`autospecial` dépend des paramétres suivants :

- `ope.nbvar` le nombre de variables de l'opération
- une fonction `rendph` (rendu placeholder), qui elle, prend un entier i en entrée et qui lui associe par quoi il faut remplacer le placeholder i, ici, `(rendph i)` retournera simplement `x_` i
- `ope.affichage` : notre code en LanguInv pour le motif de l'opération

`evaluelatex` retournera l'affichage voulu pour notre opération.
Voici le typage de `evaluelatex` : `int -> (int->char list) -> char list -> char list`
Par exemple, `evaluelatex 3 string_of_int "ope(%;|-;%)"` retournera : `ope(0;1;2)` (On confond `string` et `char list`)
Les détails de l'automate et de sa fonction, `evaluelatex`, associée, seront explicités dans la section 3.2.

3 De quoi construire la fonction `evaluelatex`

3.1 Automates utiles

En OCaml, voici une manière d'implémenter les automates :

```
type ('q, 's) automate_nondet = ('q list)*('q -> bool)*('q*'s*('q list));;
type ('q, 's) automate_det = ('q)*('q -> bool)*('q*'s*'q);;
(*('q,'s) automate_* = (i,f,delta) correspond a un automate ('q,'s,i,f,delta)*)
(*'q : ensembles des etats, 's : alphabet des mots a reconna\^itre, i : etats initiaux, f :
    etats finaux, delta : fonction de transition*)
```

3.1.1 Automate déterministe qui écrit

Definition 3.1 (Automate déterministe qui écrit)**.** Ce type d'automate permet de transformer un mot en un autre.

$$\mathcal{A} = (Q, \Sigma_1, \Sigma_2, i, F, \delta, \eta)$$

Q est l'ensemble des états, comme dans un automate standard.
Σ_1 est l'alphabet d'entrée, comme dans un automate standard.
Σ_2 est l'alphabet de sortie, c'est dans cet alphabet que l'automate écrira.
δ est la fonction de transition, comme dans un automate standard.
η est la fonction d'écriture, ressemblant à δ, à un état $q \in Q$ et une lettre $l \in \Sigma_1$, elle associera un mot à écrire $m \in (\Sigma_2)^\star$.

$$i \in Q, F \in \mathcal{P}(Q)$$

$$\begin{aligned}\delta : Q \times \Sigma_1 &\to Q \\ \eta : Q \times \Sigma_1 &\to (\Sigma_2)^\star\end{aligned}$$

$$\begin{aligned}\delta^\star : Q \times \Sigma_1^\star &\to Q \\ (q, \epsilon) &\mapsto q \\ (q, l.m) &\mapsto \delta^\star(\delta(q,l), m), \text{ avec } l \in \Sigma_1\end{aligned}$$

$$\begin{aligned}\eta^\star : Q \times \Sigma_1^\star &\to (\Sigma_2)^\star \\ (q, \epsilon) &\mapsto \epsilon' \\ (q, l.m) &\mapsto \eta(q,l).\eta^\star(\delta(q,l), m), \text{ avec } l \in \Sigma_1\end{aligned}$$

Implémentation en OCaml :

```
type ('q, 's, 't) automatequiecrit = ('q*('t list), 's ) automate;;
```

`('q,'sig1,'sig2) automatequiecrit = (i,f,g)` correspond à un automate $('q,'s,'t,i,f,delta,eta)$.
Si $delta(q,l) = q'$ et que $eta(q,l) = p$, alors g est telle que $g((q,m),l) = (q', p :: m)$.

correspondance entre automate qui écrit et fonction de $(\Sigma_1)^\star$ **dans** $(\Sigma_2)^\star$:
Ľe but de l'automate qui écrit est de créer une fonction de $(\Sigma_1)^\star$ dans $(\Sigma_2)^\star$.
Pour l'automate $\mathcal{A} = (Q, \Sigma_1, \Sigma_2, i, F, \delta, \eta)$, cette fonction est la fonction

$$\begin{aligned}(\Sigma_1)^\star &\to (\Sigma_2)^\star \\ m &\mapsto \eta^\star(i, m)\end{aligned}$$

3.1.2 Automate avec mémoire

Ce type d'automate n'a pas de d'intérêt théorique, mais il permet de mieux se représenter des automates un peu compliqués.

Definition 3.2 (Automate qui écrit avec mémoire)**.** :
C'est un automate standard où pour ensemble d'états on prend $(Q \times S_m)$.
Pour un état (q, m) d'un tel automate, on appelera $q \in Q$ l'état pur et $m \in S_m$ l'état mémoire.

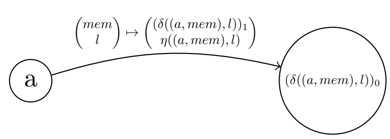
Ainsi, on se représente toujours les automates avec les états de Q mais maintenant, avec un état de mémoire associé.
Maintenant pour un automate avec mémoire $\mathcal{A} = (Q \times S_m, \Sigma, i, F, \delta)$,
$i \in (Q \times S_m)$,
$F \in \mathcal{P}(Q \times S_m)$,
$\delta : (Q \times \S_m) \times \Sigma \to (Q \times S_m)$.

3.1.3 Automate qui écrit avec mémoire

On peut maintenant simplement combiner la définitions d'un automate qui écrit et celle d'un automate avec mémoire pour avoir celle d'un automate qui écrit avec mémoire.
Un tel automate sera sous cette forme : $\mathcal{A} = (Q \times S_m, \Sigma_1, \Sigma_2, i, F, \delta, \eta)$
On aura pour cet automate les fonctions $\delta^\star$ et $\eta^\star$.
Implémentation en OCaml :

```
type ('q, 'm, 's, 't) automate_quiecrit_avecmemoire = ('q*'m,'s,'t) automatequiecrit
```

Pour les représenter graphiquement, nous utiliserons, comme les automates, un graphe orienté, mais avec des annotation sur les arêtes différentes :



(L'indice 0 désigne le premier élement du couple qu'est l'état $\delta((a, mem), l)$, c'est l'état pur. l'indice 1 est pour désigner le deuxième élement de ce même couple, c'est l'état mémoire.)

3.2 Et dans notre cas

Voici l'automate `autospecial n rendph s` (correspond à $\delta^\star(i, mem_i, s)$ pour l'automate suivant):
avec n un entier et $N = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$

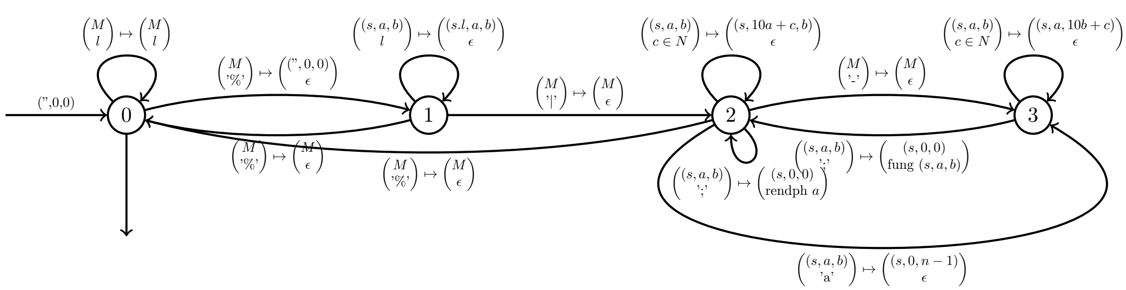


Figure 2: Automate de `evaluelatex`

Notre automate qui écrit avec mémoire est déterministe, il a pour ensemble d'états pur $\{0, 1, 2, 3\}$, pour ensemble d'états mémoire `(char list)*int*int`, le `char list` sert à stocker le délimiteur en mémoire, et `int*int` à stocker le i et j désignants les bornes de l'ensemble des indices de placeholder. L'automate a omme état initial 0, comme ensemble d'états finaux $\{0\}$ et comme alphabet de départ et d'arrivée, `char`.
Si A est cet automate qui écrit, `n = 6`, `rendph a` retourne *x_* a (a entier)
fung est défini ainsi en OCaml :

```
let rec fung (s,a,b) =
    if a > b then [] else
    if a = b then rendph a
    else (rendph a)@s@(fung (s,a+1,b)) in
```

Par exemple, `fung (s,0,3) = (rendph 0)^s^(rendph 1)^s^(rendph 2)^s^(rendph 3)`

On arrive à ce résultat, satisfaisant :
$\eta^\star((0, ('', 0, 0)),$ *frac{{%|0;%+%|15;%}^{(%*|1-4;%)}}{%+|a;%}*$) =$
frac{{x_0+x_{15}}^{x_1*x_2*x_3}}{x_0+x_1+x_2+x_3+x_4+x_5}

Explications partielles :
Sur l'état 0, l'automate recopie ce qui rentre tant qu'on ne tente pas de mettre un placeholder.
Quand on veut mettre un placeholder, c'est quand on reconnaît le *%*, alors on passe dans la partie de traitement du placeholder, soit les états 1, 2, 3.
Dans cette partie de traitement du placeholder, on n'écrit que lorsqu'on revient à l'état 0 (en reconnaîssant un autre *%*), alors on écrit ce que l'on veut à la place du placeholder, via les fonctions `rendph` et `fung`.
Dans cette partie, l'ensemble des transitions internes permettent de changer la mémoire en interprétant ce qui était voulu pour délimiteur et pour valeurs de i et j.

3.2.1 Resultat

Maintenant qu'on a la fonction `evaluelatex`, on peut enfin construire notre fonction `affiche` :

```
let rec affiche f = match f with
    |C(x) -> string_of_float x
    |V(i) -> "{x_{"^(string_of_int i)^"}}"
    |F(g,va) -> evaluelatex (g.nbvar) (fun a -> List.rev (explode (affiche va.(a)))) (g.
    affichage);;
```

4 Evaluation

Construisions une fonction evalue qui prend en entrée une expression et un vecteur en lequel l'évaluer, donc du type : `expression -> float array -> float`, telle que si a est l'arbre d'expression Figure1, alors `evalue a [|2.;3.|]` retourne `-1.6666`
L'évaluation est alors plutôt simple, se faisant par induction.

- Pour une constante, on retourne la constante
- Pour la variable x_i on retourne `v.(i)`
- Pour une opération sur plusieurs expressions, on retourne la fonction d'évaluation de l'opération appliquée à l'évaluation de chaque fils.

d'où une telle implémentation en OCaml :

```
let rec evalue f v = match f with
    |C(x) -> x
    |V(i) -> v.(i)
    |F(g,fa) -> g.evaluation (Array.map (fun unef -> evalue unef v) fa );;
```

5 Dérivation

Pour renseigner pour une opération si elle est dérivable et comment évaluer sa dérivée, nous allons utiliser le type dérivation défini en première page que nous rappelons :
`deriv = NonDeriv | Deriv of (( expression array)->(expression -> expression) ->expression)`.
Soit l'opération n'est pas dérivable, soit elle l'est et alors on donne une expression de sa dérivée.
Pour l'expression de sa dérivée nous aurons besoin d'utiliser la fonction `derive` à l'intérieur, qui sera alors renseignée dans Deriv (c'est le `expression -> expression`). Le `expression array` correspond aux fils de la fonction qui auront leur rôle dans l'expression de la dérivée.
Alors on a `Derive` prenant en paramètres `ar` et `d`, `ar` étant les fils de l'opération et `d` étant la fonction de dérivation.

Un exemple sur l'opération plus,
`plus.derive = Deriv(fun ar d -> F(plus,[|d ar.(0);d ar.(1)|]) )`.
On retrouve $(u + v)' = u' + v'$
Un deuxième exemple, sur l'opération de multiplication,
`fois.derive = Deriv(fun ar d -> F(plus,[| F(fois,[|d ar.(0);ar.(1)|]) ; F(fois,[|ar.(0);d ar.(1)|]) |]))`.
On retrouve $(u * v)' = u' * v + u * v'$

Pour ce qui est de la fonction `derive`, il y a plusieurs fonctions de dérivation que l'on pourra renseigner dans le type Derive de notre opération, ce sont les dérivées par rapport aux différentes variables.
C'est pourquoi nous allons prendre en arguments un entier `k` et une expression `f` pour retourner l'expression de la dérivée selon la variable x_k.
Si `f` est la variable x_i, alors on retourne $\delta_{k,i}$.
Si `f` est une constante, alors on retourne le flottant `0`.
Si `f` est une opération `g` dont les fils sont `va` et que `g.derive` est `NonDeriv`, on renvoit une erreur mais si c'est `Deriv(laf)` alors tout est renseigné dans `laf`, il n'y a qu'a retourner `laf va (derive k)`
Voici l'expression de `derive` en OCaml :

```
let rec derive k f = match f with
  |C(x)->C(0.);
  |V(i)-> if i=k then C(1.) else C(0.);
  |F(g,va) -> match g.derive with
    |NonDeriv->failwith "Fonction non derivable !";
    |Deriv(laf)->laf va (derive k);;
```

6 Parser

Le plus dur reste à faire : transformer une expression `entree` en LaTeXen un arbre d'expression. L'idée est de faire du pattern matching, mais sur une chaîne de caractères. En effet, on voudrait une application recursive `parselatex` telle que par exemple
`parselatex "` x + y `"` retourne `F(plus,[|parselatex` x `; parselatex` y `|] )`
Pour se faire, pourquoi ne pas réutiliser notre language défini section 2.1, LanguInv. On voudrait par exemple ici reconnaître le pattern suivant en LanguInv : `%|0;%+%|1;%`.

L'idée est la suivante : pour chaque pattern `p` en LanguInv, utiliser une fonction associée à un automate que nous expliciterons plus loin `automate_de_pattern p` qui retournerait un automate qui écrit reconnaîssant `entree` et ressortant les chaines de caractères dans `entree` à la place des placeholders de notre motif en LanguInv.

Pour le pattern du plus et pour *banane+pomme* comme valeur pour `entree`, l'automate du pattern de plus reconnaîtrait `entree` et ressortirait un tableau `[|"banane","pomme"|]` auquel il ne resterait qu'à associer `F(plus,[|parselatex "banane" ; parselatex "pomme"|])`
La difficultée réside en la construction d'un tel automate.
Un détail : plusieurs pattern pourraient être reconnus, alors on utilisera seulement la sortie de l'automate du premier de ces pattern dans notre liste de pattern. Ils seront ainsi priorisés.

La tache sera découpée en deux étapes, il faut construire l'automate à partir d'un motif en LanguInv, puis appliquer la fonction associée à cet automate construit à du latex comme décrit précedement. Construisons alors les fonctions `automate_de_pattern` et `parselatex`.

6.1 `automate_de_pattern`

Comme dit précedemment, `automate_de_pattern` sera la fonction associée à un automate qui ecrit (qui sera appelé `autoMagique`) ressortant l'automate voulu.
Par nécessité, nous allons géneraliser le type de sortie d'un automate qui écrit, en introduisant les espaces d'écriture.

6.1.1 Espace d'écriture

Quand on a un automate qui écrit, l'espace dans lequel il écrit a besoin de trois choses :

- un type `'a`
- une opération appelée concaténation de type `'a -> 'a -> 'a`
- un neutre `e` de type `'a` tel que ∀ `x` ∈ `'a`, `concat x e = x`

Par exemple pour l'automate de `evaluelatex`, l'espace d'ecriture est celui des mots, que l'on appelera espace d'écriture `classique`, alors son type est `char list`, sa concaténation est `fun a b -> a@b` et son neutre est `[]`.

Ici nous aurons besoin des espaces d'écriture suivants : `dansph` (ph pour placeholder) et autoprio.
`dansph` est alors défini ainsi :

```
type type_dansph = (char list) array

let arrayvide n = Array.make n []

let concat_dansph x y =
  let n = (Array.length x) in
  let res = Array.make n [] in
  for i=0 to n-1 do res.(i)<-x.(i)@y.(i);done;
  res

let dansph n = {neutre = arrayvide n; operation = concat_dansph}
```

Pour ce qui est de `autoprio`, il sera bien plus long à expliquer, cela constituera alors une prochaine partie.

6.2 `autoMagique`

L'automate retourné par autoMagique doit alors reconnaître un `char list` et son espace d'écriture sera `dansph`, ainsi dans le tableau qu'il retournera, à l'indice i on retrouvera ce qu'il y a dans à la place du placeholder i. Pour reconnaître par exemple le motif en LanguInv suivant : `frac%|0;%%|1;%`, le premier automate imaginé était celui figure 3. Mais avec cet automate déterministe, on rencontre vite un problème.

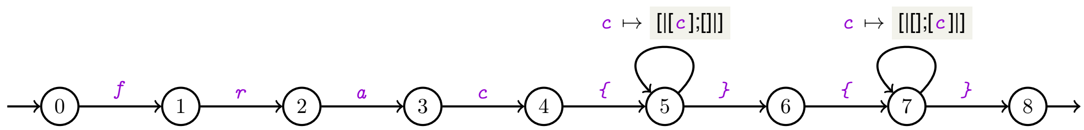


Figure 3: Automate naïf

Pour `frac{{aHA}}{bab}`, l'automate retournera à l'état 6 `[|"{aHA",[]|]`, et sera bloqué.
Pour remédier à ce problème, l'idée est de passer par un automate non déterministe, où à l'état 5, pour la lettre `}`, il y ait une transition vers l'état 6 et une vers l'état 5.
Ainsi deux parcours de l'automate seront possibles :

- Le parcours décrit précédement, qui échoue à l'étape 6
- Le parcours qui reste à 5 après avoir reconnu une première fois `{`

Ce dernier parcours reconnaît bien le mot et retournera le résultat attendu : `[|"aHA";"bab"|]`
Alors c'est maintenant la question de comment construire cet automate qui écrit avec un autre automate qui écrit, l'autoMagique.

L'idée est la suivante : On va assembler des automates avec des opérations de concaténation bien spécifiques. Tout va résider dans la définition de l'espace d'ecriture d'autoMagique, que l'on appelera `autoprio`.

6.3 L'espace d'écriture `autoprio`

6.3.1 Le type et le neutre

Dans cet `autoprio`, on aura deux "briques" principales, et si on concatène une "brique" A à une "brique" B, on veut pas la même opération de concaténation qu' en concaténant une "brique" B à une "brique" A.
Alors on aura un type `typautoprio` en deux sous type (chacun associé à une "brique").

La première "brique" sera appellée `autobrique` et la deuxième `recoph`.

TODO : AJOUTER FIGURE AUTOBRIQUE ET RECOPH

Alors, associé à `autobrique` nous aurons le type `AutDom` (automate dominant), dans ce type on aura un `char` de renseigné (correspondant à un caractère à reconnaitre pour aller d'un état i à un état $i+1$), en plus d'un automate non déterministe qui écrit.
Associé à `recoph`, nous aurons le type `Aut` (automate), dans ce type nous aurons un `int` de renseigné (correspondant à un indice de placeholder), ainsi qu'un automate non déterministe qui écrit Enfin nous aurons un `AutoVide`, neutre pour notre concaténation.

6.3.2 La concaténation

Pour l'opération de concaténation de `a` et `b` (a,b de type `typautoprio`), nous allons discerner les 4 cas suivants, utilisants les fonctions `ajoufin` et `agraphefin` :

- `a = Aut(autoa)` et `Aut(b =autob)`, alors `concat_autoprio a b` retournera
  `a`

- `a = AutDom(la,autoa)` et `b = Aut(autob)`, alors `concat_autoprio a b` retournera `Aut(agraphefin autoa autob)`
- `a = Aut(autoa)` et `b = AutDom(lb,autob)`, alors `concat_autoprio a b` retournera `AutDom('%',ajoufin autoa autob lb)`
- `a = AutDom(la,autoa)` et `b = AutDom(lb,autob)`, alors `concat_autoprio a b` retournera `AutDom('%', ajoufin autoa autob lb)`

  `ajoufin auta autb l` retournera l'automate où on "ajoute" l'automate `autb` à la fin de l'automate `auta`.
  C'est à dire que l'on crée une nouvelle transition de tous les états finaux de l'automate a vers l'état initial de l'automate b, le nouvel état initial est celui de `auta` et les nouveaux états finaux sont ceux de `autb`.

  `agraphefin auta autb l` retournera l'automate où on "agraphe" l'automate `autb` de manière à ce que l'état final de `auta` se voit "fusionné" avec l'état initial de `autb`.
  C'est à dire que les transitions allant vers un état final de `auta` sont redirigées vers l'état initial de `autb`.

List of Figures