Institut Galilée

INFO 2 :
FOURNIER Stéphane
ROUSSET Yohan

Devoir de Programmation Fonctionnelle

Année 2009-2010

Table des matières

I. Introduction 4
II. Les combinateurs 5
A. Pourquoi ce langage ? 5
B. Comment est constitué un terme combinateur ? 5
C. Les différents combinateurs 6
1. Le combinateur I 6
2. Le combinateur K 6
3. Le combinateur S 6
4. Le combinateur B 6
5. Le combinateur C 6
III. Les types de données 8
A. Le type lambda_terme 8
B. Le type combinateur 8
IV. Les fonctions 10
A. La fonction de compilation 10
B. La fonction d'abstraction 11
C. La fonction d'optimisation 12
D. La fonction d'évaluation 13
E. Les fonctions permettant l'affichage 15
V. Les exemples 18
1. Le test base1 18
2. Le test base2 18
3. Le test base3 19
4. Le test base4 20
5. Le test base5 22
6. Le test base6 23
7. Le test base7 24
8. Le test base8 25

I. Introduction

Dans le cadre de notre formation d'ingénieur, nous sommes amenés à suivre un cours de programmation fonctionnelle. A travers ce cours, nous abordons le λ-calcul avec toutes les notions inhérentes à ce sujet. Nous sommes alors amenés à manipuler des termes afin de les évaluer, les simplifier, les calculer.

En outre, un terme en λ-calcul peut être traduit dans un langage proche utilisant ce qu'on appelle des combinateurs. On entre alors dans la logique combinatoire. Ainsi, il nous a été proposé pour ce devoir de créer un compilateur en CAML qui transforme un terme de λ-calcul en code combinateur. Ce rapport sera alors divisé en plusieurs parties afin de rendre compte de notre travail. Nous verrons tout d'abord les différents combinateurs qui existent et que nous utilisons. Ensuite, nous expliciterons les types de données qui nous permettent de créer nos termes. Puis, nous donnerons le code des fonctions permettant la compilation, ainsi que les explications nécessaires à leur compréhension. Enfin, nous terminerons par des exemples d'utilisations.

II. Les combinateurs

Avant toute chose, il est important de définir pourquoi ce langage a été introduit, puis sa composition. Ensuite nous expliciterons les différents combinateurs existants.

A. Pourquoi ce langage ?

Lorsque l'on cherche à évaluer un λ-terme, il est nécessaire d'utiliser une règle d'évaluation, appelée β-réduction. Ainsi, lorsqu'on évalue un λ-terme, nous avons trois cas :

- Si ce terme est une variable, son évaluation retourne cette variable ;
- Si ce terme est une abstraction, la réduction renvoie l'abstraction dont le corps a été normalisée ➔ soit M un λ-terme : *évaluation*$(\lambda x.M) = \lambda x.(évaluation\ M)$;
- Si ce terme est une application, il faut dans un premier temps évaluer le terme de gauche. Si ce dernier, une fois évalué, commence par une abstraction (typiquement : *λx)*, alors il faut remplacer toutes les occurrences de *x* du terme évalué par la partie droite de l'application. Or cette opération de substitution n'est pas facilement implémentable.

L'opération de substitution nécessite en effet l'implémentation de l'α-renommage, ce qui n'est pas trivial. L'accumulation de ces difficultés nous a ainsi empêchés d'implémenter un évaluateur de λ-termes dans ce devoir dans le temps imparti.

B. Comment est constitué un terme combinateur ?

Ces difficultés identifiées Moses Schönfinkel et Haskell Curry ont eu l'idée de définir une nouvelle notation afin de s'affranchir du problème lié à l'abstraction du λ-calcul : la logique combinatoire. Ainsi, la seule opération disponible est l'application d'un terme sur un autre. Ainsi, en ajoutant des combinateurs (S, K, I, etc.), ayant un comportement applicatif défini, et des variables à l'opération d'application, on peut créer un terme dans le langage combinateur.

Ainsi, voici des exemples de termes combinateurs, avec x y et z des variables :

- (x y)
- (I x)
- (K x) y
- S x y z

Il est à noter que le parenthésage aurait pu être omis ici car il ne présente aucune ambigüité. On rappelle ainsi ici que le parenthésage des termes (que ce soit dans le λ-calcul ou dans le langage combinateur), se fait de gauche à droite. Ainsi :

$$S\ x\ y\ z\ t\ u\ v = (((((S\ x)y)z)t)y)v$$

Présentons maintenant les différents combinateurs présents dans la logique combinatoire ainsi que leur δ-règle d'évaluation associée.

C. Les différents combinateurs

1. Le combinateur I

Le combinateur **I** est le combinateur identité. Il peut s'appliquer à un autre terme combinateur. On a alors : $I\,x = x$.

2. Le combinateur K

Le combinateur **K** (parfois appelé T) est le combinateur d'annulation (aussi appelé absorbeur). Il permet d'absorber un terme. Travaillant sur deux termes, son fonctionnement est le suivant : $K\,x\,y = x$ ➔ y a été absorbé par K.

3. Le combinateur S

Le combinateur S représente l'opération de distribution. Ainsi, il opère sur trois termes combinateurs et les distribue entre eux de la façon suivante : $S\,x\,y\,z = (x\,z)\,(y\,z)$.On a alors le troisième terme qui subit l'application du premier terme, puis du deuxième terme. Le résultat de l'application (x z) s'applique ensuite au résultat de l'application (y z).

4. Le combinateur B

Ce combinateur est aussi appelé combinateur de composition. Il prend trois termes. Il applique le deuxième sur le troisième, puis le premier sur le résultat de la première application. On a alors : $B\,f\,g\,x = f\,(g\,x)$.

5. Le combinateur C

Ce dernier combinateur que nous utiliserons dans notre devoir permet de permuter des termes entre eux. Aussi appelé permutateur, ou combinateur de condition (d'où le C), il représente une sorte de « if ». Son comportement est défini ainsi : $C\,b\,x\,y = b\,x\,y$. En fait, comme on a pu le voir avec le if du λ-calcul en cours, si b est T ou F (vrai ou faux, sachant que T est représentable par K en langage combinateur), alors le C permettra de renvoyer x (si K), ou y (si F, avec $F\,x\,y = y$).

Ce comportement peut se vérifier ainsi :

- Si $b = K$, on a vu que K prend deux termes et renvoie le premier. On a alors $C\,b\,x\,y = K\,x\,y = x$, ce qui représente bien le comportement du if du λ-calcul vu en cours.

- Si $b = F$, on a vu ci-dessus que F prend deux termes et renvoie le second. On a alors $C\ b\ x\ y = F\ x\ y = y$, ce qui représente bien le comportement « else » du « if » vu en cours.

Nous venons d'expliciter le langage combinateur qui sera le langage de sortie de notre compilateur. Nous allons donc maintenant expliquer les types de données CAML qui permettront d'implémenter des λ-termes et des termes combinateurs.

III. Les types de données

Commençons par le type de données présent en entrée de notre compilateur permettant d'implémenter des λ-termes.

A. Le type lambda_terme

Ce type, dont le code sera révélé plus tard, permet d'implémenter les λ-termes. Ainsi, il permet qu'un λ-terme soit constitué soit :

- d'une variable ;
- d'une abstraction, suivie d'un λ-terme;
- d'une application d'un λ-terme sur un λ-terme.

Il découle de cette définition le code CAML suivant :

```
type lambda_terme = V of string
                  | AP of lambda_terme * lambda_terme
                  | AB of string * lambda_terme;;
```

On remarque alors que ce type est récursif. En outre, le choix du type « string » pour définir une variable a été fait dans le but de simplifier le traitement (il est simple de tester si deux chaines de caractères sont identiques lorsque l'on souhaite savoir si une variable est égale à une autre) et de faciliter la lecture (V "x" permet de coder la variable x, ceci est lisible très facilement).

On peut alors représenter des variables à l'aide du préfixe « V », des applications avec le préfixe « AP », et des abstractions avec « AB ». Voici des exemples d'utilisations :

- x ➔ V "x"
- λxy.x ➔ AB("x", AB("y",V "x"))
- λxyz.(x y) z ➔ AB("x",AB("y", AB("z", AP(AP(V "x", V "y"),V "z"))))

Le type permettant d'implémenter les λ-termes étant définis, explicitons le type permettant de coder dans le langage combinateur. Il représente le type de sortie de notre compilateur.

B. Le type combinateur

Ce type doit permettre à l'utilisateur d'implémenter simplement un terme combinateur en CAML. Ainsi, l'utilisateur doit pouvoir :

- utiliser les combinateurs de bases S, K, I, B et C ;
- utiliser des variables ou constantes ;
- appliquer un terme combinateur sur un autre terme combinateur.

On peut alors définir le type combinateur à l'aide du code CAML suivant :

```
type combinateur = S
            |K
            |I
            |B
            |C
            |VS of string
            |APPLIS of combinateur * combinateur;;
```

Comme le type précédent, celui-ci est récursif. Le choix des variables string est motivé par la même raison que précédemment, c'est-à-dire dans un souci de lisibilité et de praticité d'utilisation.

Grâce à ce type de donnée, on peut avec le préfixe VS définir une variable ou une constante, avec APPLIS on peut appliquer un premier combinateur sur un second. On retrouve enfin les combinateurs de base. Leurs comportements respectifs ne seront définis que dans la fonction d'évaluation que nous verrons plus tard dans ce document.

IV. Les fonctions

Maintenant, revenons au but principal de notre devoir : celui de créer un compilateur qui permet de compiler (de transformer) un λ-terme en un terme combinateur. Pour cela, nous avons besoin de plusieurs fonctions annexes à la fonction de compilation. Chacune de ces fonctions verront leur fonctionnement expliqué. Les choix d'implémentations seront par ailleurs explicités au maximum afin de rendre compte du travail fourni.

Pour compiler, il est nécessaire d'avoir trois fonctions de bases. Ces fonctions sont :

- "opt" : la fonction d'optimisation ;
- "a" : la fonction d'abstraction d'une variable dans un terme combinateur ;
- "comp" : la fonction de compilation à proprement parler.

A. La fonction de compilation

La fonction **comp** permet la compilation d'un λ-terme en un terme combinateur. Ainsi, cette fonction prend un argument en entrée qui est le λ-terme à traduire. Elle renvoie alors cette traduction en langage combinateur. Cette fonction récursive peut alors rencontrer très exactement trois cas lors de la traduction :

- Si le λ-terme courant à traduire est une variable, alors il suffit de traduire cette variable directement en langage combinateur avec le préfixe « VS » ;
- Si le λ-terme courant est une application, elle se traduit en langage combinateur en elle-même. Il faut alors propager la compilation sur les parties gauches et droites de l'application afin d'avoir un terme combinateur valide ;
- Si le λ-terme est une abstraction (λx.M par exemple), il faut alors abstraire la variable définie par cette abstraction (ici x) dans le terme compilé (ici M -> comp M).

Le comportement de cette fonction étant maintenant défini, on peut en donner son code CAML :

```
let rec comp e = match e with
      V x -> VS x
      | AP(x,y) -> APPLIS(comp x, comp y)
      | AB(x,y) -> a x (comp y);;
```

Le type de cette fonction est : `lambda_terme -> combinateur = <fun>`

On remarque alors l'utilisation d'une fonction qui prend deux arguments : **a**. Cette fonction est la fonction d'abstraction d'une variable (ici x), dans un code combinateur (ici fourni par la compilation du code de y). Cette fonction sera définie plus tard dans ce document.

En outre, on remarque bien qu'une variable est traduite en elle-même, une application est traduite en une application identique, dont les parties gauches et droites sont les codes compilés des

parties gauches et droites de l'application en code λ-terme. Enfin, on remarque que dans le cas de l'abstraction, on abstrait bien la variable dans le code combinateur qui suit.

B. La fonction d'abstraction

La fonction d'abstraction **a** d'une variable dans un terme combinateur permet l'élimination des abstractions (λx par exemple) d'un λ-terme. Ainsi, la fonction d'abstraction va prendre deux arguments en entrée. Le premier argument est la variable à abstraire. Le second argument représente le code combinateur dans lequel on souhaite abstraire la variable. Ainsi, la fonction va procéder à un « matching » sur le code combinateur. Elle distinguera alors plusieurs cas différents :

- Si le terme combinateur passé en argument est une application ((M N) par exemple), alors on peut l'optimiser à l'aide de la fonction **opt** que nous définirons dans la prochaine sous-partie. Elle prendra en argument un terme combinateur formé ainsi :
  - ((S (a x M)) a x N). On appelle donc récursivement la fonction **a** sur les parties gauche à et droite de l'application, en ajoutant le combinateur S de distribution. L'abstraction de x est ainsi propagée au reste du code combinateur.
- Si le terme combinateur est une variable, alors il se pose à nous deux cas différents :
  - Si la variable en question est la même que la variable à abstraire, l'abstraction s'effectue et la fonction **a** renvoie le combinateur identité I ;
  - Si la variable en question (on note cette variable y) est différente de la variable à abstraire, alors dans ce cas il faudra renvoyer une application formée ainsi : K y ;
- Sinon, si aucun des « patterns » précédent n'a été reconnu, c'est que nous somme en présence d'un combinateur de base (S, K, I, B, C). Ainsi, il n'y a rien de plus à faire car on ne peut pas abstraire une variable dans un combinateur de base. On décide alors de renvoyer un terme de la forme (K, l), avec l le terme dans lequel l'abstraction doit se faire (typiquement un combinateur de base).

Grace à toutes ces définitions, nous pouvons alors définir le code CAML de la fonction d'abstraction. Il est le suivant :

```
let rec a x l = match l with
      APPLIS(f1, f2) -> opt (APPLIS(APPLIS(S, a x  f1), a x f2))
      |VS y -> (if (x = y) then I
                  else APPLIS(K, VS y) )
      | _ -> APPLIS(K, l);;
```

Le type de cette fonction est : `string -> combinateur -> combinateur = <fun>`

Le string représente la variable à abstraire, le premier combinateur représente le code dans lequel la variable doit être abstraire (argument l). La fonction renvoie alors un code combinateur.

On remarque alors que tous les cas traités précédemment sont présent dans le code CAML. Notamment le troisième cas qui, si aucun des « matching » précédent n'a été effectué, c'est qu'il n'y a plus de possibilité d'abstraction, donc on arrête l'abstraction courante et on renvoie le terme l tel quel. Cela se produit si l est un combinateur de base.

C. La fonction d'optimisation

Cette fonction est appelée par la fonction d'abstraction. Elle permet dans un cas bien précis d'optimiser le terme combinateur afin de réduire le nombre d'étapes nécessaires à l'évaluation du terme. En effet, lorsque la fonction abstraction doit abstraire une variable dans un terme combinateur qui est une application de deux termes, alors elle demande une optimisation de cette application. C'est le rôle de la fonction **opt**. Ainsi, cette fonction prend des objets de la forme : (S p) q , avec p et q des termes combinateurs.

Cette fonction va alors procéder à un « matching » sur le terme passé en paramètre. Elle va alors tenter d'appliquer cinq règles d'optimisations qui sont les suivantes :

- Si le terme à optimiser est de la forme (S (K p) (K q) , alors on peut appliquer la distribution S. On peut alors appliquer $S\ x\ y\ z = (x\ z)\ (y\ z)$ avec $x = K\ p$, $y = K\ q$, et pour tout terme z. On a alors la distribution (K p) z => p , (K q) z => q. Or on ne connait pas z au moment de l'appel à la fonction d'optimisation. Donc on rajoute le combinateur K dans le terme à renvoyer afin d'absorber le terme z lors de l'évaluation. **opt** renverra donc dans ce cas là : K($p\ q$) ;
- Si le terme à optimiser est de la forme (S (K p) I , on commence par appliquer la distribution S. on a alors $S\ x\ y\ z = (x\ z)\ (y\ z)$ avec $x = K\ p$, $y = I$, et pour tout terme z. On a alors la distribution (K p)z => p , I z => z . On se retrouve alors avec un terme de la forme : p z . Or le terme z n'est pas connu au moment de l'optimisation (z est en fait situé plus haut dans l'arbre du terme combinateur global issu de la compilation). Donc la fonction **opt** renvoie dans ce cas p.
- Si le terme à optimiser est de la forme (S (K p) q , alors on applique la distribution issue du combinateur S. On peut alors appliquer ce qui suit : $S\ x\ y\ z = (x\ z)\ (y\ z)$ avec $x = K\ p$, $y = q$, et pour tout terme z. On a alors la distribution (K p) z => p et (q z) . Or z étant inconnu à ce moment là, on doit utiliser le combinateur de base B afin de forcer l'application $(q\ z)$ de s'effectuer en priorité, avant de l'utiliser comme argument à p. **opt** doit donc renvoyer $(B\ p)\ q$
- Si le terme à optimiser est de la forme (S p (K q) , alors on applique la distribution issue du combinateur S. On peut alors appliquer ce qui suit :

$S\ x\ y\ z = (x\ z)\ (y\ z)$ avec $x = p$, $y = K\ q$, et pour tout terme z. On a alors la distribution $(\mathrm{p\ z})$ et $(K\ q)\ z => q$. Comme précédemment, z est inconnu au moment de l'évaluation. Or ici, il faut en appliquer en priorité p sur z, avant d'appliquer ce résultat sur q. Il suffit alors d'utiliser le combinateur de base de permutation : C. **opt** doit donc renvoyer $(C\ p)\ q$.

- Enfin, si le terme à optimiser est de la forme $(\mathrm{S\ p\ q})$, c'est qu'il n'y a aucune optimisation applicable. La fonction **opt** se contente alors de renvoyer $(\mathrm{S\ p})\mathrm{q}$.

Nous venons de définir le comportement de la fonction d'optimisation, donnons le code CAML de cette fonction :

```
let opt e = match e with
      APPLIS(APPLIS(S, APPLIS(K,p)),APPLIS(K,q)) -> APPLIS(K,APPLIS(p,q))
      |APPLIS(APPLIS(S, APPLIS(K,p)),I) -> p
      |APPLIS(APPLIS(S, APPLIS(K,p)),q) -> APPLIS(APPLIS(B,p),q)
      |APPLIS(APPLIS(S, p),APPLIS(K,q)) -> APPLIS(APPLIS(C,p),q)
      |APPLIS(APPLIS(S,p),q) -> APPLIS(APPLIS(S,p),q);;
```

Le type CAML de cette fonction d'optimisation est le suivant :

```
combinateur -> combinateur = <fun>
```

Néanmoins, il est à préciser que le filtrage de cette fonction n'est pas exhaustif. En effet, le type combinateur possède d'autres éléments, tels que les combinateurs de bases et les variables. Or il n'est pas ici nécessaire de les inclure dans le filtrage car l'appel à **opt** se fait dans **a**. Or cette dernière met en forme le terme combinateur à optimiser. Elle se charge ainsi de faire en sorte que le terme envoyé à **opt** soit « matchable » par cette dernière.

A ce titre, il est probablement possible à notre sens d'optimiser l'utilisation de la fonction **opt**. En effet, la fonction **a** met en forme un terme à partir d'un combinateur de base S. Or ce combinateur n'est pas déterminant dans le matching. On pourrait alors créer une fonction d'optimisation qui omettrait le combinateur S (au lieu de reconnaitre des termes du type $(\mathrm{S\ p\ q})$, on tenterait une reconnaissance des termes de la forme $(\mathrm{p\ q})$, en supposant que l'appel à **opt** n'est fait qu'à partir de **a**. Cette dernière mettrait en forme un terme combinateur de la forme $(\mathrm{p\ q})$ et non plus $(\mathrm{S\ p\ q})$.

D. La fonction d'évaluation

Nous allons présenter dans cette partie notre fonction d'évaluation de terme combinateur. Celle-ci devra appliquer les comportements définis pour chaque combinateur de base sur le terme donné en paramètre. Ainsi, elle tentera de simplifier au maximum le terme, jusqu'à atteindre sa forme normale.

Cette fonction récursive devra alors opérer une reconnaissance de motif sur le terme reçu en paramètre :

- Si ce terme est un combinateur de base, il est en forme normale et ne peut être réduit d'avantage. La fonction le renvoie tel quel ;
- Si ce terme est une variable, même constat, une variable est normale. On la renvoie tel quel ;
- Si le terme est une application, alors il faut travailler sur les parties gauche et droite de cette application. Soit ces parties *a* et *b*, on procède à une nouvelle reconnaissance de pattern sur la partie gauche (sur a donc) :
  - Si *a* est le combinateur identité, le comportement défini requiert alors de renvoyer b ;
  - Si *a* est une application dont la partie gauche est le combinateur de base K, alors il faut renvoyer la partie droite de *a ;*
  - Si a est une application, donc la partie gauche est une nouvelle application dont la partie gauche est le combinateur de base S, il faut procéder à la distribution. On est donc plus précisément dans le cas où $a = (S\ x)y$. Le combinateur de base B s'applique alors sur x, y et b. La fonction va donc créer le terme $\big((x\ b)(y\ b)\big)$. Puis elle va relancer l'évaluation sur ce dernier récursivement.
  - Si a est une application dont la partie gauche est une autre application dont la nouvelle partie gauche est le combinateur de base B, il faut composer les termes voisins de la bonne manière. On a alors $a = (B\ f)g$. Il faut donc créer l'application composée suivante : $(f\ (g\ b))$. Il suffit ensuite de rappeler la fonction d'évaluation récursivement sur ce terme ;
  - Puis, si a est une application formée comme ci-dessus, sauf qu'au lieu de retrouver un combinateur de base B, c'est un combinateur de base C, alors il faut effectuer la permutation adéquat. Ici, en reprenant les mêmes variables que le cas précédent, on aurait donc à appeler récursivement la fonction d'évaluation sur le terme : $\big((f\ b)g\big)$.
  - Enfin, si aucun des motifs précédent de *a* n'est reconnu, c'est que *a* est en forme normale, on ne peut plus le réduire davantage. On évalue alors la partie droite du terme qui était en entrée de la fonction d'évaluation, c'est-à-dire *b*.

La fonction d'évaluation ainsi définie, il en découle le code CAML suivant :

```
let rec evaluateur t = match t with
                        S | K | I | B | C-> t (* pas d'évaluation possible ici *)
                        | VS x -> t (* idem, une variable est normale *)
                        | APPLIS (a, b) -> match evaluateur a with
    (*cas identité I*)                          I -> evaluateur b
    (*cas K, on absorbe b*)                   | APPLIS (K, z) -> z
    (*cas S, on distribue : (zb)(yb)*)        | APPLIS (APPLIS (S, z), y) -> evaluateur (APPLIS (APPLIS (z, b), APPLIS (y, b)))
    (*cas B, on compose : (f(gb))*)           | APPLIS (APPLIS (B, f), g) -> evaluateur (APPLIS (f, APPLIS (g, b)))
    (*cas C, on permute : ((f,b)g)*)          | APPLIS (APPLIS (C, f), g) -> evaluateur (APPLIS (APPLIS (f, b),g))
    (*sinon, forme normale*)                  | x -> APPLIS (x, evaluateur b);; (*on evalue alors b*)
```

On remarque alors que le type de cette fonction est :

```
combinateur -> combinateur = <fun>
```

Ce qui correspond parfaitement à nos attentes de typage.

E. Les fonctions permettant l'affichage

Nous avons décidé d'implémenter un affichage des arbres produits. En effet, des fonctions ont été définies afin d'afficher des arbres issues de codes combinateurs ou de λ-termes.

Nous retrouvons alors deux principales fonctions qui prennent respectivement un λ-terme en paramètre, et un terme combinateur. Ces fonctions sont `afficheLTerme` et `afficheComb.`

Ces fonctions appellent deux fonctions, respectivement `afficheLTerme1` et `afficheComb1`, avec les bons arguments. Voici donc le code de ces quatre fonctions, ainsi que le code des fonctions auxiliaire nécessaires :

- `hauteurComb` et `hauteurLTerme` qui prennent soit un terme combinateur, soit un λ-terme et rend la hauteur de l'arbre formé par ces termes ;
- `max` qui prend deux entier et rend le plus grand des deux ;
- `afficheString` qui prend une chaine de caractères et deux entiers, et qui affiche cette chaine sur le dessin à la position donnée par les deux entiers ;
- `puissance` qui prend deux entiers et qui renvoie le premier à la puissance du second. Cette fonction est récursive. Typiquement, elle nous servira à calculer des valeurs de puissances de deux.

```
let rec hauteurComb arbre = match arbre with
            S -> 0
            |K -> 0
            |I -> 0
            |B -> 0
            |C -> 0
            |VS _ -> 0
            |APPLIS (a,b) -> 1+ (Max (hauteurComb a)  (hauteurComb b));;
```

```
let rec hauteurLTerme arbre = match arbre with
            V _ -> 0
            |AB(x,y) -> 1+(hauteurLTerme y)
            |AP(a,b) -> 1+ (Max (hauteurLTerme a) (hauteurLTerme b));;
```

```
let Max x y = if (x>y) then x else y;;
```

```
let rec puissance x y = match y with
                  0 -> 1
                  |_ ->x * (puissance x (y-1));;
```

```
let afficheString chaine x y = (moveto x y);(draw_string chaine);;
```

```
let rec afficheLTerme1 arbre func x y =match arbre with
        V z -> func z x (y-10)

        |AB(z,t) -> func (("l")^z) (x) y ; moveto x (y-6) ; lineto (x) (y-(10*(puissance 2 (hauteurLTerme t)+1))-10) ;
        afficheLTerme1 t func (x) (y-(10*(puissance 2 (hauteurLTerme t)+1))-10)

        |AP(a,b) -> (func "@" x y);moveto (x-3) y;
        lineto (x-(10*(puissance 2 (hauteurLTerme a)+1))) (y-(10*(puissance 2 (hauteurLTerme a)+1)));
        moveto (x+6) y;lineto (x+(10*(puissance 2 (hauteurLTerme b)+1))) (y-(10*(puissance 2 (hauteurLTerme b)+1)) +10);
        (afficheLTerme1 a func (x-(10*(puissance 2 (hauteurLTerme a)+1))) (y-(10*(puissance 2 (hauteurLTerme a)+1))));
        (afficheLTerme1 b func (x+(10*(puissance 2 (hauteurLTerme b)+1))) (y-(10*(puissance 2 (hauteurLTerme b)+1))));;
```

```
let afficheLTerme arbre = afficheLTerme1 arbre afficheString 800 800;;
```

```
let rec afficheComb1 arbre func x y =match arbre with
                S -> func "S" x y
                |K -> func "K" x y
                |I -> func "I"  x  y
                |B -> func "B" x y
                |C -> func "C" x y
                |VS v -> func v x y
                |APPLIS(a,b) -> (func "@" x y);moveto (x-3) y;
                lineto (x-(10*(puissance 2 ((hauteurComb a)+1)))) (y-(10*(puissance 2 ((hauteurComb a)+1))));
                moveto (x+6) y;
                lineto (x+(10*(puissance 2 ((hauteurComb b)+1)))) (y-(10*(puissance 2 ((hauteurComb b)+1))));
                 (afficheComb1 a func (x-(10*(puissance 2 ((hauteurComb a)+1)))) (y-(10*(puissance 2 ((hauteurComb a)+1)))));
                 (afficheComb1 b func (x+(10*(puissance 2 ((hauteurComb b)+1)))) (y-(10*(puissance 2 ((hauteurComb b)+1)))));;
```

```
let afficheComb arbre = afficheComb1 arbre afficheString 800 800;;
```

On décide que la racine de l'arbre soit située au point (800,800). La fonction d'affichage à utiliser est la fonction afficheString. Cela peut être utile de préciser la fonction d'affichage au cas où nous décidons de changer le mode de représentation des variables de string vers un autre type. Il suffira alors de modifier simplement la fonction d'affichage et quelques morceaux de codes dans les fonctions d'affichages.

On précise également le comportement des primitives de dessin : lineto et moveto. moveto prend deux arguments (deux entiers représentant un point). Elle sert à déplacer le crayon sur la surface de dessin, crayon levé, vers le point défini en paramètre. Cela est utile pour se placer en un point particulier afin de commencer à dessiner à partir de celui-ci. La primitive lineto se déplace du point courant vers le point donné en paramètre (par deux entiers), crayon baissé. Ainsi, elle trace une droite jusqu'à ce point.

De plus, nous avons besoin de : #open "graphics";; open_graph "";; clear_graph ();; .

Il est cependant à préciser que les fonctions d'affichages sont incompatibles avec le système OCAML fourni sur les ordinateurs de l'Institut Galilée. Ainsi, ces fonctions requièrent la bibliothèque « graphics » disponible dans le système *CAML Light for Windows*.

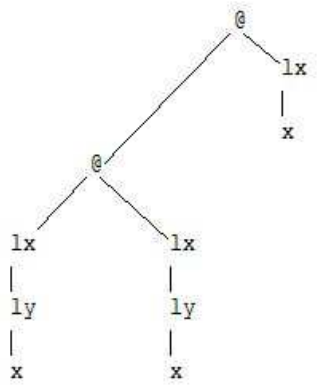

Figure 1 : (T T) I

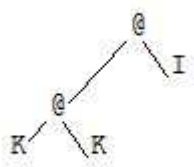

Figure 2 : Compilation de (T T) I

V. Les exemples

Pour prouver la qualité de notre application, nous allons exposer des exemples qui nous semblent pertinent. Nous observerons donc dans un premier temps des exemples de λ-terme que nous compilerons. Ces termes, une fois compilés en langage combinateur, nous les évaluerons à l'aide de notre fonction d'évaluation.

Nous commencerons par compiler des termes simples afin de montrer la bonne marche du compilateur pour les cas de bases. Nous verrons par la suite la compilation de termes plus compliqués. Pour chaque terme compilé, nous l'évaluerons afin de montrer que la compilation s'est bien déroulée.

1. Le test base1

Soit l'exemple suivant : $\lambda xy.(x\ y)$ codé par :

```
let base1 = AB("x",AB("y",AP(V "x", V "y")));;
```

Le code combinateur sera fourni lors de l'appel let cbase1 = `comp base1 ;;`. On a alors le résultat suivant :

```
#base1 : lambda_terme = AB ("x", AB ("y", AP (V "x", V "y")))
#cbase1 : combinateur = I
```

On a également l'arbre suivant généré par `afficheLTerme` sur base1 :

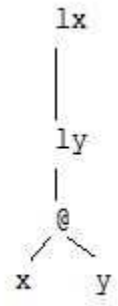

Figure 3 : arbre de base1

L'arbre issu du terme compilé est l'arbre dont le seul élément est une racine valant `I`.

Ici, il est inutile d'évaluer `I`. En effet, ce terme est déjà sous forme normale, l'évaluateur ne pourra rien réduire. En outre, `I` est bien le résultat que l'on désirait avoir.

2. Le test base2

Soit l'exemple suivant : $\lambda xy.(x\ z)$ codé par :

```
let base2 = AB("x",AB("y",AP(V "x", V "z")));;
```

Le code combinateur sera fourni lors de `l'appel let cbase2 = comp base2 ;;`. On a alors le résultat suivant :

#base2 : lambda_terme = AB ("x", AB ("y", AP (V "x", V "z")))
#cbase2 : combinateur = APPLIS (APPLIS (B, K), APPLIS (APPLIS (C, I), VS "z"))

On a également les arbres suivant pour base2 et cbase2:

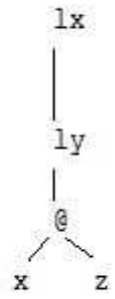

Figure 4 : arbre de base2

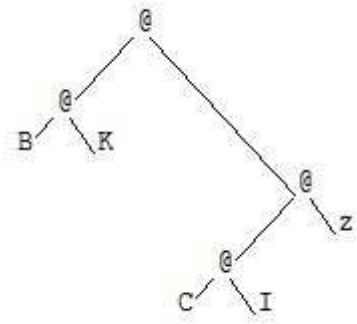

Figure 5 : arbre de cbase2

Lorsqu'on évalue le terme combinateur avec notre évaluateur, le terme renvoyé est le même que le terme en entrée. En fait, le terme en entrée est en forme normale, donc il n'est pas possible de le réduire d'avantage. Cela s'explique par le fait que le combinateur de base B n'a pas assez d'argument lors de son appel. En effet, il lui en faut trois. Or, au moment de son évaluation, il n'a que deux arguments (K et (C I) z), donc l'évaluation de la partie gauche échoue. Ensuite, l'évaluation reprend sur (C I) z. Mais pour les mêmes raisons (C nécessite trois arguments, il n'en a malheureusement que deux), l'évaluation ne peut réduire le sous arbre. C'est pourquoi la fonction renvoie le terme combinateur sans modification.

3. Le test base3

Soit l'exemple suivant : $(\lambda x.(x\ y))z$ codé par :

```
let base3 = AP(AB("x", AP(V "x",V "y")), V "z");;
```

Le code combinateur sera fourni lors de `l'appel let cbase2 = comp base2 ;;`. Il vient alors le résultat suivant :

#base3 : lambda_terme = AP (AB ("x", AP (V "x", V "y")), V "z")
#cbase3 : combinateur = APPLIS (APPLIS (APPLIS (C, I), VS "y"), VS "z")

On a également les arbres suivant pour base3 et cbase3:

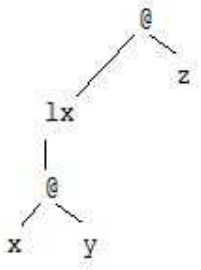

Figure 6 : arbre de base3

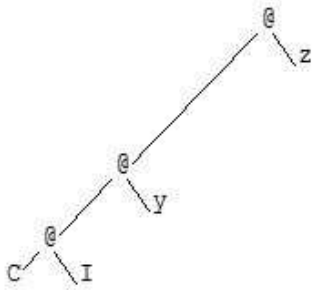

Figure 7 : arbre de cbase3

La compilation est juste. En effet, on distingue bien que le terme λx.x a été traduit en I, et que les deux termes y et z, non lié, sont restés tels quels, et que les applications se sont traduites en elles-mêmes.

En outre, l'évaluation de C provoque dans un premier temps la permutation de y et z. Puis l'évaluateur élimine le terme identité en l'appliquant à z. Il sort donc de l'évaluation (z y). Ce qui se vérifie ici : `combinateur = APPLIS (VS "z", VS "y").`

Nous allons maintenant passer à des cas un peu plus compliqués. Vu en cours en λ-termes, leurs évaluations en λ-calcul pourront être comparées aux évaluations faites par notre évaluateur plus tard dans ce devoir.

4. Le test base4

Avant tous, on pose dorénavant les termes suivant qui nous seront utile à partir de maintenant.

Soit les λ-termes suivant :

- $t = \lambda xy.x$ représente le « vrai » (ou T) ;
- $f = \lambda xy.y$ représente le « faux » (ou F) ;
- $iF = \lambda xyz.(x\ y)\ z$ représente le « ifthenelse », avec y le then, z le else, et x le test ;
- $delta = \lambda x.(x\ x)$;
- $id = \lambda x.x$ représente l'identité.

Maintenant que ces termes sont définis, ils nous serviront à simplifier l'écriture de certain termes plus élaborés que nous verrons à partir de maintenant. Commençons par le terme base4.

Soit $base4 = (f\ t)id$

On a alors en CAML base4 et cbase4 le code compilé de base4 :

```
#base4 : lambda_terme =
 AP
  (AP (AB ("x", AB ("y", V "y")), AB ("x", AB ("y", V "x"))),
   AB ("x", V "x"))
#cbase4: combinateur = APPLIS (APPLIS (APPLIS (K, I), K), I)
```

Ces deux termes sont représentables par les arbres suivants :

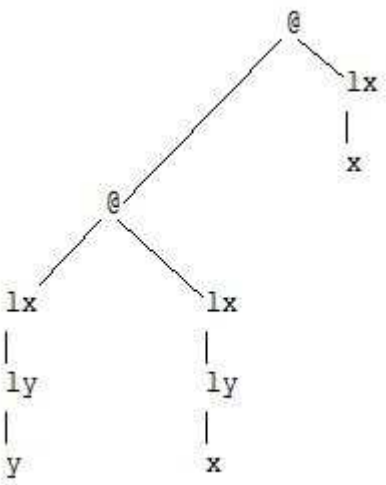

Figure 8 : arbre de base4

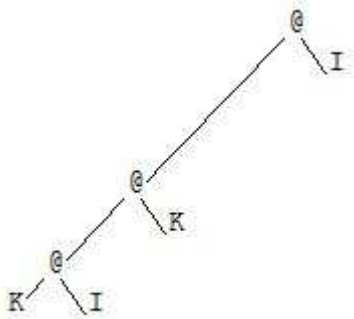

Figure 9 : arbre de cbase4

Lorsqu'on tente l'évaluation du terme cbase4, on sent immédiatement que le terme retourné par l'évaluateur doit être l'identité I. En effet, le terme combinateur K I K doit renvoyer I. Puis, on se retrouve avec une application de la forme (I I). Ceci se réduit alors en terme combinateur I.

Lorsqu'on appel l'évaluateur en lui donnant cbase4 en paramètre, il nous renvoie bien l'identité. L'évaluateur fonctionne donc pour ce cas là.

5. Le test base5

Soit $base5 = (t\ t)id$

On a alors en CAML base5 et cbase5 le code compilé de base5 :

```
#base5 : lambda_terme =
 AP
  (AP (AB ("x", AB ("y", V "x")), AB ("x", AB ("y", V "x"))),
   AB ("x", V "x"))
#cbase5 : combinateur = APPLIS (APPLIS (K, K), I)
```

On peut alors obtenir les deux arbres suivants grâce à nos fonctions d'affichages :

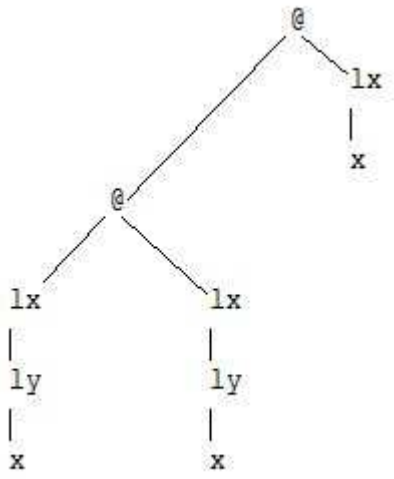

Figure 10 : arbre de base5

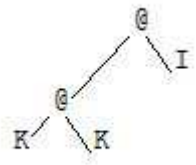

Figure 11 : arbre de cbase5

On remarque alors que l'arbre compilé à la même forme que l'arbre dont il est issu. En outre, on remarque que les deux sous arbres de la partie gauche du λ-terme, représentant le « vrai » en λ-calcul, sont représentés pas des K en langage combinateur. Cela montre la bonne fonctionnalité de notre compilateur, puisque nous avons vu plus tôt dans ce document que le K représentait le « vrai » dans le langage combinateur.

En outre, lorsque l'évaluation de ce terme est tenté à la main, on remarque immédiatement que (K K) I doit renvoyer K lorsqu'on applique le comportement de K (prend deux arguments, renvoie le premier, et absorbe le second).

Lorsqu'on appelle l'évaluateur, ce dernier nous renvoie bien K. C'est bien le comportement que l'on voulait avoir.

6. Le test base6

Soit $base6 = ((if\ t)id)delta$

On pressent alors que ce terme doit valoir, après évaluation, l'identité. En effet, le terme t représentant le « vrai » en prenant deux arguments (ici id et delta), renverra le premier argument (ici id).

On a alors les représentations CAML suivante :

```
#base6 : lambda_terme =
 AP
  (AP
    (AP
      (AB ("x", AB ("y", AB ("z", AP (AP (V "x", V "y"), V "z")))),
       AB ("x", AB ("y", V "x"))),
     AB ("x", V "x")),
   AB ("x", AP (V "x", V "x")))
#cbase6 : combinateur =
 APPLIS (APPLIS (APPLIS (I, K), I), APPLIS (APPLIS (S, I), I))
```

On a alors le terme combinateur compilé à partir de base6 suivant :

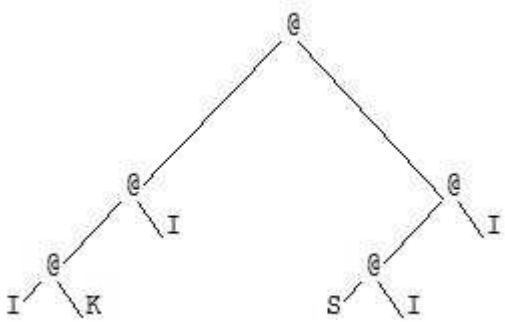

Figure 12 : arbre de cbase6

On cherche ensuite à évaluer ce terme. L'évaluation commence par le sous terme (I K). Ce dernier est évalué en K. Ensuite, nous avons ((K I)((S I) I)). Or nous rappelons que le combinateur de base K prend deux arguments. Il absorbe ici (S I) I. Il renvoie alors I. C'est par ailleurs ce qu'on

cherchait à avoir lorsqu'on évalue le λ-terme (((iF t) id) delta). Le if, si t est le « vrai », alors il renvoie le « then ». Ici le « then » est l'identité. C'est bien ce que l'évaluation du terme combinateur montre.

L'évaluateur confirme le résultat calculé à la main en renvoyant :

```
ecbase6 : combinateur = I
```

Ainsi, cela montre que la compilation a conservé le code représentant le if. Cela montre également que notre évaluateur fonctionne dans ce cas la.

7. Le test base7

Soit $base7 = ((if\ f)id)delta$

On pressent alors que ce terme doit valoir, après évaluation, le terme delta. En effet, le terme f représentant le « faux » en prenant deux arguments (ici id et delta), renverra le second argument (ici delta).

On a alors les représentations CAML suivante :

```
#base7 : lambda_terme =
 AP
  (AP
    (AP
      (AB ("x", AB ("y", AB ("z", AP (AP (V "x", V "y"), V "z")))),
       AB ("x", AB ("y", V "y"))),
     AB ("x", V "x")),
   AB ("x", AP (V "x", V "x")))
#cbase7 : #- : combinateur =
 APPLIS (APPLIS (APPLIS (I, APPLIS (K, I)), I), APPLIS (APPLIS (S, I), I))
```

On a alors le terme combinateur compilé à partir de base7 suivant :

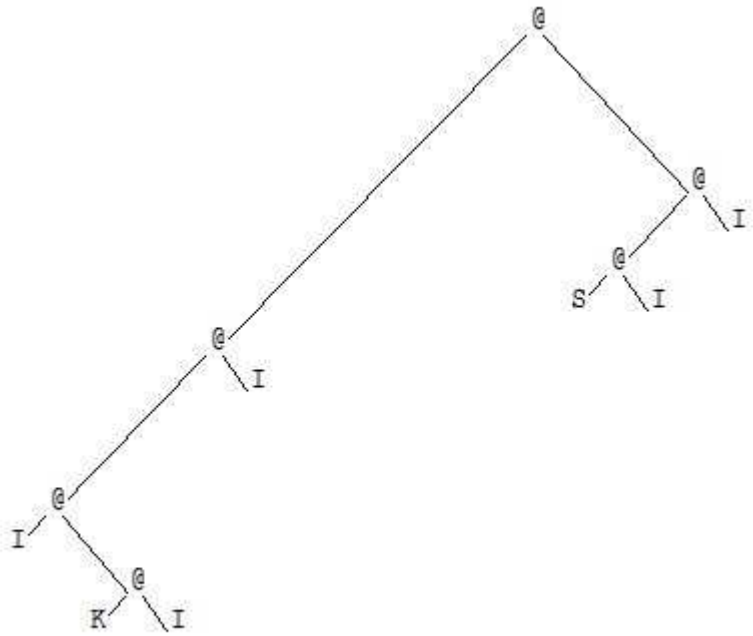

Figure 13 : arbre de cbase7

On essaye alors d'évaluer ce terme. On rappelle que notre pressentiment nous donnait comme résultat de l'évaluation delta. Or la compilation de delta est (S I) I d'après notre compilateur. Ici, à l'aide de l'arbre, on peut tenter dans un premier temps une évaluation à la main. Tout d'abord, le terme (K I) du sous arbre gauche va chercher son deuxième argument I. on se retrouve alors une application (I I). Ceci se simplifie en I. Puis, I s'applique à (S I) I, ce qui donne (S I) I. C'est notre delta. Le comportement de l'ifthenelse est alors conservé. C'est bien ce qu'on voulait avoir.

Appelons maintenant notre évaluateur sur ce terme afin de vérifier que notre pressentiment était le bon, et surtout que l'évaluation faite à la main précédemment est bonne. Notre évaluateur nous fournit : `combinateur = APPLIS (APPLIS (S, I), I).` Notre évaluateur fonctionne donc sur ce cas là.

8. Le test base8

Soit $base8 = \left(\left(\left(\lambda x.\left(\lambda w.\left(x\ y\right)\right)\right)z\right)t\right)$

On code alors ce terme dans le type lambda_terme construit, ce qui nous donne :

```
#base8 : lambda_terme =
 AP (AP (AB ("x", AB ("w", AP (V "x", V "y"))), V "z"), V "t")
```

Puis, une fois compilé, nous avons cbase8 tel que :

```
#cbase8 : combinateur =
 APPLIS
  (APPLIS (APPLIS (APPLIS (B, K), APPLIS (APPLIS (C, I), VS "y")), VS "z"),
   VS "t")
```

Ce qui provoque la création des deux arbres suivants :

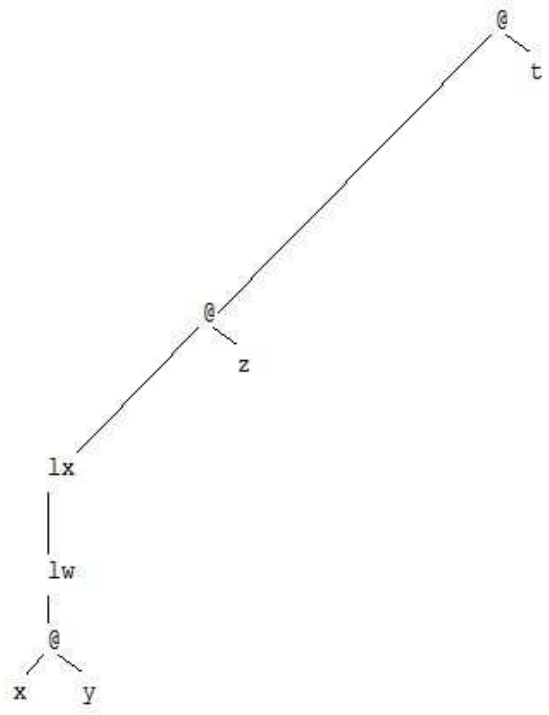

Figure 14 : arbre de base8

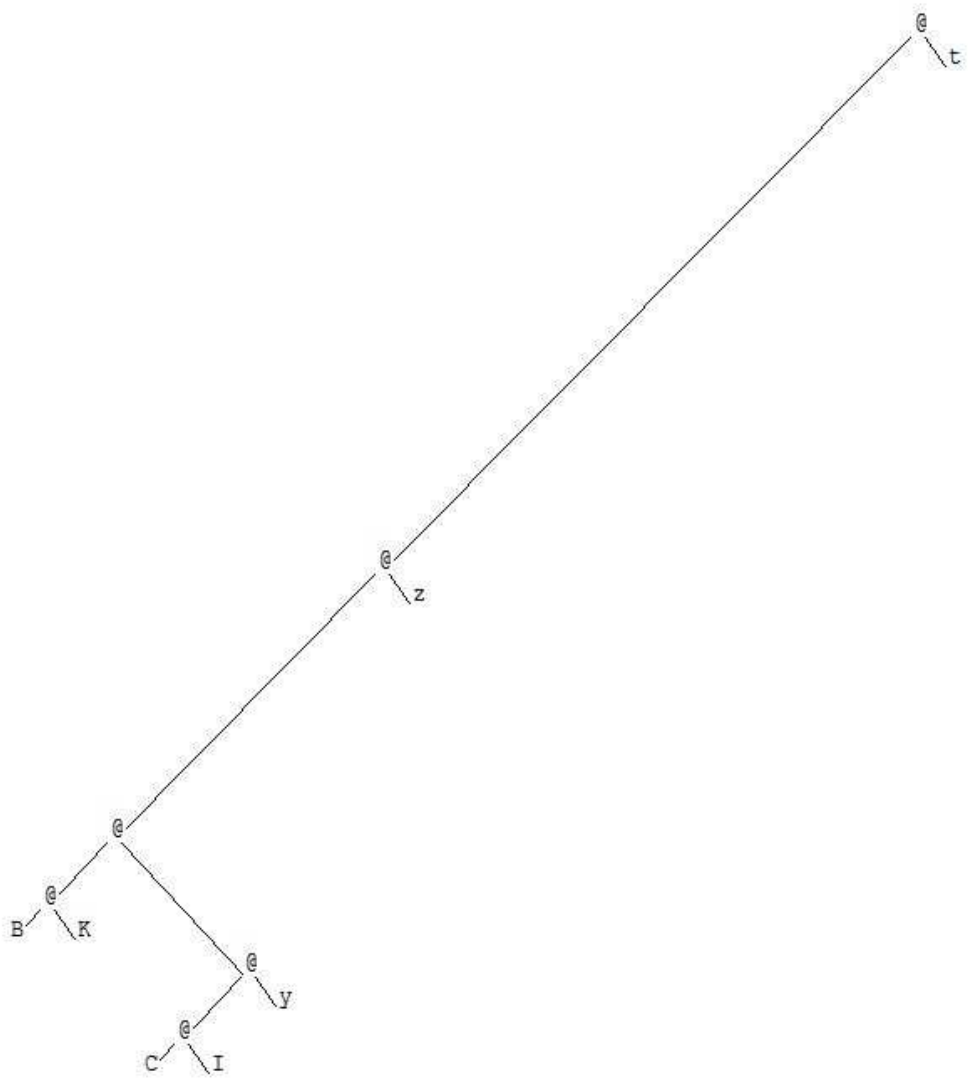

Figure 15 : arbre de cbase8

L'évaluation de base8 se fait dans l'ordre suivant. Dans un premier temps, il faut abstraire x dans le sous arbre composé de λw.(x y). Ainsi, on remplace les occurrences de x par z. On a alors λw.(z y). Puis, on abstrait le w, or il n'y a aucune occurrence de w. Donc le t est éliminé. Il ne reste que l'application (z y).

On évalue alors à la main cbase8 afin de bien voir qu'on a la même chose que l'évaluation de base8. On commence par évalué le B K (C I y) z. On applique la composition, on a alors K (C I y z). Puis on récupère la variable t. On a alors K (C I y z) t, le K absorbe le t, et retourne donc (C I y z). Le C de permutation s'applique, on a alors (I z y). L'identité s'effectue. On a alors (z y). Ce qui correspond bien à l'évaluation base8 compilé.

Notre évaluateur nous renvoie, quant à lui : combinateur = APPLIS (VS "z", VS "y").

C'est bien ce que nous cherchions, donc l'évaluateur est à nouveau fonctionnel sur ce cas là.

Le jeu de test est maintenant terminé, et a montré que notre compilateur donnait les bons résultats, puisque la compilation, suivi de l'évaluation, a permis d'avoir des résultats cohérents avec nos attentes. En outre, notre évaluateur est également bon, puisque pour chaque exemple, il s'accordait avec nos calculs faits à la main.