1.) Types de données.

En Caml, il existe des types constants prédéfinis comme `bool`, `int`, `float`, `string`... Cependant, l'utilisateur peut également créer ses propres types.

Fondamentalement, on peut définir un type de deux façons :

- En faisant une union disjointe de types déjà connus. C'est ce qu'on appelle un *type somme*.
- En faisant un produit cartésien de types déjà connus. C'est un *type produit*.

1.1.) Types « somme ».

Les définitions de type se font à l'aide du mot-clé **`type`**. L'identificateur de l'union disjointe dans les types sommes est la barre verticale « **|** ».

```
#type Booleen = Vrai | Faux;;
Le type Booleen est défini.
```

On pourrait utiliser le type « `Booleen` » ainsi :

```
#let f p = if p=Vrai then "Vrai" else "Faux";;
f : Booleen -> string = <fun>
#f Vrai;;
- : string = "Vrai"
#f Faux;;
- : string = "Faux"
```

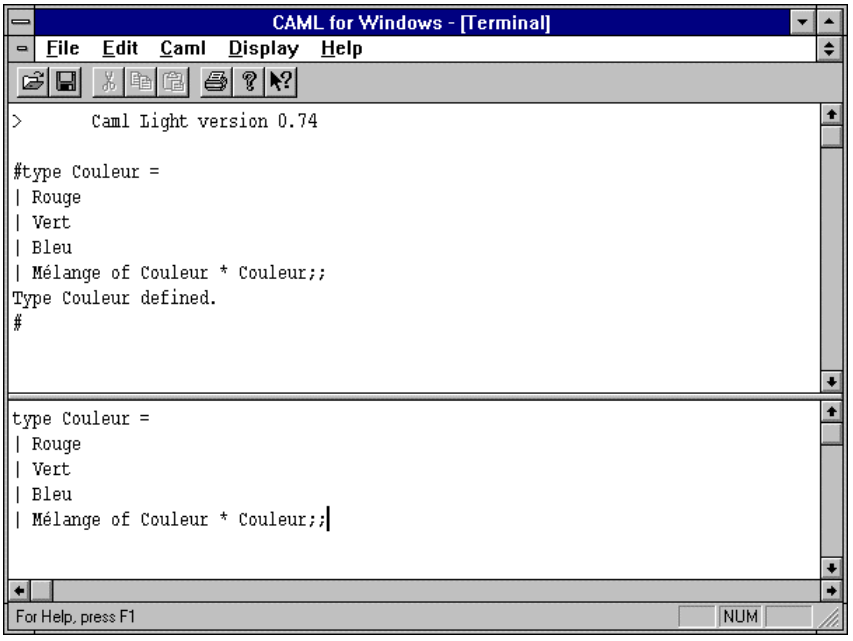


Figure 1 Définition de type somme.

```
#type Sexe = Masculin | Féminin | Indéterminé;;
Le type Sexe est défini.
```

Le type « `Sexe` » pourra s'utiliser ainsi :

```
#let poids_ideal = fun
| taille age Féminin -> ((taille/2 - 30) * (180+age)) / 200
| taille age Masculin -> ((3*taille-250) * (age+270)) / 1200
| _ _ _ -> failwith "Calcul impossible";;
poids_ideal : int -> int -> Sexe -> int = <fun>
#poids_ideal 175 24 Féminin;;
- : int = 58
#poids_ideal 180 30 Masculin;;
- : int = 72
```

Les types somme servent à modéliser des données comprenant des alternatives. On les appelle aussi types « ou » car une donnée modélisée avec un type somme est d'une espèce ou d'une autre ou...

1.2.) Types à constructeurs non constants.

Il s'agit d'un type qui utilise un constructeur ayant un argument. Par exemple, on pourrait définir le type Couleur_de_fond ainsi :

```
#type Couleur_de_fond =
| Rouge
| Vert
| Bleur
| numero of int;;
Type Couleur_de_fond defined.
```

Ceci reviendrait à dire qu'une couleur de fond peut être soit le Rouge, soit le Vert soit le Bleu ou bien une couleur numérotée.

1.3.) Types récursifs.

Il est également possible d'indiquer qu'une couleur de fond peut être constituée du mélange de trois couleurs. On introduira alors un nouveau constructeur avec pour argument un couple de couleurs :

```
#type Couleur_de_fond =
| Rouge
| Vert
| Bleur
| Numero of int
| Mélange of Couleur_de_fond * Couleur_de_fond;;
Type Couleur_de_fond defined.
```

De la même manière, on pourrait définir le type RGB[1] ainsi :

```
#type RGB =
| Rouge of int
| Vert of int
| Bleu of int
| Mélange of RGB * RGB * RGB;;
Type RGB defined.
#let ma_couleur = Bleu 255;;
ma_couleur : RGB = Bleu 255
#let b = Vert 30;;
b : RGB = Vert 30
#let gris =Mélange(Rouge 128, Vert 128, Bleu 128);;
gris : RGB = Mélange (Rouge 128, Vert 128, Bleu 128)
#let blanc =Mélange(Rouge 255, Vert 255, Bleu 255);;
blanc : RGB = Mélange (Rouge 255, Vert 255, Bleu 255)
#let noir =Mélange(Rouge 0, Vert 0, Bleu 0);;
noir : RGB = Mélange (Rouge 0, Vert 0, Bleu 0)
```

Pour convertir un RGB en un entier on pourrait utiliser la fonction :

```
#let convert = function
| Rouge(n) -> n
| Vert(n) -> n *256
| Bleu(n) -> n*256*256
| Mélange (Rouge a, Vert b, Bleu c) -> a+(b*256)+c*256*256
| _ -> failwith "Erreur";;
convert : RGB -> int = <fun>
#convert noir;;
- : int = 0
#convert blanc;;
- : int = 16777215
#convert gris;;
- : int = 8421504
```

1.4.) Exemples d'utilisation des types somme.

1.4.1.) JEU DE CARTES.

On pourrait modéliser un jeu de carte en utilisant les types somme. Les couleurs forment un type énuméré :

```
#type couleur = Carreau | Coeur | Pique | Trèfle;;
Type couleur defined.
```

[1] Red Green Blue : Mode de composition des couleurs, basé sur la synthèse additive. La composition des trois couleurs rouge, vert et bleu donent le blanc, l'absence des trois forment le noir.

Et les cartes un type somme à plusieurs possibilités :

```
type carte =
| As of couleur
| Roi of couleur
| Dame of couleur
| Valet of couleur
| Autre of int * couleur;;
Type carte defined.
```

Pour illustrer le filtrage sur les types sommes, voici une fonction qui retourne la valeur d'une carte dans le jeu de la « belote ». Cette valeur dépend d'une couleur particulière, l'atout, choisie par les joueurs à chaque tour. Les cartes dont la valeur change sont le valet et le neuf. Le neuf compte d'ordinaire pour 0 mais vaut 14 quand il est de la couleur de l'atout. Le valet d'atout vaut 20 au lieu de 2 d'ordinaire. D'autre part, les 10 valent 10 points et les autres petites cartes 0.

```
#let valeur atout = function
| As   _ -> 11
| Roi  _ -> 14
| Dame _ -> 3
| Valet c -> if c=atout then 20 else 2
| Autre (10,_) -> 10
| Autre (9,c) -> if c = atout then 14 else 0
| _ -> 0;;
valeur : couleur -> carte -> int = <fun>
```

1.4.2.) TYPE « NOMBRE »,

On pourrait également définir un type « `Nombre` » qui inclurait à la fois les entiers et les réels :

```
#type Nombre =
| Entier of int
| Réel of float;;
Le type Nombre est défini.
```

Avec ce type « `Nombre` », on peut facilement redéfinir l'addition :

```
#let add = fun
| (Entier a) (Entier b) -> Entier (a + b)
| (Entier a) (Réel b) -> Réel (float_of_int(a) +. b)
| (Réel a) (Entier b) -> Réel (a +. float_of_int(b))
| (Réel a) (Réel b) -> Réel(a +. b);;
add : Nombre -> Nombre -> Nombre = <fun>
##infix "add";;
let a = Réel(3.);;
a : Nombre = Réel 3.0
#let b = Entier(5);;
b : Nombre = Entier 5
#a add b;;
- : Nombre = Réel 8.0
```

1.5.) Types produit.

Les types produits sont des enregistrements dont chaque composante est nommée (on parle d'*étiquettes*). La syntaxe est la suivante :

```
type nom_du_type = {Etiquette_1 : type_de_l_etiquette_1;
                    Etiquette_2 : type_de_l_etiquette_2;
                    ...
                    Etiquette_n : type_de_l_etiquette_n};;
```

On pourrait par exemple, définir ainsi les nombres complexes :

```
#type complexe =
  {Partie_réelle : float ; Partie_imaginaire : float};;
Le type complexe est défini.
#let un_nombre_complexe =
{Partie_réelle = 0. ; Partie_imaginaire = 1.0};;
un_nombre_complexe : complexe =
 {Partie_réelle = 0.0; Partie_imaginaire = 1.0}
```

1.6.) Gestion des événements en Caml.

En Caml, on gère les événements du clavier et de la souris avec un type produit et un type somme :

```
type status =
  { mouse_x : int;          (* Coordonnée X de la souris *)
    mouse_y : int;          (* Coordonnée Y de la souris *)
    button : bool;          (* true si un bouton de la souris est enfoncé *)
    keypressed : bool;      (* true si une touche du clavier est enfoncée *)
    key : char }            (* Le caractère de la touche enfoncée *)

type event =
  | Button_down             (* Un bouton de la souris est enfoncé *)
  | Button_up               (* Un bouton de la souris est relâché *)
  | Key_pressed             (* Une touche est enfoncée *)
  | Mouse_motion            (* La souris est déplacée *)
  | Poll                    (* Aucun événement *)
```

Pour spécifier les événements à attendre :

```
value wait_next_event : event list -> status
```

2.) Structures arborescentes.

Un arbre généalogique :

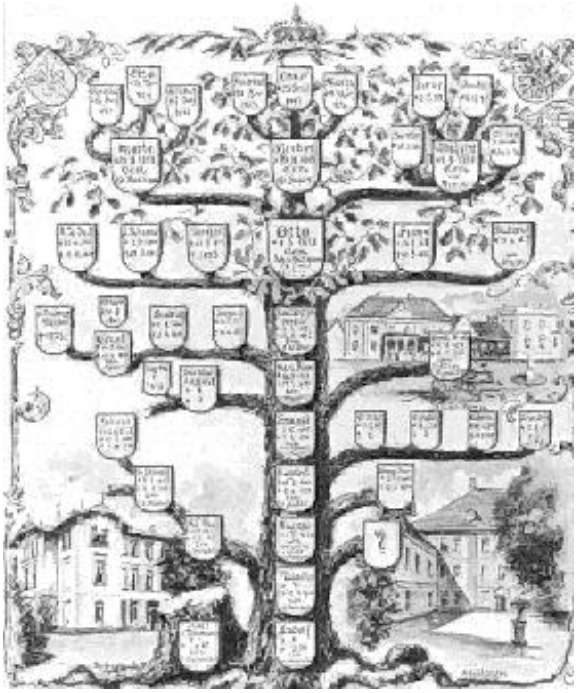


Figure 2 : Arbre généalogique de Bismarck

Un arbre est une structure formée de *nœuds* et de *liens.*

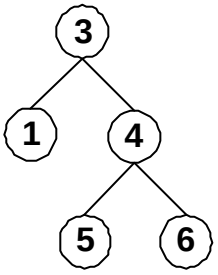


Figure 3 : Arbre.

2.1.) Vocabulaire.

Arbre enraciné Un arbre enraciné est un arbre dans lequel un des sommets se distingue des autres. Ce sommet - souvent appelé noeud - s'appelle la racine de l'arbre. Dans l'exemple de la Figure 3, *3* est la racine de l'arbre.

Ancêtre Un noeud quelconque *y* sur le chemin unique allant de *r* à *x* est appellé ancêtre de *x*. Dans l'exemple de la Figure 3, *4* est l'ancêtre de *5*.

Descendant. Si *y* est un ancêtre de *x*, alors *x* est un descendant de *y*. Dans l'exemple de la Figure 3, *5* est un descendant de *4*.

Sous-arbre Le sous-arbre de racine x est l'arbre composé des descendants de x, enraciné en x. Dans l'exemple de la Figure 3, on peut construire un sous--arbre enraciné en *4*.

Père	Si le dernier arc sur le chemin de la racine *r* d'un arbre *T* vers un noeud x est *(y,x)* alors *y* est le père de *x*. Dans l'exemple de la Figure 3, *4* est le père de *5*.
Fils	Si *y* est le père de *x* alors *x* est le fils de *y*. Dans l'exemple de la Figure 3, *5* est le fils de *4*.
Degré	Le nombre de fils d'un noeud *x* dans un arbre T est appelé le degré de *x*.
Profondeur	La longueur du chemin entre la racine r et un noeud y est la profondeur de x dans T.
Hauteur	La plus grande profondeur que puisse avoir un noeud quelconque de T est la hauteur de T.

2.2.) Arbres binaires.

2.2.1.) DEFINITION DU TYPE.

On pourrait définir le type `arbre_binaire` ainsi :

```
#type arbre_binaire =
| Feuille of int
| Noeud of int * arbre_binaire * arbre_binaire ;;
Le type arbre_binaire est défini.
```

On pourrait également définir une version polymorphe du type `arbre_binaire` :

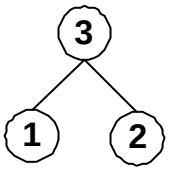
```
#type ('f,'n) arbre_binaire =
| Feuille of 'f
| Noeud of 'n * ('f,'n)arbre_binaire * ('f,'n)arbre_binaire ;;
Le type arbre_binaire est défini.
```

On pourra définir un arbre ainsi :

```
#let arbre = Feuille 3;;
arbre : tarbre = Feuille 3
```

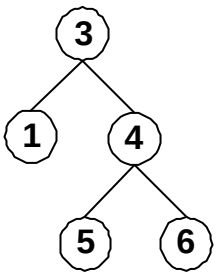
ou bien

```
#let arbre1 = Noeud(3,Feuille 1,Feuille 2);;
arbre1 : tarbre = Noeud (3, Feuille 1, Feuille 2)
```



ou encore :

```
#let arbre2 = Noeud(3,Feuille 1,Noeud(4,Feuille
5,Feuille 6));;
arbre2 : tarbre = Noeud (3, Feuille 1, Noeud (4,
Feuille 5, Feuille 6))
```



Un arbre binaire T est donc une structure récursive qui :

- Ne contient aucun nœud ou
- est formée de trois ensemble de nœuds disjoints : un nœud racine, un arbre binaire appelé **sous-arbre gauche** et un arbre binaire appelé **sous-arbre droit**.

L'arbre binaire qui ne contient aucun nœud est appelé **arbre vide** , parfois noté *NIL*.

2.2.2.) FONCTIONS DE MANIPULATION.

Nombre de feuilles :

```
#let rec nombre_feuille = function
| Feuille (_) -> 1
| Noeud (_,g,d) -> nombre_feuille g + nombre_feuille d;;
nombre_feuille : ('a, 'b) arbre_binaire -> int = <fun>
#nombre_feuille arbre2;;
- : int = 3
```

Nombres de nœuds :

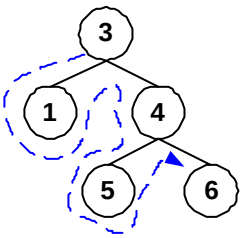
```
#let rec nombre_noeuds = function
| Feuille (_) -> 0
| Noeud (_,g,d) -> 1 + nombre_noeuds g + nombre_noeuds d;;
nombre_noeuds : ('a, 'b) arbre_binaire -> int = <fun>
#nombre_noeuds arbre2;;
- : int = 2
```

Profondeur de l'arbre :

```
#let max a b = if a > b then a else b;;
max : 'a -> 'a -> 'a = <fun>
#let rec profondeur = function
| Feuille (_) -> 0
| Noeud (_,g,d) -> 1 + max (profondeur g) (profondeur d);;
profondeur : ('a, 'b) arbre_binaire -> int = <fun>
#profondeur arbre2;;
- : int = 2
```

Parcours de l'arbre en profondeur:

On part du nœud 3 puis on explore le nœud 1 puis le nœud 4, puis le nœud 5 puis le nœud 6.




```
#let rec parcours = function
|Feuille n -> print_string " F : ";print_int n;print_string " ";
|Noeud (p, g, d)  -> print_string " N : ";print_int p;print_string "
";parcours g;parcours d;;
parcours : (int, int) arbre_binaire -> unit = <fun>
#parcours arbre2;;
```

```
 N : 3  F : 1  N : 4  F : 5  F : 6 - : unit = ()

#let rec parcours = function
|Feuille _ -> ()
|Noeud (_, g, d)  -> parcours g;parcours d;;
parcours : ('a, 'b) arbre_binaire -> unit = <fun>
```

Parcours de l'arbre en largeur :

```
#let parcours_largeur arbre =
let file = ref [arbre] and a = ref arbre in
while list_length(!file) > 0 do
     a := hd(!file);
     file:=tl(!file);
     match !a with
     | Feuille x -> print_string " F : ";print_int x;
     | Noeud (x,g,d) -> begin
                        print_string " N : ";
                        print_int x;
                        file:=!file@[g;d];
                      end;
done;;
parcours_largeur : (int, int) arbre_binaire -> unit = <fun>
#parcours_largeur arbre2;;
N : 3 F : 1 N : 4 F : 5 F : 6- : unit = ()
```

3.) Algorithmes de compression élémentaires[2].

3.1.) Codage des répétitions ou Run Length Coding.

Certains fichiers textes ou certaines images présentent des successions d'octets identiques. Une méthode simple pour réduire la taille des données est de choisir un caractère de contrôle (par exemple #) et de coder les plages de *k* octets identiques par

octet k

Par exemple, la chaîne de caractères `01abbbbbbcZeed` sera codée par `01a#b6Zeed`. Le codage se simplifie si l'on connaît à l'avance un caractère dominant dans le fichier. Par exemple, si l'on sait que « `b` » est l'élément dominant, la chaîne précédente devient :

`01a#6Zeed`

Une séquence de k octets identiques O est remplacée par le triplet d'octets (#, O, k) si k est inférieur à 256. Ce codage ne réduit la taille du fichier qu'à partir de 4 répétitions (ou 3 si on connaît le caractère répétitif). Il n'est réalisable que lorsqu'on peut trouver un octet # non utilisé dans le fichier. Pour des fichiers exécutables, qui utilisent en général les 256 octets, on ne pourra pas appliquer simplement cette méthode. Pour contre, pour des fichiers contenant beaucoup de caractères espace, cette méthode de compression peut se révéler intéressante.

3.2.) Codage topologique.

Il est appliqué sur des fichiers où un octet O est sensiblement dominant. Le principe est le suivant : on lit les données par blocs de 8 octets et on utilise un octet de description, appelé topologique, dans lequel les bits de valeur 1 désignent les positions de l'octet O dans le bloc. Par exemple, le bloc AABDEACA appartient à un fichier dans lequel l'octet dominant est A. Ce bloc sera codé ainsi :

`(11000101)BDEC`

où `(11000101)` est l'octet descriptif.

Ce codage amène une réduction de taille dès que la probabilité p de l'octet O dans le fichier est supérieure à 1/8. On peut itérer cet algorithme en choisissant les autres octets dominants de probabilité supérieure à 1/8 mais son efficacité décroît assez vite.

3.3.) Compression de bas niveau.

3.3.1.) PRINCIPE DE LA METHODE.

Tout fichier peut être décrit comme une succession de bits 0 et 1. On s'intéresse aux différentes suites de bits 0.

Par exemple, dans la suite de bits S = `0001000010110001100001000100100001001` on a les suites de bits `0` suivantes : `0`, `00`, `000` et `0000`. Ces suites ont pour longueurs respectives 1,2,3,4 et comme fréquence d'apparitions 2, 2, 2 et 1.

[2] D'après « Compression et cryptage en informatique » Xavier Marsault, Editions Hermès 1992.

	Suite de bits	Longueur t	fréquence f
s_1	0	1	1
s_2	00	2	2
s_3	000	3	4
s_4	0000	4	3

Le principe de cette méthode de compactage est de remplacer certaines suites de bits s_i par d'autres s_j afin de réduire la taille de S.

3.3.2.) CONDITION DE COMPACTABILITE.

Soit t_i la longueur de la suite s_i et f_i sa fréquence d'apparition.

La suite s est compactable s'il existe deux suites de bits si et sj telles que

$$(t_i * f_j) + (t_j * f_i) < (t_i * f_i) + (t_j * f_j)$$

Dans l'exemple précédent (S = 000100001011000111000010001001000100010000) peut remplacer s_3 par s_1.

On a en effet :

$$(t_1 * f_3) + (t_3 * f_1) < (t_1 * f_1) + (t_3 * f_3)$$
$$(1*4) + (3*1) < (1*1) + (3*4)$$
$$7 < 13$$

On pourrait donc recoder S ainsi :

0100001000110110000101001010010000

3.3.3.) CODAGE PYRAMIDAL.

S est compactable car il existe plusieurs couples (s_i,s_j) vérifiant la propriété :

$$(t_i * f_j) + (t_j * f_i) < (t_i * f_i) + (t_j * f_j)$$

Il suffit ensuite d'ordonner les fréquences f_i par ordre décroissant et de leur attribuer le plus petit motif binaire s_i. Cela s'appelle le *codage pyramidal* :

```
f3 = 4        -->   s1 = 0
f4 = 3        -->   s2 = 00
f2 = 2        -->   s3 = 000
f1 = 1        -->   s4 = 0000
```

La suite initiale S = 000100001011000111000010001001000100010000 comprend 5 octets soit 40 bits.

Après transformation, on obtient S' = 0100100001101110010100010100010000. Cette nouvelle chaîne comprend 31 bits et peut se coder sur 4 octets. On a donc une compression de l'ordre de 30 %.

Bien entendu, pour reconstruire le fichier initial à partir de sa forme compactée, il est nécessaire de savoir comment a été fait le codage pyramidal. Il faut donc sauvegarder en début de fichier la table de substitution des s_i.

3.3.4.) RESULTATS.

Quelques essais ont été réalisés pour tester l'efficacité de l'algorithme. Il est capable de donner de bons résultats sur certains fichiers (taux de compression > 30 %) mais il se comporte de manière atypique (par comparaison avec d'autres algorithmes de compression) du fait que des fichiers du même type peuvent avoir des taux de compression totalement différents.

Fichier	Taille (octets)	Compacté (octets)	Gain (%)
bc.ico	766	583	23.89
bccx.exe	502432	482089	4.00
bccx.exe	32850	26847	18.27
bccx.ovy	502676	481783	4.15
bcx.exe	33918	28074	14.22
cpp.exe	117332	101625	13.38
dist2.exe	54338	28247	47.68
emstest.com	19664	19577	0.40
grep.com	7023	6551	6.72
grep2msg.exe	7316	6614	9.59
impl2msg.exe	7020	6299	10.27
make.exe	41174	39600	3.82
prj2mak.exe	33076	30399	8.09
rc.exe	57805	53410	7.60
tasm.tah	166816	164273	1.52
tasmx.exe	109234	106913	2.12
tcconfig.tc	1820	1534	15.71
tcdef.dpr	4171	1133	72.83
tempc.exe	47968	32814	31.60
tf386.exe	24614	21058	14.44
thelp.com	10448	9823	5.98
tkernel.exe	104771	78148	25.40
tkinst.exe	33962	30163	11.18
tprof.exe	331104	311347	5.96
tlink.exe	72585	69858	3.75
tlinkx.exe	31568	25680	18.65
touch.com	5124	4894	4.48
trancopy.exe	17620	16918	3.98
trigraph.exe	12150	11463	5.65
Total :	**2391345**	**2197897**	**8.08**

4.) Codage de Huffmann.

On cherche à compresser la phrase suivante :

Beauté, mon beau souci[3]

Pour faciliter la démonstration, on ne tiendra pas compte des majuscules, caractères accentués et caractères de ponctuation. On codera donc :

`beautemonbeausouci`

Codé en Ascii, on aurait :

```
 98 101 97 117 116 101 109 111 110 98 101 97 117 115 111 117 99 105
```

soit en binaire :

```
01100010 01100101 01100001 01110101 01110100 01100101 01101101
01101111 01101110 01100010 01100101 01100001 01110101 01110011
01101111 01110101 01100011 01101001
```

Le but de cet algorithme est de trouver un codage moins onéreux en taille tout en restant non ambiguë.

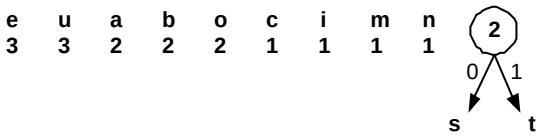
Dans un premier temps, il faut recenser les caractères utilisés et leur nombre d'apparitions :

Caractères	**a**	**b**	**c**	**e**	**i**	**m**	**n**	**o**	**s**	**t**	**u**
Nombre d'apparitions	**2**	**2**	**1**	**3**	**1**	**1**	**1**	**2**	**1**	**1**	**3**

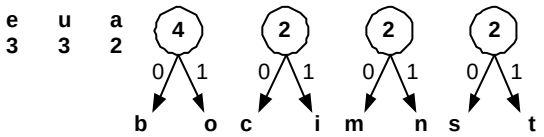
Puis les trier par ordre croissant :

e	**u**	**a**	**b**	**o**	**c**	**i**	**m**	**n**	**s**	**t**
3	**3**	**2**	**2**	**2**	**1**	**1**	**1**	**1**	**1**	**1**

On regroupe ensuite les caractères les moins fréquemment utilisés dans un arbre binaire auquel on attribue un poids. Ce poids sera égal à somme des fréquences d'apparition des deux caractères. Ici le poids sera égal à 2.

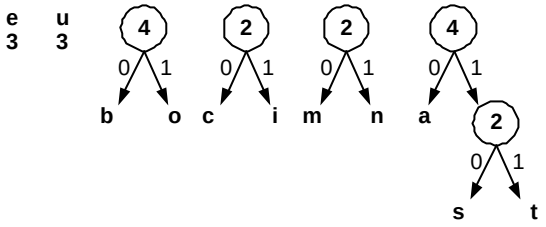


On procède de même pour les autres caractères qui ont une fréquence inférieure ou égale à 2.

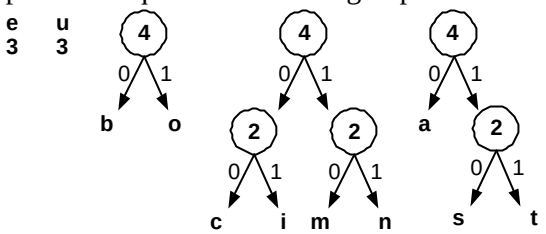


On applique toujours le même algorithme, mais cette fois on regroupe les poids 2 ce qui nous amène à regrouper un caractère et un arbre :

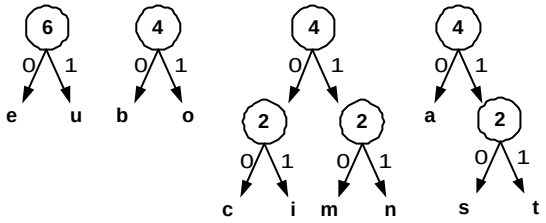
[3] Malherbe.



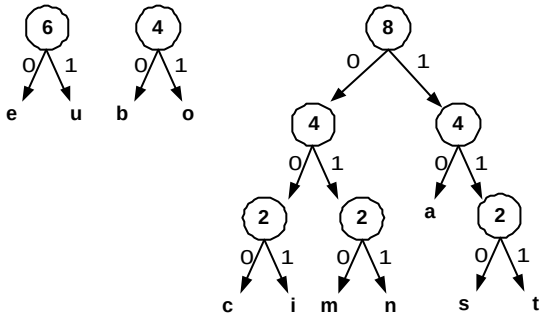
On continue à regrouper les poids 2 ce qui nous amène à grouper deux arbres :



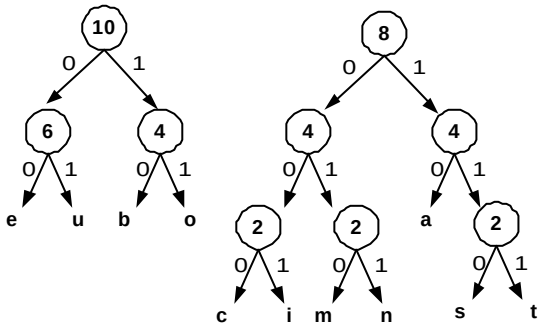
On regroupe maintenant les poids 3 :



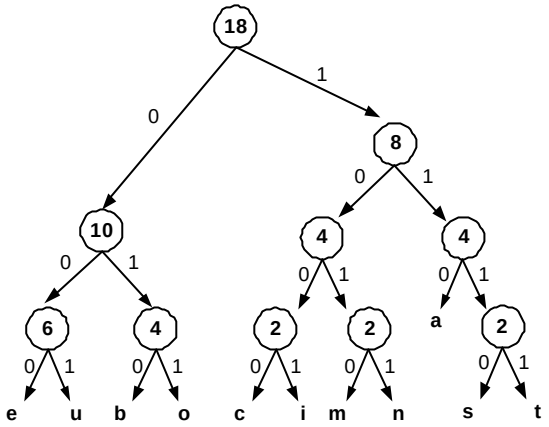
Puis les poids 4 :



Les poids 6 et 4 :



Et enfin les poids 10 et 8 :



Le parcours de l'arbre maintenant complet nous donne le codage à appliquer. Par exemple, pour la lettre *a* il faut passer par les branches `1 1` et `0`. La lettre *a* sera donc codée `110`. On appliquera donc le codage suivant :

Lettre	Fréquence	Code
e	3	`000`
u	3	`001`
a	2	`110`
b	2	`010`
o	2	`011`
c	1	`1000`
i	1	`1001`
m	1	`1010`
n	1	`1011`
s	1	`1110`
t	1	`1111`

On remarquera que les lettres les plus souvent utilisées sont codées sur moins de bits que les lettres moins souvent utilisées et qu'aucun code n'est le préfixe d'un autre. La phrase «`beautemonbeausouci`» sera donc codée ainsi :

`010000110001111100010100111011010000110001111001100110001001`

On a donc codé la phrase sur 60 bits (8 octets) au lieu de 144 bits (18 octets) précédemment, soit un gain d'environ 41%.

5.) Codage de Shannon - Fano.

Il s'agit d'un algorithme concurrent de Huffman, qui donne des résultats pratiquement équivalents. Tout comme dans l'exemple précédent, on cherche à compresser la phrase suivante :

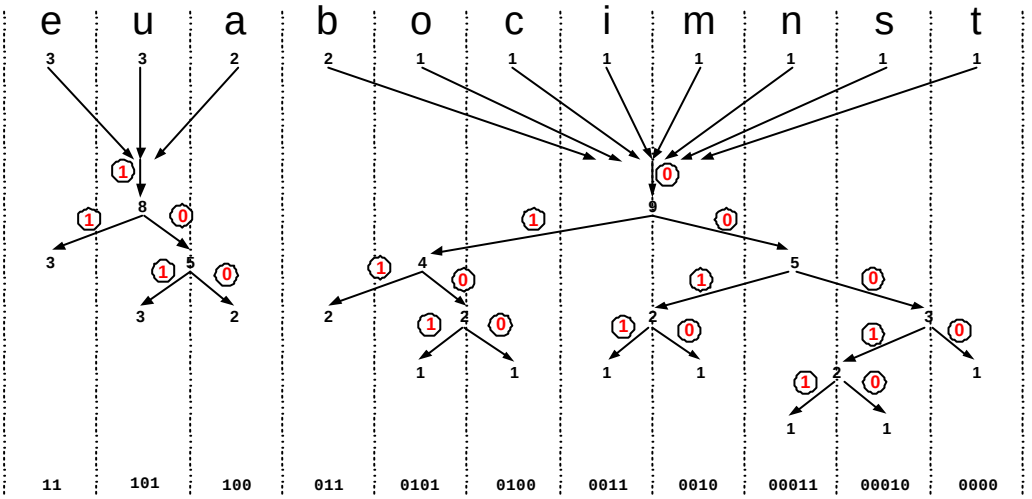
`beautemonbeausouci`

La compression se fait en 4 étapes :

1. Classer les n fréquences non nulles par ordre décroissant :

Caractères	**a**	**b**	**c**	**e**	**i**	**m**	**n**	**o**	**s**	**t**	**u**
Nombre d'apparitions	**2**	**2**	**1**	**3**	**1**	**1**	**1**	**2**	**1**	**1**	**3**

2. Partitionner la table des fréquences en deux sous tables de fréquences proches. Poursuivre l'arborescence jusqu'à ce que toute fréquence soit isolée. Lors d'une partition d'une table en deux sous-tables, on peut hésiter entre deux solutions possibles. Dans ce cas, il faut retenir la partition qui rend le rapport des deux groupes le plus proche de 1. Dans l'exemple suivant, la table (euabocimnst) est divisée en (eua) et (bocimnst) et non pas en (euab) et (ocimnst) car 8/9 est plus proche de 1 que 10/7.



3. Attribuer aux symboles dans l'arborescence le bit 0 à chaque première sous-table.
4. Attribuer aux symboles les codes binaires correspondants aux bits de description de l'arborescence. On aura donc :

Lettre	Fréquence	Code
e	3	11
u	3	101
a	2	100
b	2	011
o	2	0101
c	1	0100
i	1	0011
m	1	0011
n	1	1011
s	1	00010
t	1	0000

La phrase «`beautemonbeausouci`» sera donc codée ainsi :

```
01111100101000011001001010001101111100101000100101101010000011
```

On a donc codé la phrase sur 61 bits (8 octets) au lieu de 144 bits (18 octets) précédemment, soit un gain d'environ 41%.

6.) Algorithmes de type dictionnaire.

Une caractéristique importante des langues est de n'utiliser qu'un petit nombre de mots par rapport aux possibilités lexicographiques de la mathématique combinatoire. Ainsi, le dictionnaire français comptant environ 2000 mots de 6 lettres, on peut utiliser une compression statistique sur ces mots et réduire considérablement la quantité de bits de codage à émettre. Pour cela, il suffit de posséder le dictionnaire des mots de 6 lettres et de renvoyer le rang du mot dans l'ordre alphabétique. Sachant que pour coder 2000 mots, on a besoin de 11 bits, on vérifie que le gain en compression qu'amène ce codage est de 19 bits par mots. En disposant de dictionnaires pour chaque taille de mots, il est ainsi possible de compresser des textes, avec des gains très significatifs, en émettant pour chaque mot sa taille suivie du rang du mot dans le dictionnaire approprié. Les tailles des mots pourront être codées au moyen d'un algorithme statistique, après analyse du texte à compresser.

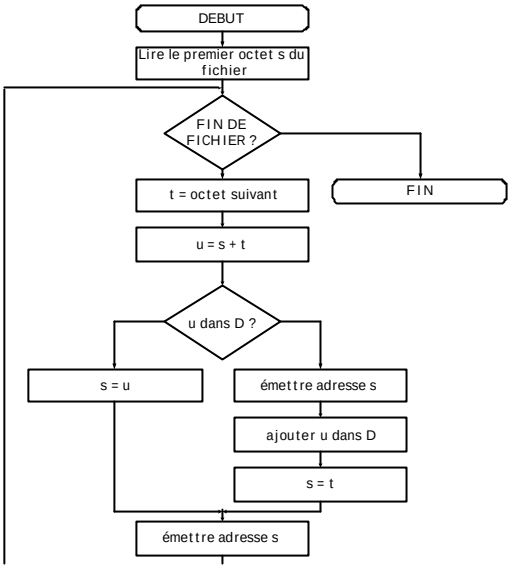
6.1.) Algorithme LZW.

La technique de compression/décompression LZW porte le nom de ses trois inventeurs : Lempel et Ziv qui l'on conçue en 1977 et Welch qui l'a améliorée en 1984. C'est un algorithme général de compression, nettement plus performant que les algorithmes statistiques, permettant d'obtenir des gains élevés sur la majorité des fichiers, sans les inconvénients des tables et des dictionnaires pré-établis. Le mécanisme repose sur l'utilisation d'un dictionnaire dynamique qui contient des séquences d'octets répétitives du fichier traité. La compression consiste à émettre non pas ces séquences mais leurs adresses dans le dictionnaire généré. L'algorithme LZW se distingue des méthodes statistiques :

- D'abord parce que le dictionnaire en tant que tel n'a pas à être sauvé avec le fichier compressé. Il est en effet reconstruit automatiquement à la décompression en fonction des données du fichier comprimé, ce qui représente un net avantage sur les méthodes statistiques qui traînent avec elles les tables d'en-tête.
- Ensuite, parce qu'il représente un algorithme d'apprentissage, dans la mesure où les séquences répétitives de symboles sont détectées puis compressées lors de leur prochaine occurrence. Les méthodes statistiques ignorent cela, puisqu'elles travaillent uniquement avec les probabilités de présence des symboles.
- Enfin parce qu'il permet le compactage « au vol » : n'ayant pas à lire le fichier au préalable, l'algorithme compresse des séquences d'octets au fur et à mesure de leur lecture ce qui est très rapide.

6.1.1.) ALGORITHME DE COMPRESSION.

Le dictionnaire D est un tableau dans lequel sont rangées des séquences de symboles de tailles variables, repérées par leurs adresses, c'est à dire leurs positions dans le tableau. Sa taille n'est pas fixe, mais en général il comprend de 2^8 à 2^{15} adresses. Les adresses 0 à 255 contiennent les codes des octets de 0 à 255 ; les séquences de symboles ont des adresses supérieures à 255. Le fichier est lu octet par octet, en effectuant les opérations suivantes :




```
 1  on lit le premier octet s du fichier
 2  Tant qu'on est pas en fin de fichier
 3       t = octet suivant
 4       u = s ⊕ t
 5       si u ∈ D alors faire s = u
 6       sinon
 7            émettre adresse(s)
 8            ajouter u dans D
 9            faire s = t
10       émettre adresse(s)
```

6.1.2.) EMISSION D'UNE ADRESSE.

La taille des adresses n'est pas constante au cours du processus : on part avec une adresse sur 8 bits (octets 0 à 255) et, dès qu'une adresse à émettre dépasse 255, il faut la coder sur 9 bits. Donc, de manière générale, on incrémente de 1 bit la taille des adresses à émettre dès qu'on atteint une nouvelle puissance de 2. On remarque donc que :

- Cet algorithme permet d'éviter de partir, dés le départ, avec la taille maximum des adresses du dictionnaire.
- Une fois la taille des adresses incrémentée, toute adresse s'écrivant sur moins de bits sera quand même émise en taille pleine.
- Il faut trouver un moyen de renseigner l'algorithme de décompression sur les endroits où la taille d'adresse a été incrémentée.

6.1.3.) EXEMPLE.

On cherche à compacter par LZW la séquence suivante :

COCORICO, CORI

On lit le premier caractère s=*C*.

Ensuite on lit le suivant : t=*O*. On forme u = *'CO'* qui n'est pas dans le dictionnaire, donc on émet l'adresse de s=*C* c'est à dire 67 (sur huit bits 01000011) ; on ajoute *'CO'* dans le dictionnaire à l'adresse 256 et on pose s=t='*O*'.

Chaîne compressée								***Dictionnaire***	
								Adresse	**Séquence**
67								256	*CO*

On continue avec le caractère suivant t = *C* On forme u = *'OC'* qui n'est pas dans le dictionnaire ; donc on écrit *'OC'* à l'adresse 257 du dictionnaire et on émet l'adresse de s=*O* c'est à dire 79 (sur huit bits 01001111) et on pose s=t='*C*'.

Chaîne compressée								***Dictionnaire***	
								Adresse	**Séquence**
67	79							256	*CO*
								257	*OC*

On continue avec le caractère suivant t = *O*. On forme u = *'CO'* qui est déjà présent dans le dictionnaire. Dans ce cas on pose s=t='*CO*'.

On continue avec le caractère suivant t = *R*. On forme u = *'COR'* qui n'est pas dans le dictionnaire ; donc on écrit *'COR'* à l'adresse 258 du dictionnaire et on émet l'adresse de s=*CO* c'est à dire 256 (sur neuf bits 100000000) et on pose s=t='*R*'.

Chaîne compressée								***Dictionnaire***	
								Adresse	**Séquence**
67	79	256						256	*CO*
								257	*OC*
								258	*COR*

On continue avec le caractère suivant t = *I*. On forme u = '*RI*' qui n'est pas dans le dictionnaire ; donc on écrit '*RI*' à l'adresse 259 du dictionnaire et on émet l'adresse de s=*R* c'est à dire 82 (sur neuf bits 001010010) et on pose s=t='*I*'.

Chaîne compressée								***Dictionnaire***	
								Adresse	**Séquence**
67	79	256	82					256	*CO*
								257	*OC*
								258	*COR*
								259	*RI*

On continue avec le caractère suivant t = *C*. On forme u = '*IC*' qui n'est pas dans le dictionnaire ; donc on écrit '*IC*' à l'adresse 260 du dictionnaire et on émet l'adresse de s=*I* c'est à dire 73 (sur neuf bits 001001001) et on pose s=t='*C*'.

Chaîne compressée								***Dictionnaire***	
								Adresse	**Séquence**
67	79	256	82	73				256	*CO*
								257	*OC*
								258	*COR*
								259	*RI*
								260	*IC*

On continue avec le caractère suivant t = *O*. On forme u = '*CO*' qui est déjà présent dans le dictionnaire. Dans ce cas on on pose s=t='*CO*'.

On continue avec le caractère suivant t =', '. On forme u = '*CO,*' qui n'est pas dans le dictionnaire ; donc on écrit '*CO,*' à l'adresse 261 du dictionnaire et on émet l'adresse de s=*CO* c'est à dire 256 (sur neuf bits 100000000) et on pose s=t=','.

Chaîne compressée								***Dictionnaire***	
								Adresse	**Séquence**
67	79	256	82	73	256			256	*CO*
								257	*OC*
								258	*COR*
								259	*RI*
								260	*IC*
								261	*CO,*

On continue avec le caractère suivant t ='_'. On forme u = ',_' qui n'est pas dans le dictionnaire ; donc on écrit ',_' à l'adresse 262 du dictionnaire et on émet l'adresse de s=',' c'est à dire 44 (sur neuf bits 000101100) et on pose s=t='_'.

Chaîne compressée								***Dictionnaire***	
								Adresse	**Séquence**
67	79	256	82	73	256	44		256	*CO*
								257	*OC*
								258	*COR*
								259	*RI*
								260	*IC*
								261	*CO,*
								262	,_

On continue avec le caractère suivant t ='*C*'. On forme u = '_*C*' qui n'est pas dans le dictionnaire ; donc on écrit '_*C*' à l'adresse 263 du dictionnaire et on émet l'adresse de s='_' c'est à dire 32 (sur neuf bits 00 100000) et on pose s=t='*C*'.

Chaîne compressée								***Dictionnaire***	
								Adresse	**Séquence**
67	79	256	82	73	256	44	32	256	*CO*
								257	*OC*
								258	*COR*
								259	*RI*
								260	*IC*
								261	*CO,*
								262	*,_*
								263	*_C*

On continue avec le caractère suivant t = *O*. On forme u = *'CO'* qui est déjà présent dans le dictionnaire. Dans ce cas on on pose s=t=*'CO'*.

On continue avec le caractère suivant t =*R*. On forme u = *'COR,'* qui est déjà présent dans le dictionnaire. Dans ce cas on on pose s=t=*'COR'*.

On continue avec le caractère suivant t =*I*. On forme u = *'CORI'* qui n'est pas dans le dictionnaire ; donc on écrit *'CORI,'* à l'adresse 264 du dictionnaire et on émet l'adresse de s=*COR* c'est à dire 258 (sur neuf bits 100000010) et on pose s=t=*I*.

Chaîne compressée								***Dictionnaire***	
								Adresse	**Séquence**
67	79	256	82	73	256	44	32	256	*CO*
258								257	*OC*
								258	*COR*
								259	*RI*
								260	*IC*
								261	*CO,*
								262	*,_*
								263	*_C*
								264	*CORI*

On émet l'adresse de s=*I* c'est à dire 73 (sur neuf bits 00 1001001).

6.1.4.) ALGORITHME DE DECOMPRESSION.

On lit les codes en partant d'une taille de 8 bits. Le principe du décompacteur est de reconstruire le dictionnaire au fur et à mesure du décompactage du fichier compressé.

1	**A** = premier code du fichier
2	emettre sequence (**A**)
3	**chaine_courante** = **A**
4	**chaine_precedente** = **chaine_courante**
5	Tant qu'on est pas en fin de fichier
6	**A** = code suivant
7	si **A** ∈ **D** alors faire
8	**chaine_courante** = sequence (**A**)
9	emettre **chaine_courante**
10	.....**t** = premier_caractere (**chaine_courante**)
11	ajouter (**chaine_precedente** + **t**) dans **D**
12	**chaine_precedente** = **chaine_courante**
13	sinon
14	emettre sequence(**A**)
15	ajouter (**chaine_precedente** + sequence(**A**)) dans **D**
16	**chaine_precedente** = sequence(**A**)
13	fin tant que

L'un des aspects les plus remarquables du processus LZW est la similitude de comportement entre le codeur et le décodeur. En effet, le dictionnaire construit en un point du fichier est le même au compactage et au décompactage. Le décodeur reconstruit linéairement la table des séquences repérées par le codeur en se servant des données mêmes du fichier compressé.

6.1.5.) EXEMPLE.

On cherche à décompresser la séquence suivante :

67	79	256	82	73	256	44	32	258	73

On lit le premier code **A**=67 et on émet sequence(**A**).
On a alors **chaine_courante** = **chaine_precedente** = sequence(**A**) = "*C*"

Chaîne décompressée								***Dictionnaire***	
								Adresse	**Séquence**
C	*O*								

On lit le suivant : **A** =79 qui n'est pas dans le dictionnaire.
On émet sequence(**A**), c'est à dire "*O*".
On ajoute **chaine_precedente** +sequence(**A**) (c'est à dire '*CO*') dans le dictionnaire à l'adresse 256
On pose **chaine_precedente** = sequence(**A**) ="*O*".

Chaîne décompressée								*Dictionnaire*	
								Adresse	**Séquence**
C	*O*							256	*CO*

On lit le suivant **A** =256, qui est dans le dictionnaire.
On forme alors **chaine_courante** = sequence(**A**) = *CO*
On émet **chaine_courante** (c'est à dire *CO*)
On pose **t** = premier caractère de **chaine_courante**, c'est à dire *C*.
On ajoute **chaine_precedente + t** (c'est à dire '*OC*') dans le dictionnaire à l'adresse 257.
On pose **chaine_precedente** = **chaine_courante** = "*CO*"

Chaîne décompressée								*Dictionnaire*	
								Adresse	**Séquence**
C	*O*	*C*	*O*					256	*CO*
								257	*OC*

On lit le suivant : **A** =82 qui n'est pas dans le dictionnaire.
On émet sequence(**A**), c'est à dire "*R*".
On ajoute **chaine_precedente** +sequence(**A**) (c'est à dire '*COR*) dans le dictionnaire à l'adresse 258
On pose **chaine_precedente** = sequence(**A**) ="*R*.

Chaîne décompressée								*Dictionnaire*	
								Adresse	**Séquence**
C	*O*	*C*	*O*	*R*				256	*CO*
								257	*OC*
								258	*COR*

On lit le suivant : **A** =73 qui n'est pas dans le dictionnaire.
On émet sequence(**A**), c'est à dire "*I*".
On ajoute **chaine_precedente** +sequence(**A**) (c'est à dire '*RI')* dans le dictionnaire à l'adresse 259
On pose **chaine_precedente** = sequence(**A**) ="*I*"

Chaîne décompressée								*Dictionnaire*	
								Adresse	**Séquence**
C	*O*	*C*	*O*	*R*	*I*			256	*CO*
								257	*OC*
								258	*COR*
								259	*RI*

On lit le suivant **A** =256, qui est dans le dictionnaire.
On forme alors **chaine_courante** = sequence(**A**) = *CO*
On émet **chaine_courante** (c'est à dire *CO*)
On pose **t** = premier caractère de **chaine_courante**, c'est à dire *C*.
On ajoute **chaine_precedente + t** (c'est à dire '*IC*') dans le dictionnaire à l'adresse 260.
On pose **chaine_precedente** = **chaine_courante** = "*CO*"

Chaîne décompressée								***Dictionnaire***	
								Adresse	**Séquence**
C	*O*	*C*	*O*	*R*	*I*	*C*	*O*	256	*CO*
								257	*OC*
								258	*COR*
								259	*RI*
								260	*IC*

On lit le suivant : **A** =44 qui n'est pas dans le dictionnaire.

On émet sequence(**A**), c'est à dire ",".

On ajoute **chaine_precedente** +sequence(**A**) (c'est à dire '*CO,'*) dans le dictionnaire à l'adresse 261

On pose **chaine_precedente** = sequence(**A**) =","

Chaîne décompressée								***Dictionnaire***	
								Adresse	**Séquence**
C	*O*	*C*	*O*	*R*	*I*	*C*	*O*	256	*CO*
,								257	*OC*
								258	*COR*
								259	*RI*
								260	*IC*
								261	*CO,*

On lit le suivant : **A** =32 qui n'est pas dans le dictionnaire.

On émet sequence(**A**), c'est à dire "_".

On ajoute **chaine_precedente** +sequence(**A**) (c'est à dire '*,_'*) dans le dictionnaire à l'adresse 262

On pose **chaine_precedente** = sequence(**A**) ="_"

Chaîne décompressée								***Dictionnaire***	
								Adresse	**Séquence**
C	*O*	*C*	*O*	*R*	*I*	*C*	*O*	256	*CO*
,	_							257	*OC*
								258	*COR*
								259	*RI*
								260	*IC*
								261	*CO,*
								262	*_,*

On lit le suivant **A** =258, qui est dans le dictionnaire.

On forme alors **chaine_courante** = sequence(**A**) = *COR*

On émet **chaine_courante** (c'est à dire *COR*)

On pose **t** = premier caractère de **chaine_courante**, c'est à dire *C*.

On ajoute **chaine_precedente + t** (c'est à dire '*_C*') dans le dictionnaire à l'adresse 263.

On pose **chaine_precedente** = **chaine_courante** = "*COR*"

Chaîne décompressée								***Dictionnaire***	
								Adresse	**Séquence**
C	*O*	*C*	*O*	*R*	*I*	*C*	*O*	256	*CO*
,	*_*	*C*	*O*	*R*				257	*OC*
								258	*COR*
								259	*RI*
								260	*IC*
								261	*CO,*
								262	*_,*
								263	*_C*

On lit le suivant : **A** =73 qui n'est pas dans le dictionnaire.

On émet sequence(**A**), c'est à dire "*I*".

On ajoute **chaine_precedente** +sequence(**A**) (c'est à dire *'CORI')* dans le dictionnaire à l'adresse 264

On pose **chaine_precedente** = sequence(**A**) ="*I*"

Chaîne décompressée								***Dictionnaire***	
								Adresse	**Séquence**
C	*O*	*C*	*O*	*R*	*I*	*C*	*O*	256	*CO*
,	*_*	*C*	*O*	*R*	*I*			257	*OC*
								258	*COR*
								259	*RI*
								260	*IC*
								261	*CO,*
								262	*_,*
								263	*_C*
								264	*CORI*

6.2.) Exercices.

EXERCICE 8.1

Il y a 4000 ans, les Sumériens savaient déjà calculer $\sqrt{a}$ à partir de a ($a \in R^{+*}$). Leur méthode était la suivante :

Si on part de l'égalité $a = \sqrt{a} * \sqrt{a}$. On peut en déduire $\sqrt{a} = \frac{a}{\sqrt{a}}$ et $\sqrt{a} = \frac{1}{2}\left(\sqrt{a} + \frac{a}{\sqrt{a}}\right)$. Cette relation conduit à définir la récurrence $x_{n+1} = \frac{1}{2}\left(x_n + \frac{a}{x_n}\right)$ à partir d'un $x_0 \in R^{+*}$. Cette récurrence a les propriétés suivantes :

$$\begin{cases} x_1 > x_2 > x_3 \ \ldots \ > x_n \\ \lim\limits_{n \to +\infty} x_n = \sqrt{a} \end{cases}$$

En utilisant cette méthode de calcul, écrivez une fonction récursive calculant la racine d'un nombre réel positif en n itérations.

EXERCICE 8.2

Un palindrome est une phrase, un nombre, un message, qui, si l'on ne tient compte ni des espaces ou apostrophes, ni des signes de ponctuation, peut être lu de droite à gauche ou de gauche à droite en gardant le même sens ou plutôt la même signification. Exemples de palindromes célèbres :

- Bob.
- Ressasser.
- Esope reste ici et se repose.
- Elu par cette crapule.
- « Oh ! Cela te perd ! » Répéta l'écho.
- Red rum, Sir, is murder[4].
- Tu l'as trop écrasé, César, ce Port-Salut.
- Eric, notre valet, alla te laver ton ciré[5].
- La mariée ira mal.
- L'âme des uns jamais n'use de mal[6].

Ecrire une fonction qui reçoit une chaîne *S* en paramètre, la converti en majuscules, supprime la ponctuation, les espaces, les apostrophes, les traits d'union... et renvoie vrai si *S* est un palindrome.

EXERCICE 8.3

Ecrire une fonction qui écrive un prénom passé en paramètre sous la forme :

- Première lette en majuscules.

[4] Le rhum vieux, Monsieur, est meurtrier.
[5] Pour ce palindrome, il vaut mieux que le valet ne s'appelle pas Luc !
[6] Ce palindrome date du moyen âge, époque où l'on confondait le « i » et le « j ».

- Lettes suivantes en minuscules.

Pour les prénoms composés, chaque partie du prénom devra être traitée séparément.

Exemples :

« **Maud** », « **Paul** », « **Jean-Paul** », « **Anne Victoire** » ...