AMC DACTYLO

Rapport algorithmique

Aurélien Méré
FIIFO1

SOMMAIRE

INTRODUCTION GENERALE 4

A PROPOS DE CE MANUEL 4

PREMIERE PARTIE : GESTION DU DICTIONNAIRE 5

Exploitation du dictionnaire 5
Format propre du dictionnaire 5
Structure de donnée employée 5
Paquetage Dictionnaire 6
Variables utilisées dans le paquetage Dictionnaire 6
Liste des différentes procédures et fonctions 7
Fonction Charge_Dictionnaire 7
Fonction Convertit_Dictionnaire 7
Procedure Infos_Dictionnaire 7
Procedure Tirage_Mot 7
Fonction Charge_Dictionnaire 8
Fonction Convertit_Dictionnaire 9
Procedure Infos_Dictionnaire 10
Procedure Tirage_Mot 10
Exemple complet de fonctionnement 11
1) Chargement du dictionnaire 11
2) Tirage aléatoire d'un mot 12
3) Informations sur le dictionnaire 12

DEUXIEME PARTIE : GESTION DU CLAVIER 13

Introduction 13
Fichier de clavier 13
Stockage et exploitation du clavier 13
Paramètres des touches 14
1) Le code ASCII 14
2) La main 14
3) La position sur le clavier 14
4) Indice de présence 14
Paquetage Clavier 15
Fonction Charge_Clavier 15
Exemple de fonctionnement 17

TROISIEME PARTIE : CALCUL DE COMPLEXITE 18

Introduction **18**
Analyse d'un mot **18**
Calcul de la distance entre deux lettres **18**
Exemple théorique **20**
Avantages de cet algorithme **20**
Doubles lettres particulières **20**
Les différentes sous-routines et macros **21**
1) Procedure SepareMains 21
2) Fonction Distance 22
3) Fonction CompF 22
4) Fonction Complexite_Mot 22
Exemple pratique **23**
Position des lettres 23
Tableau de complexité F par défaut (présence dans la langue française) 23
Exemple n°1 : TOASTS **23**
Passage dans la procédure SepareMains 23
Passage dans la boucle de la fonction Complexite_Mot 24
Calcul de la complexité de présence 25
Exemple n°2 : BEBE **25**
Séparation des mains 25
Algorithme de distance 26
Complexité de présence 26
Différents exemples résultants **26**
Complexité des algorithmes **26**

QUATRIEME PARTIE : PROCESSUS PRINCIPAL 27

Diagramme de fonctionnement **27**
L'interface de test **27**
L'interface de test (diagramme) **28**
Paquetage Interface **29**
Ligne de commande **29**
Rappels sur les modes de fonctionnement **29**
Rappels sur le choix des mots durant le test **29**
Affichage du clavier **30**
Entrée du mot par l'utilisateur **30**
Interface de test (algorithmique) **30**
Calcul du score **31**
Fonction VerifieMot **32**
1) Algorithme 32
2) Test de fiabilité 32
Evaluation du score global **33**
Mode débutant 33
Mode expert 33
Evolution de la complexité au cours du jeu **34**

Introduction générale

A propos de ce manuel

Ce manuel explique le fonctionnement du logiciel AMC Dactylo et en détaille les algorithmes, les structures de données employées, ainsi que les moyens mis en œuvre pour optimiser au mieux le programme. Des jeux d'essai sont présentés afin de montrer pratiquement l'effet de ces algorithmes.

Ce manuel est divisé de la même manière que le programme source d'AMC Dactylo. Effectivement, le programme a été conçu de façon modulaire dans un souci de clarté et de facilité de compréhension du code source.

Les différents modules qui seront étudiés tout au long de ce manuel concernent le processus principal, la gestion du dictionnaire de mots, le calcul de la complexité des mots ainsi que la gestion de l'interface utilisateur.

Un diagramme représentant la structure générale du programme est présenté au début de la quatrième partie, pour comprendre rapidement le fonctionnement global du programme.

Le sommaire détaillé est la meilleure manière de naviguer à travers cette documentation.

Première partie : Gestion du dictionnaire

Exploitation du dictionnaire

AMC Dactylo fonctionne avec un format propre de dictionnaire où les mots sont triés par complexité croissante. Il possède en standard une procédure qui lui permet de convertir un dictionnaire classique en un dictionnaire propre. Il faut rappeler qu'un dictionnaire classique est un fichier texte contenant un mot différent à chaque ligne. Ce dictionnaire n'est pas obligatoirement trié.

Format propre du dictionnaire

Le dictionnaire propre d'AMC Dactylo est formellement très similaire à un dictionnaire classique. L'unique différence est la présence du mot CPX à chaque progression d'un niveau de complexité. Ainsi, en supposant que l'on ait 4 mots de complexité 1 et 2 mots de complexité 2, le dictionnaire ressemblerait à ceci :

```
CPX             -- On passe au niveau de complexité 1
MOT1
MOT2
MOT3
MOT4
CPX             -- On passe au niveau de complexité 2
MOT5
MOT6
```

Lors du démarrage du programme, le processus principal vérifie l'existence du dictionnaire propre. Si celui-ci est introuvable, il lance la procédure de conversion du dictionnaire classique, qui pourra éventuellement être entré en paramètre lors du lancement du programme. Ceci a un avantage double : d'une part, il est possible d'utiliser des dictionnaires totalement différents avec le programme. D'autre part, le tri par complexité permet de manipuler beaucoup plus aisément la progression de l'utilisateur au cours du programme.

Structure de donnée employée

Nous avons décidé d'utiliser une structure assez particulière pour gérer les différents mots du dictionnaire. Tout d'abord, il est nécessaire de mettre les mots de complexité égale ensemble, et de les séparer de ceux de complexité différente. Il faut par ailleurs être en mesure de récupérer aisément un mot parmi tous ceux présents.

Une liste chaînée circulaire a été employée pour stocker une liste de mots de même complexité. Il suffit ainsi de choisir le pointeur vers l'élément suivant pour obtenir un autre mot : la navigation en est ainsi facilitée. Le fait qu'elle soit circulaire est purement pratique : nous étudierons ceci dans la fonction de tirage d'un mot.

A côté de la liste chaînée, on emploie un tableau de pointeurs vers le début de chaque liste, chacune contenant des mots de complexités différentes. En langage algorithmique, cela se traduit ainsi :

```
Type ListeMots = Enreg(Mot :chaîne(1..MAX_TAILLE) ;
                       Ln : entier ;
                       Suivant : pointeur(ListeMots)) ;

ptrMots :Tableau(1..MAX_COMPLEXITE) de pointeur(ListeMots) ;
```

Chaque élément d'une liste chaînée contient trois champs : un mot de taille fixe, celle-ci étant majorée par la constante MAX_TAILLE, la longueur du mot Ln, et le pointeur sur l'élément suivant.

Paquetage Dictionnaire

Le paquetage dictionnaire exporte les fonctions suivantes :

- Charge_Dictionnaire
- Convertit_Dictionnaire
- Infos_Dictionnaire
- Tirage_Mot

Variables utilisées dans le paquetage Dictionnaire

Le paquetage dictionnaire exploite en interne les variables suivantes :

```
-- Gestion des mots
PtrMots :tableau(1..MAX_COMPLEXITE) de pointeur(ListeMots);

-- Statistiques
NbMots : entier ;
NbCaracteres : entier ;

-- Nombres aléatoires
Gen : Générateur
```

Liste des différentes procédures et fonctions

Fonction Charge_Dictionnaire

La fonction Charge_Dictionnaire a pour rôle de charger en mémoire tous les mots d'un dictionnaire. Ce dictionnaire est un dictionnaire propre AMC Dactylo, et de ce fait trié par complexité. Elle prend en argument le nom du dictionnaire. Elle retourne un booléen pour confirmer ou non le chargement.

Fonction Convertit_Dictionnaire

La fonction Convertit_Dictionnaire a pour rôle, comme son nom l'indique, de convertir un dictionnaire classique en un dictionnaire propre. Cette fonction fait appel à la procédure de calcul de complexité d'un mot que nous détaillerons dans les chapitres suivants. Cette fonction prend en entrée le nom du fichier source, ainsi que le nom du fichier de destination.

Procedure Infos_Dictionnaire

Cette procédure renvoie des informations sur un dictionnaire, et notamment le nombre de mots ainsi que le nombre de caractères. C'est une fonction purement statistique.

Procedure Tirage_Mot

Cette procédure renvoie aléatoirement un mot d'une complexité CPX. La procédure renvoie le mot tiré ainsi que sa longueur respectivement dans les arguments Mot et Ln.

On suppose que le générateur de nombre aléatoires *GEN* a été initialisé auparavant. Ceci se fait durant l'implémentation dans le langage choisi.

Cette procédure montre parfaitement l'intérêt d'une liste chaînée circulaire. Un nombre est choisi aléatoirement et on parcourt la liste autant de fois que nécessaire. Que la liste soit très grande ou très réduite, la valeur tirée aléatoirement peut être quelconque, étant donné que la liste boucle sur elle-même. Cela permet d'éviter la recherche de fin de liste, pour boucler manuellement sur le premier élément, qu'il faudra avoir préalablement stocker en mémoire, ce qui consomme un espace non négligeable.

Fonction Charge_Dictionnaire

```
Fonction Charge_Dictionnaire(nom : chaîne) : booléen
Var            str : chaîne(1..255) ;
               L , X: entier := 0;
               F : fichier ;
               CpxMots:tableau(1..MAX_COMPLEXITE)de ptr(ListeMots);

Début
        Ouvrir (f, nom)     -- Ouvre le fichier

        ■ On vérifie que la première ligne est bien CPX
        ■ Autrement le fichier n'est pas un dictionnaire valide

        Lire_ligne(f, str, L) ;
        Si str(1..3) /= « CPX » alors retourne faux ; fin si ;

        ■ On initialise le tableau de pointeurs
        ■ CpxMots conserve le pointeur sur le début de liste

        Pour i de 1..MAX_COMPLEXITE faire
              PtrMots(i) := Allouer (ListeMots) ;
              CpxMots(i) := PtrMots(i);
        Fin Pour;

        ■ On parcourt tout le dictionnaire
        ■ Et on ajoute tous les mots dans PtrMots

        X := 1 ;
        Tant que non Fin_fichier(f) faire
              Lire_ligne(f, str, L) ;

              ■ S'il s'agit du mot « CPX » alors on augmente
              ■ le niveau de complexité d'un point
              ■ Autrement, on ajoute le mot à la liste courante.

              Si str(1..3) = « CPX » alors
                    X := X + 1 ;
              Sinon
                     ■ On créé un nouvel élément
                     ■ Et on y place le nouveau mot
                    PtrMots(X)^.Suivant := Allouer(ListeMots) ;
                    PtrMots(X) := PtrMots(X).Suivant ;
                    PtrMots(X)^.Mot := str(1..32) ;
                    PtrMots(X)^.Ln := L ;

                    NbMots := NbMots + 1 ;
                    NbCaracteres := NbCaracteres + L ;
              Fin si ;
        Fin Tant que ;

        Fermer (f) ;

        ■ On effectue la liaison circulaire en chargeant comme
        ■ suivant du dernier élément le pointeur vers le premier élément
        ■ Si aucun mot n'a été ajouté à la liste, celle-ci reste nulle

        Pour i de 1..MAX_COMPLEXITE
              PtrMots(X)^.Suivant := CpxMots(X)^.Suivant ;
              Desallouer(CpxMots(X)) ;
              ■ On revient au début de la liste
              PtrMots(X) := PtrMots(X)^.Suivant
        Fin Pour
      Retourne(vrai) ;
Fin ;
```

Fonction Convertit_Dictionnaire

```
 Fonction Convertit_Dictionnaire(  src : chaîne,
                                   dst : chaîne) : booléen
Var     str : chaîne(1..255) ;
        L , X: entier := 0;
        F : fichier ;
        tmpMots: tableau(1..MAX_COMPLEXITE) de pointeur(ListeMots);
        CpxMots: tableau(1..MAX_COMPLEXITE) de pointeur(ListeMots);

Début
        Ouvrir (f, src)     -- Ouvre le fichier source

        ■  On initialise un tableau de pointeurs temporaire
        ■  CpxMots conserve le pointeur sur le début de liste

        Pour i de 1..MAX_COMPLEXITE faire
              tmpMots(i) := Allouer (ListeMots) ;
              CpxMots(i) := tmpMots(i);
        Fin Pour;

        ■  On parcourt le fichier source
        ■  On calcule dynamiquement la complexité
        ■  Et on ajoute le mot dans la liste correspondante

        Tant que non Fin_fichier(f) faire
              Lire_ligne(f, str, L) ;

              X := Complexite_Mot(str, L) ;

              tmpMots(X)^.Suivant := Allouer(ListeMots) ;
              tmpMots(X) := tmpMots(X).Suivant ;
              tmpMots(X)^.Mot := str(1..32) ;
              tmpMots(X)^.Ln := L ;
              tmpMots(X)^.Suivant := NULL ;

              NbMots := NbMots + 1 ;
              NbCaracteres := NbCaracteres + L ;
        Fin Tant que ;

        Ferme (f) ;

        ■  On écrit le nouveau dictionnaire

        Créer (f, dst) ;

        Pour i de 1..MAX_COMPLEXITE
              Ecrire(« CPX ») ;   -- Changement de complexité
              Tant que (cpxMots(i)^.Suivant /= NULL) faire

                    ■  On écrit les mots ligne par ligne

                    Ecrire_ligne(f, cpxMots(i)^.Mot(1..cpxMots(i)^.Ln)) ;
                    CpxMots(i) := cpxMots(i)^.Suivant;
              Fin tant que
        Fin Pour;

        Ferme (f) ;
        Retourne(vrai) ;
Fin ;
```

Procedure Infos_Dictionnaire

```
Procedure Infos_Dictionnaire (NbMot : Entier {S}, NbCar : Entier {S})
Début
   NbMot := NbMots ;
   NbCar := NbCaracteres ;
Fin ;
```

Procedure Tirage_Mot

```
Procedure Tirage_Mot(Cpx : Entier {E}, Mot : Chaîne {S}, Ln : Entier {S})
Var Rnd : Entier ;   -- Nombre tiré aléatoirement
Début

        ■ Si tous les mots de la liste sont épuisés,
        ■ On renvoie un mot vide

  Si (PtrListe(Cpx) = NULL) alors
        Ln := 0 ;
  Sinon
        Rnd := Entier(Random(Gen) * 100.0) ;

        ■ On parcourt rnd fois la liste de complexité donnée
        ■ Pour obtenir un mot au hasard

        Pour i de 1 à rnd faire
              PtrListe(cpx) := ptrListe(cpx)^.Suivant ;
        Fin Pour


        ■ On enregistre le mot tiré
        ■ Dans les variables prévues à cet effet

        Mot := PtrListe(cpx)^.Suivant^.Mot ;
        Ln := PtrListe(cpx)^.Suivant^.Ln ;

        ■ On vérifie qu'il y a plus d'un mot dans la liste

        Si PtrListe(cpx)^.Mot := PtrListe(cpx)^.Suivant^.Mot alors

        ■ Si ce n'est pas le cas, on considère que
        ■ La liste est désormais vide

              Desallouer(PtrListe(cpx)) ;
              PtrListe(cpx) := NULL ;
        Sinon

        ■ Sinon, on efface le mot de la liste
        ■ Pour éviter de l'avoir en double

              Désallouer(PtrListe(cpx)^.Suivant) ;
              ptrListe(cpx)^.Suivant := ptrListe(cpx)^.Suivant^.Suivant ;
        Fin Si ;
  Fin Si ;
Fin ;
```

Exemple complet de fonctionnement

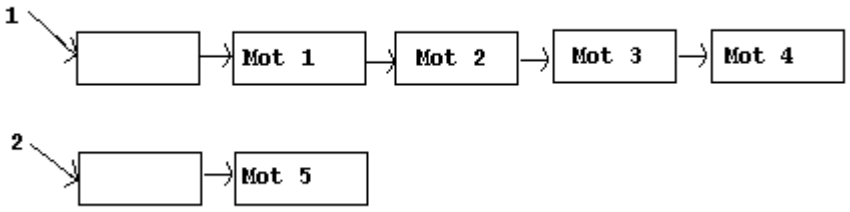
Imaginons le dictionnaire propre suivant :

```
CPX
MOT1
MOT2
MOT3
MOT4
CPX
MOT5
```

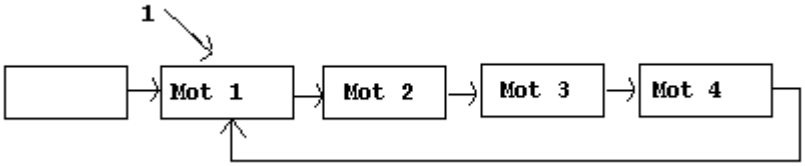
1) Chargement du dictionnaire

Appelons la procédure Charge_Dictionnaire. Nous omettrons de détailler toutes les appels relatifs aux traitements du fichier et nous considérerons que MAX_COMPLEXITE est fixée à la valeur 2 :

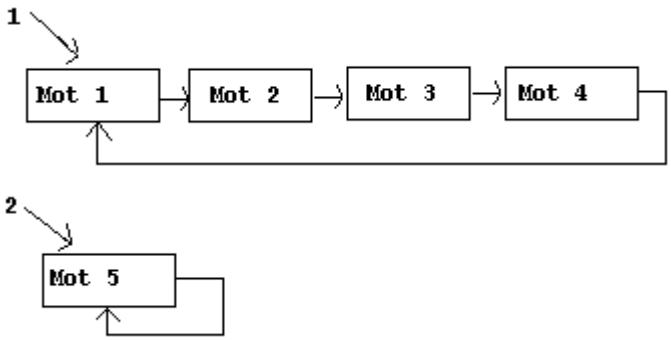
- Vérifie que la première ligne vaut bien CPX => OUI
- Initialise le tableau de pointeurs et crée 2 liste chaînées
- X=1. La lecture sort « MOT1 ». On l'ajoute à la liste ptrMots(1)
- De même avec « MOT2 », « MOT3 » et « MOT4 »
- La lecture sort « CPX » donc on incrémente X => X=2
- La lecture sort « MOT5 ». On l'ajoute à la liste ptrMots(2)
- A ce stade, voici l'état de ptrMots (on ne présentera pas Ln) :



- On effectue ensuite la liaison circulaire :



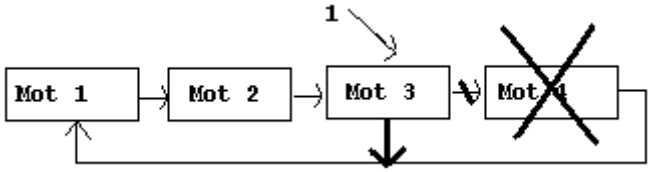
- Puis on désalloue l'élément inutilisé :



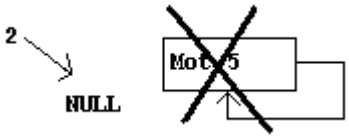
2) Tirage aléatoire d'un mot

Imaginons l'appel à la procédure Tirage_Mot avec comme argument CPX=1 et supposons que la fonction RANDOM renvoie la valeur 2 :

- Après la boucle, nous ne pointons plus sur « mot1 » mais sur « mot3 »
- On affecte les valeurs aux variables, à savoir mot = « MOT4 » et Ln = 4
- Comme la liste n'est pas nulle, on désalloue l'élément suivant :



- Imaginons le cas extrême où la liste ne possède qu'un seul élément. On appelle la procédure avec CPX = 2. La valeur de la fonction RANDOM n'a aucune importance puisque la liste boucle toujours sur le même élément.
- Comme on a ptrMots(2)=ptrMots(2)^.Suivant, la procédure renvoie Mot=« MOT5 » et Ln=4 et au lieu de désallouer l'élément suivant, la liste « s'autodétruit » :



3) Informations sur le dictionnaire

Un appel à la procédure Infos_Dictionnaire renverrait tout naturellement les valeurs suivantes :

NbMots = 5
NbCaracteres = 20

Deuxième partie : Gestion du clavier

Introduction

Le clavier a une importance majeure dans AMC Dactylo, étant donné que la disposition des touches sur le clavier est la principale source de difficulté pour taper un mot. Il convient donc de le gérer le plus efficacement possible, avec une évolutivité maximale. Dans notre structure, chaque touche est gérée individuellement et possède un certains nombres de caractéristiques, au nombre de cinq :

- Le caractère ASCII qu'elle représente
- La main qui doit être utilisée pour frapper la touche
- La position horizontale de la touche
- La position verticale de la touche
- L'indice de complexité F (indice de mémorisation)

Chacun de ses paramètres sera détaillé dans la suite de ce chapitre. Il était possible de coder en dur les différents paramètres dans le programme, mais ceci n'est pas une manière propre de faire les choses. Le gain de vitesse offert par le codage en dur ne peut compenser le manque de lisibilité du programme. En revanche, l'utilisation d'un fichier pour enregistrer tous ces paramètres s'avère beaucoup plus utile car ceci offre une évolutivité et une adaptation supplémentaire au programme. Effectivement, il est ainsi possible de créer un fichier de clavier pour chaque pays, d'ajouter des touches de fonction, le pavé numérique, et bien d'autres possibilités.

Fichier de clavier

Le fichier de clavier est structuré pour un accès séquentiel. Chaque ligne contient un paramètre et toutes les cinq lignes, on change de lettre. Les différents paramètres sont listés dans l'ordre cité précédemment. Il sera éventuellement possible dans l'avenir de réaliser un logiciel de gestion de clavier pour créer dynamiquement des claviers, ce qui facilite grandement l'évolutivité du programme.

Stockage et exploitation du clavier

Il semble évidemment hors de question pour des problèmes de rapidité et de respect du matériel d'accéder au fichier de clavier à chaque recherche d'une valeur. La structure la plus pratique pour stocker chaque lettre et ses paramètres est un tableau à deux dimensions. Le premier indice sert à identifier une touche, et le deuxième indice sert à choisir le paramètre de la lettre que l'on veut récupérer.

```
Clavier : tableau(1..NBTOUCHES, 1..NBPARAMS) d'entiers

NB_TOUCHES : Constante := 102 ;
NB_PARAMS : Constante := 5 ;

PRM_CODEASCII := 1 ;
PRM_MAIN := 2 ;
PRM_POSH := 3 ;
PRM_POSV := 4 ;
PRM_CPXF := 5;
```

L'utilisation de constantes permet de modifier la structure du clavier sans avoir à réécrire toutes les routines et fonctions faisant appel au tableau clavier, ce qui procure un confort de programmation et de relecture du code source non négligeable.

Paramètres des touches

1) Le code ASCII

Le Code ASCII de la touche permet tout simplement d'identifier la touche. La majorité des langages de programmation possèdent une fonction permettant de convertir un code ASCII en caractère, ce qui permet également de faire des recherches.

2) La main

Ce paramètre permet de déterminer si la main utilisée pour frapper la touche est la main gauche ou la main droite. Le calcul de la complexité d'un mot fonctionne différemment si deux lettres contigües ne sont pas tapées par la même main. L'indice 1 représente la main gauche, l'indice 2 la main droite. L'évolutivité du programme rend même possible le test des individus à 3 mains en mettant l'indice 3.

3) La position sur le clavier

La position de la touche sur le clavier permet d'une part d'évaluer facilement la distance entre deux touches sur le clavier, mais également de faire une représentation graphique aisée du clavier à l'écran. Sur un clavier standard, la position (1,1) correspond à la touche Escape.

4) Indice de présence

Cet indice est utilisé dans l'algorithme de complexité. Il donne une idée de la fréquence du caractère dans la langue française, et donc la capacité à l'utilisateur de se souvenir de cette touche et donc de l'atteindre plus facilement. Si un caractère est plus fréquent, l'utilisateur le repèrera plus vite sur le clavier et la complexité en est donc réduite.

Paquetage Clavier

Dans un soucis de modularité, un paquetage Clavier est créé. Celui-ci contient toutes les routines relatives au clavier. Le paquetage Clavier exporte une fonction unique :

```
Charge_Clavier (src :string) : clavier ;
```

A noter que le type clavier est défini comme suit :

```
Type Clavier:Tableau(1..NB_TOUCHES, 1.. NB_PARAMS) d'entiers
```

Fonction Charge_Clavier

La fonction Charge_Clavier sert à retourner une structure de type clavier extraite du fichier src mis en argument de la fonction. La structure de clavier retournée par la fonction sera utilisée notamment pour évaluer la complexité des mots.

Le temps mis par l'algorithme pour charger un clavier dépend uniquement de la taille du fichier clavier. L'algorithme a donc une complexité de l'ordre de n * NB_PARAMS, où n représente le nombre de touches stocké dans le fichier de clavier. L'algorithme est présenté dans la page suivante.

Il est possible qu'une erreur se produise lors du chargement du clavier, soit parce que le fichier est inexistant, soit parce qu'il est corrompu. Si un tel cas se produit, une exception de fin de fichier se produit et on rentre dans un bloc de traitement spécifique. La fonction renvoie alors un clavier contenant la valeur 0 dans la case d'indice (1,1) pour indiquer qu'une erreur est survenue durant le chargement et que le clavier n'est pas valide.

```
Fonction Charge_Clavier (src :string) : clavier ;
Var     c :clavier ;
        I, l :entier ;
        F : fichier ;
        Str : chaîne(1..255) ;
Début
        C(1,1) := 0 ;

        ■ On ouvre le fichier source

        Ouvrir (F, src) ;

        ■ On charge les paramètres jusqu'à arriver
        ■ A la fin du fichier

        I := 1 ;

        Tant Que Non Fin_de_Fichier(f) faire

              Pour k de 1 à NB_PARAMS faire
                    Lire_ligne (f, str, l) ;

                    C(i, k) := valeur(str(1..l)) ;
              Fin Pour ;

              I := i +1 ;
        Fin Tant Que

        ■ On ferme le fichier source
        ■ Et on renvoie le clavier terminé

        Fermer (f) ;

        Retourne(c) ;

        Bloc_Exceptions
        Début

              ■ Si jamais on atteint la fin du fichier pendant
              ■ La boucle, le fichier est corrompu
              ■ C'est également le cas si une erreur se produit
              ■ Lors de l'ouverture du fichier
              ■ Dans ce cas, on renvoie un clavier nul

              C(1, 1) := 0 ;
              Retourne (c) ;
        Fin ;
    Fin
```

Exemple de fonctionnement

Nous utiliserons pour notre exemple le fichier clavier suivant :

```
97       -- Code ASCII pour la lettre « a »
1        -- Main gauche
2        -- Position horizontale : 2
3        -- Position verticale : 3
1        -- Indice de présence : 1
122      -- Code ASCII pour la lettre « z »
1        -- Main gauche
3        -- Position horizontale : 3
3        -- Position verticale : 3
3        -- Indice de présence : 3
98       -- Code ASCII pour la lettre « b »
2        -- Main droite
7        -- Position horizontale : 7
5        -- Position verticale : 5
2        -- Indice de présence : 2
```

Voici l'exécution en pas-à-pas du programme :

`I=1; k=1 ; str = « 97 ».` On a alors C(1,1)=97
`I=1; k=2 ; str = « 1 ».` On a alors C(1,2)=1
`I=1; k=3 ; str = « 2 ».` On a alors C(1,3)=2
`I=1; k=4 ; str = « 3 ».` On a alors C(1,4)=3
`I=1; k=5 ; str = « 1 ».` On a alors C(1,5)=1
`I=2; k=1 ; str = « 122 ».` On a alors C(2,1)=122
`I=2; k=2 ; str = « 1 ».` On a alors C(2,2)=1
`I=2; k=3 ; str = « 3 ».` On a alors C(2,3)=3
`I=2; k=4 ; str = « 3 ».` On a alors C(2,4)=3
`I=2; k=5 ; str = « 3 ».` On a alors C(2,5)=3
`I=3; k=1 ; str = « 98 ».` On a alors C(2,1)=98
`I=3; k=2 ; str = « 2 ».` On a alors C(2,2)=2
`I=3; k=3 ; str = « 7 ».` On a alors C(2,3)=7
`I=3; k=4 ; str = « 5 ».` On a alors C(2,4)=5
`I=3; k=5 ; str = « 2 ».` On a alors C(2,5)=2

Voici donc le clavier retourné par la fonction :

Indices	**Code ASCII**	**Main**	**Pos. H**	**Pos. V**	**Présence**
1	97	1	2	3	1
2	122	1	3	3	3
3	98	2	7	5	2

Troisième partie : Calcul de complexité

Introduction

Comme nous l'avons vu dans le chapitre de gestion du clavier, la complexité d'un mot est évaluée suivant trois critères : la position de la lettre sur le clavier, la présence de doubles-lettres ou de doubles-couples comme dans le mot « papa » et enfin un indice précisant la fréquence des lettres dans la langue française. Effectivement, des lettres plus fréquentes sont plus faciles à mémoriser et on les localise plus facilement. Cela est juste une question d'habitude et est indépendant de la position physique de la touche sur le clavier.

L'algorithme a été étendu par rapport à la version initiale du programme. Effectivement, plutôt que d'avoir un indice statique concernant la position de chaque lettre, nous avons jugé plus intéressant et plus réaliste d'évaluer la complexité d'un mot en fonction de la distance entre deux lettres sur le clavier, d'où la nécessité de disposer des indices de position verticale et horizontale de chaque touche. Ainsi, au lieu d'évaluer lettre par lettre une complexité prédéfinie, nous évaluons la complexité en fonction de la distance séparant une lettre et la suivante du mot sur le clavier.

Analyse d'un mot

L'analyse d'un mot est décomposée en plusieurs parties :

- Recherche de lettres identiques contiguës
- Recherche de groupes de 2 lettres identiques et contigus
- Evaluation de la distance entre deux lettres contiguës
- Application de l'indice de présence

Chacune de ces sous-routines est appelée par la fonction Complexite_Mot qui est elle-même exportée vers d'autres modules. Il est nécessaire de lui transmettre un clavier valide pour pouvoir fonctionner.

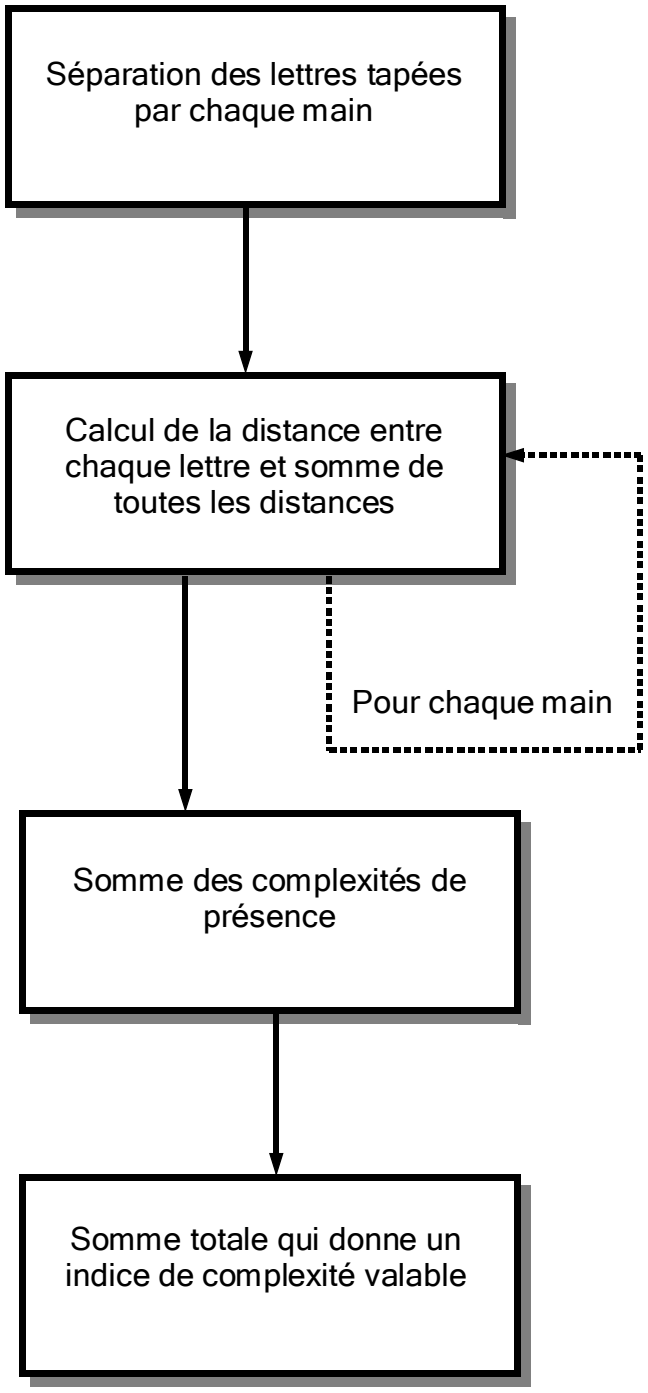
Calcul de la distance entre deux lettres

Le calcul de la distance entre deux lettres est particulièrement simple. Etant donné que l'on connaît la position horizontale et verticale de chacune des lettres, on utilise la formule suivante (Abs représente la valeur absolue) :

```
Distance = 2* Abs(PosY(2) – PosY(1)) + Abs(PosX(2)-PosX(1))
```

On pondère les changements verticaux qui sont plus longs que les changements horizontaux, en pratique.

Fonctionnement du calcul de complexité



Exemple théorique

Prenons le mot AUTEURS. Séparons le mot en deux avec d'un côté les lettres tapées par la main gauche, et de l'autre côté les lettres tapées avec la main droite. On obtient alors les deux chaînes suivantes : ATERS et UU.

Calculons la distance entre chaque lettre de la chaîne ATERS : celle-ci nous donne 10 d'après la table spécifiée plus haut. La chaîne UU présente deux caractères identiques donc la distance est considérée comme nulle. La somme des enchaînements nous donne donc la valeur 10.

Calculons ensuite à partir de la table des indices de présence le « taux de présence » des lettres. Toutes les lettres sont cotées à deux points, donc nous obtenons un indice de 14.

Comme il est préférable de pondérer les enchaînements à la table de présence pour des raisons de complexité réelle, on divise le taux de présence par 4 et on le somme au résultat précédent. Ceci nous donne une complexité finale pour le mot « auteurs » de 13.

Avantages de cet algorithme

La séparation des deux mains est très avantageuse, car elle prend directement en compte les doubles-lettres et les groupes de 2 lettres qui se répètent comme dans le mot « PAPA », à condition bien sûr que les deux lettres qui composent le groupe ne soient pas tapées par la même main, ce qui est le cas ici. Le mot « PAPA » est ainsi décomposé en « PP » et « AA », qui ressortent tous les deux des distances égales à 0. En prenant compte le taux de présence, on arrive à un taux de complexité égal à 2 pour le mot PAPA ce qui se révèle particulièrement exact.

Doubles lettres particulières

Il arrive fréquemment que la main n'ait pas besoin d'être déplacée pour taper trois lettres de suite comme dans la séquence « STS ». Effectivement, les doigts utilisés pour le S et le T ne sont pas les mêmes et la frappe est extrêmement rapide. L'algorithme précédent ne prend pas en charge cette petite particularité, et un ajout a été fait pour le gérer.

Ainsi, en reprenant l'exemple précédent : le programme calcule la distance entre S et T. Celui-ci donne 4. Le programme calcule ensuite la distance entre T et S, logiquement égale aussi à 4. Ici, le petit ajout à l'algorithme se rend compte que deux lettres précédemment la même lettre a été tapée. Comme la distance qui sépare ces deux lettres est assez importante (supérieure à 3), on considère que le même doigt va être utilisé pour retaper la même lettre et on divise alors la distance précédemment trouvée par 2. Ainsi, au lieu d'avoir une complexité de 8, on obtient une complexité de 6, ce qui est plus raisonnable.

Les différentes sous-routines et macros

Dans tous les algorithmes qui suivront, on considèrera les équivalences suivantes pour simplifier l'écriture :

```
Main (lettre : caractère) ⇔ Clavier(asc(lettre), PRM_MAIN)
PosH (lettre : caractère) ⇔ Clavier(asc(lettre), PRM_POSH)
PosV (lettre : caractère) ⇔ Clavier(asc(lettre), PRM_POSV)
CpxF (lettre : caractère) ⇔ Clavier(asc(lettre), PRM_CPXF)
```

La fonction asc renvoie le code ASCII de la lettre spécifiée. La constante NB_MAINS définit le nombre de mains de l'utilisateur.

Le paquetage Complexite ne possède qu'une seule fonction exportée, à savoir la fonction Complexite_Mot. En revanche, d'autres routines sont utilisées en interne :

1) Procedure SepareMains

La procédure SepareMains sert, comme son nom l'indique, à séparer les lettres utilisées par chaque main et à les placer dans des tableaux différents. Elle prend donc en entrée le mot et sa longueur, et renvoie deux tableaux : le premier contient les nouvelles chaînes et le second contient la longueur de ces chaînes :

```
Procedure SepareMains(str : chaîne {E}, ln : entier {E}
                      Mains : tableau(1..NB_MAINS) de chaîne {S}
                      LnMains : tableau(1..NB_MAINS) d'entier {S])

Var i : entier := 1;
    m : entier ;

Début

   ■ On parcourt toute la chaîne

   Tant que (i <= ln) faire

         ■ On récupère la main utilisée par la lettre en cours

         m := Main(str(i)) ;

         ■ On place la lettre dans la main spécifiée

         LnMains (m) := LnMains(m) + 1 ;
         Mains (m)(LnMains(m)) := str(i) ;

         I := i + 1 ;
   Fin Tant Que
Fin
```

2) Fonction Distance

Comme nous l'avons précédemment expliqué, la fonction distance renvoie la distance entre deux lettres. Il faut noter que cette fonction n'exploite en rien la main utilisée par l'utilisateur pour taper les deux lettres. La fonction abs renvoie la valeur absolue de l'argument spécifié.

```
Fonction Distance (l1 : caractère, l2 : caractère) : entier
Début
   Retourne (abs(2*PosY(l2)-PosY(l1)) + abs(PosX(l2)-PosX(l1))) ;
Fin ;
```

3) Fonction CompF

La fonction CompF renvoie la complexité de présence globale pour un mot. Elle effectue simplement la somme de la complexité de présence de chaque lettre.

```
Fonction CompF (str : chaîne, ln : entier) :entier
Var cpx : entier := 0;
Début
   Pour i de 1..ln faire
         Cpx := cpx + cpxF(str(i)) ;
   Fin Pour
   Retourne (Cpx);
Fin;
```

4) Fonction Complexite_Mot

```
Fonction Complexite_Mot (str : chaîne, ln : entier) : entier
Var      Mains : tableau (1..NB_MAINS) de chaîne ;
         LnMains : tableau (1..NB_MAINS) d'entier ;
         Cpx : entier := 0 ; k :entier ;
Début
         Si (ln < 2) alors retourne 0 ;
         SepareMains (str, ln, Mains, LnMains) ;

         Pour i de 1..NB_MAINS
               Pour j de 2..LnMains(i)
                     K := distance(Mains(i)(j-1), Mains(i)(j)) ;
                     Si  j>2 et (Mains(i)(j-2) = Mains(i)(j)) alors
                           Cpx := cpx + (k/2) ;
                     Sinon
                           Cpx := cpx + k ;
                     Fin Si
               Fin Pour
         Fin Pour

         Cpx := Cpx + CompF(str, ln)/4 ;
         Retourne (Cpx) ;
Fin ;
```

Exemple pratique

Vérifions la validité des algorithmes en prenant plusieurs exemples concrets. On considère dans tous les exemples que l'utilisateur n'a que deux mains, ce qui est amplement suffisant pour utiliser ce logiciel. Nous utiliserons le clavier par défaut fourni avec AMC Dactylo dont voici les valeurs :

Position des lettres

	2	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**
3	A	Z	E	R	T	Y	U	I	O	P	
4		Q	S	D	F	G	H	J	K	L	M
5		W	X	C	V	B	N				

Les cases grisées correspondent aux lettres tapées par la main gauche.

Tableau de complexité F par défaut (présence dans la langue française)

2 pts	E	S	A	R	I	N	T	U	L	O
3 pts	M	D	P	C	F	B				
4 pts	V	H	G	J	Q	Z	Y	X	K	W

Exemple n°1 : TOASTS

Testons nos différents algorithmes avec un premier exemple : le mot TOASTS. Examinons le cheminement jusqu'à l'obtention du score définitif.

Passage dans la procédure SepareMains

Str = « TOASTS » et Ln = 6.

- I= 1, m=Main(str(1)) = Main("T") = 1
  LnMains(1)=1 et Mains(1)(LnMains(1))="T"
- I=2, m=Main(str(2)) = Main(« O »)= 2
  LnMains(2)=1 et Mains(2)(LnMains(2))="O"
- I=3, m=Main(str(3)) = Main(« A »)= 1
  LnMains(1)=2 et Mains(1)(LnMains(1))="A"
- I=4, m=Main(str(4)) = Main(« S ») = 1
  LnMains(1)=3 et Mains(1)(LnMains(1))="S"
- I=5, m=Main(str(5)) = Main("T") = 1
  LnMains(1)=4 et Mains(1)(LnMains(1))="T"
- I=6, m=Main(str(6)) = Main(« S ») = 1
  LnMains(1)=5 et Mains(1)(LnMains(1))="S"
- I=7 : on sort de la boucle

A la fin de l'appel, les tableaux renvoyés sont donc les suivants :

i	**1**	**2**
LnMains(i)	5	1
Mains(i)	TASTS	O

Les caractères de la main gauche et les caractères de la main droite ont donc bien été séparés comme prévu.

Passage dans la boucle de la fonction Complexite_Mot

- I=1 J=2 *K=distance(Mains(i)(j-1), Mains(i)(j))*

  Comme Mains(1)(1)="T" et Mains(1)(2)="A", on calcule la distance qui sépare la lettre T de la lettre A sur notre clavier. La fonction Distance renvoie la valeur suivante :
  `2*abs(V(« A »)-V(« T »))+abs(H(« A »)-H(« T »))`, ce qui est égal à 2*0 + 4 soit 4. *On a donc K=4*

  J étant égal à 2, on exécute `cpx :=cpx + k`, d'où *cpx = 4*

- I=1 J=3 *K=distance(Mains(i)(j-1), Mains(i)(j))*

  Comme Mains(1)(2)="A" et Mains(1)(3)="S", on calcule la distance qui sépare la lettre A de la lettre S sur notre clavier. La fonction Distance renvoie la valeur suivante :
  `2*abs(V(« S »)-V(« A »))+abs(H(« S »)-H(« A »))`, ce qui est égal à 2*1 + 2 soit 4. *On a donc K=4*

  J est supérieur à 2, mais Mains(1)(3)=« S » et Mains(1)(1)=« T » donc on exécute `cpx :=cpx + k`, d'où *cpx = 8*

- I=1 J=4 *K=distance(Mains(i)(j-1), Mains(i)(j))*

  Comme Mains(1)(3)="S" et Mains(1)(4)="T", on calcule la distance qui sépare la lettre S de la lettre T sur notre clavier. La fonction Distance renvoie la valeur suivante :
  `2*abs(V(« T »)-V(« S »))+abs(H(« T »)-H(« S »))`, ce qui est égal à 2*1 + 2 soit 4. *On a donc K=4*

  J est supérieur à 2, mais Mains(1)(4)=« T » et Mains(1)(2)=« A » donc on exécute `cpx :=cpx + k`, d'où *cpx = 12*

- I=1 J=5 *K=distance(Mains(i)(j-1), Mains(i)(j))*

  Comme Mains(1)(4)="T" et Mains(1)(5)="S", on calcule la distance qui sépare la lettre S de la lettre T sur notre clavier. La fonction Distance renvoie la valeur 4 comme précédemment. *On a donc K=4*

Par ailleurs, Mains(1)(5)=« S » et Mains(1)(3)=« S ». On a donc une répétition de la lettres « S » dans le mot qui traduit une rapidité accrue de frappe. On exécute alors `cpx :=cpx + (k/2)`, d'où *cpx = 12 + 2 = 14*

- I=2 : Il n'y a pas de boucle J, car LnMots(2) = 1, ce qui est inférieur à la valeur 2, indice primaire de la boucle. On sort alors de la boucle I.

Calcul de la complexité de présence

Il s'agit uniquement de la somme de toutes les complexités CompF de chaque lettre de la chaîne « TOASTS ». D'après le tableau précédent, chaque lettre a un score de 2 points, ce qui fait une complexité F globale de 12 points. Celle-ci est divisée par 4 lors du retour dans la fonction de calcul, et additionnée à la complexité cpx précédemment trouvée. On a donc une complexité finale de cpx := 14 + (12/4) d'où *cpx := 17*

Exemple n°2 : BEBE

Testons maintenant nos algorithmes avec le mot BEBE, mot facile à taper en raison des répétitions de la chaîne « BE ».

Séparation des mains

- I= 1, m=Main(str(1)) = Main("B") = 2
  LnMains(2)=1 et Mains(2)(LnMains(1))="B"
- I=2, m=Main(str(2)) = Main(« E »)= 1
  LnMains(1)=1 et Mains(1)(LnMains(2))="E"
- I=3, m=Main(str(3)) = Main(« B »)= 2
  LnMains(2)=2 et Mains(2)(LnMains(1))="B"
- I=4, m=Main(str(4)) = Main(« E ») = 1
  LnMains(1)=2 et Mains(1)(LnMains(2))="E"

A la fin de l'appel, les tableaux renvoyés sont donc les suivants :

i	**1**	**2**
LnMains(i)	2	2
Mains(i)	EE	BB

Les caractères de la main gauche et les caractères de la main droite ont donc bien été séparés comme prévu. On notera que les deux chaînes comportent des caractères contigus identiques.

Algorithme de distance

Dans les deux appels effectués à la fonction Distance, les deux lettres fournies en paramètres sont identiques. En conséquence, elles possèdent les mêmes coordonnées (h, v) sur le clavier et la fonction renvoie donc une distance de 0, ce qui est parfaitement logique.

La complexité résultante de l'algorithme principal de la fonction est donc *cpx = 0*.

Complexité de présence

La lettre « B » vaut 3 points, et la lettre « E » en vaut 2. L'appel à CompF(str, ln) renvoie donc une complexité F de 10 points. Divisée par 4, elle donne donc 2, étant donné que nous ne travaillons qu'avec des nombres entiers. Finalement, la complexité du mot BEBE est donc *cpx = 2.*

Différents exemples résultants

Mot	Complexité
PAPA	2
BEBE	2
POUR	5
NUIT	7
SCORE	10
TESTS	12

Mot	Complexité
AUTEURS	13
ZOULOU	15
LUGUBRE	17
BOULET	19
TAXES	21
COULOIRS	22

Complexité des algorithmes

Les routines SepareMain et CompF sont toutes les deux basées sur une boucle se finissant avec le nombre de lettres de la chaîne passée en paramètre. Ces deux routines sont donc de complexité n, où n représente le nombre de caractères de la chaîne en paramètre.

La routine Distance ne fait qu'extraire des valeurs d'un tableau. Elle a donc une complexité unitaire.

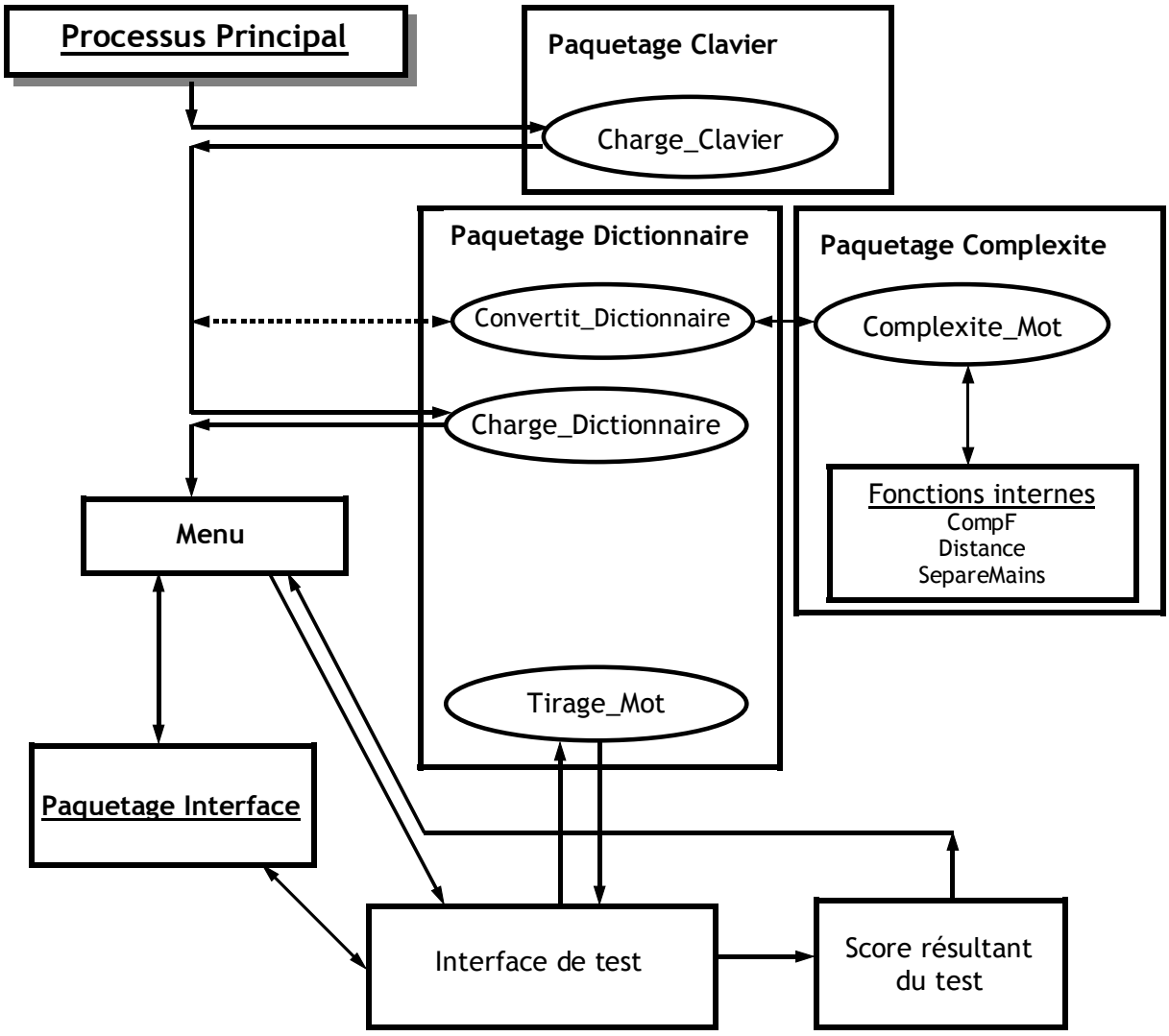
La fonction Complexite_Mot a une complexité de l'ordre de n, où n est le nombre de caractères de la chaîne passée en paramètre. Il y a deux boucles imbriquées, mais la première ne fait qu'un balayage de tableau de longueur fixe égale à NB_MAINS, ce qui n'augmente pas considérablement le temps de calcul.

Quatrième Partie : Processus principal

.

Diagramme de fonctionnement

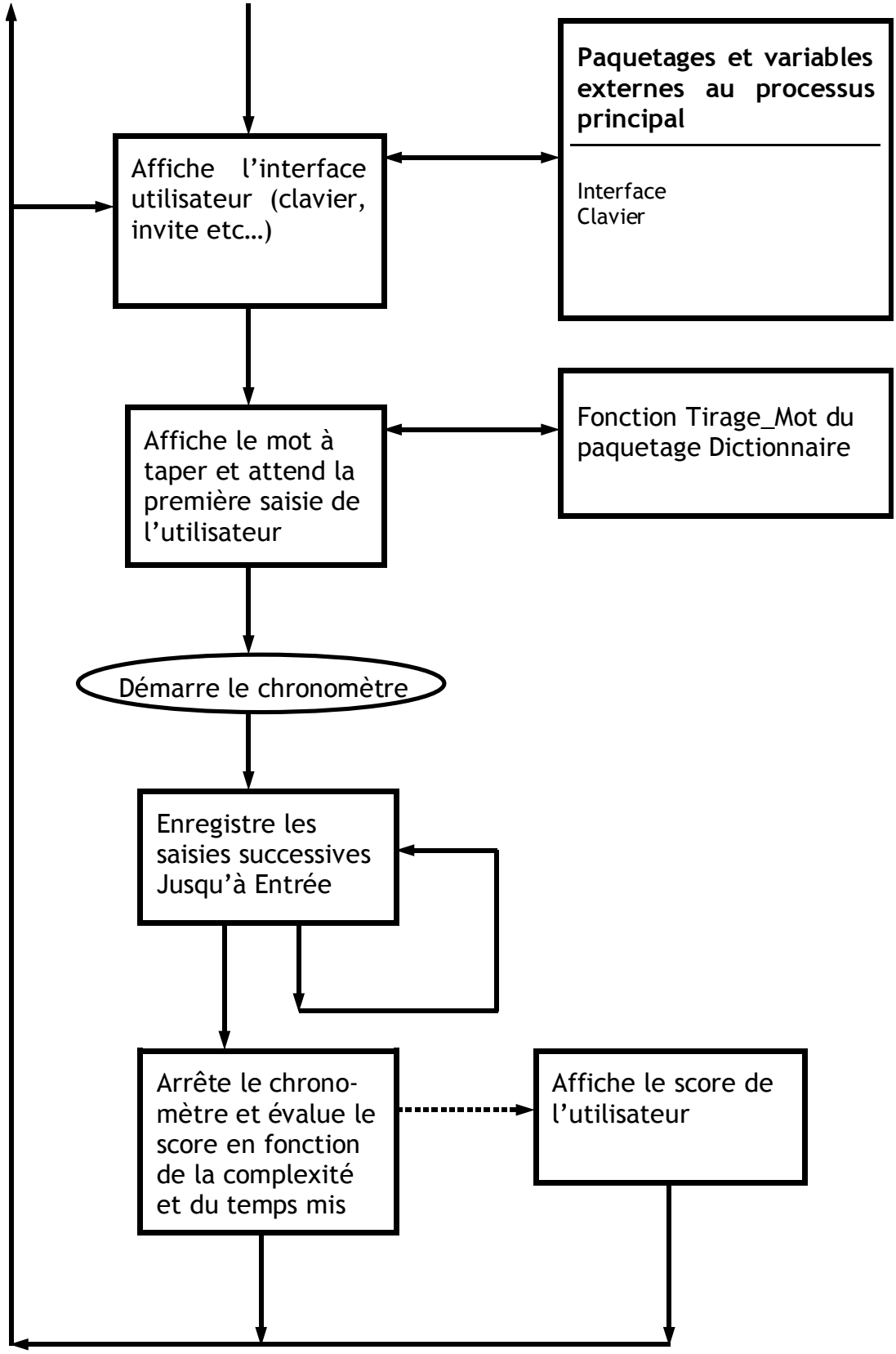
Voici l'organigramme de fonctionnement du programme :



L'interface de test

L'interface de test comprend tout le processus de test de l'utilisateur, à savoir le choix des mots en fonction du niveau de difficulté choisi, l'affichage du clavier à l'écran et des touches colorées, le chronométrage de l'utilisateur et la saisie des données qu'il entre. Vous trouverez page suivante le détail du fonctionnement de l'interface de test.

L'interface de test (diagramme)



Paquetage Interface

Le paquetage Interface regroupe un certains nombres de fonctions qui ne seront pas détaillées ici. Ces fonctions servent à gérer l'écran, à afficher des messages aux endroits désirés, à changer les couleurs du texte, à effacer l'écran etc… Ce paquetage n'est pas détaillé ici car les fonctions qui le composent seront développées au fur et à mesure des besoins, et les algorithmes employés présentent très peu d'intérêt, étant donné l'absence de structures de données et l'utilisation de valeurs arbitraires pour gérer l'écran.

Ligne de commande

Le programme accepte divers arguments en ligne de commande, et notamment la possibilité de choisir son propre dictionnaire, de le convertir, ou de choisir son propre clavier. Il suffit de spécifier le nom du fichier précédé d'un tiret et d'un code de commutation. Ces codes seront détaillés dans le manuel de référence final.

Rappels sur les modes de fonctionnement

Le déroulement du test peut s'effectuer de deux manières différentes : en mode pas-à-pas, préférentiellement utilisé pour les débutants, et un mode continu pour les personnes plus expérimentées. En mode pas-à-pas le programme affiche le score de l'utilisateur pour le mot qu'il vient d'entrer et lui propose éventuellement de recommencer le test avec le même mot. En mode continu, le programme demande à l'utilisateur d'entrer un certain nombre de mots à la suite, et affiche le score final à la fin du test.

Il existe deux modes de calcul de score différent : le mode débutant dans lequel on calcule la progression de l'utilisateur, avec la variation du score entre chaque mot ; et le mode expert dans lequel les performances pures sont prises en compte, à savoir la moyenne des différents scores qui donne un indice cohérent de comparaison.

Rappels sur le choix des mots durant le test

Le choix des mots s'effectue selon deux critères : D'une part, la complexité relative demandée par l'utilisateur. Rappelons que la complexité relative d'un mot est la complexité de ce mot divisée par la taille de ce mot. Celle-ci est divisée en trois niveaux : facile, moyen et difficile. D'autre part, en mode débutant, il est nécessaire de prendre des mots de complexité croissante de manière à ce que l'utilisateur se familiarise avec le clavier. Si l'utilisateur fait trop de fautes sur un niveau de complexité donné, le programme le réduit automatiquement.

Affichage du clavier

Etant donné que l'on connaît la position horizontale et verticale de chaque touche sur le clavier, il est particulièrement aisé d'afficher un clavier virtuel à l'écran.

En mode débutant, les touches devant être frappées sont marquées d'un code de couleur, ce qui permet aux débutants de repérer immédiatement les touches à frapper.

Entrée du mot par l'utilisateur

Le mot est entré lettre après lettre. Si l'utilisateur fait une faute, il a la possibilité de revenir en arrière à l'aide de la touche Delete (Retour Arrière). Ceci est bien sûr comptabilisé dans le score comme une faute de frappe. Lorsque l'utilisateur a fini de taper son mot, il confirme à l'aide de la touche Entrée.

Interface de test (algorithmique)

Voici le codage de l'interface de test en langage algorithmique :

```
■  Tirage du mot et affichage
■  Mot : chaîne ;
■  L : entier ;

Tirage_Mot(Mot, l) ;

■  Attend la première saisie de l'utilisateur
■  Et démarre le chronomètre
■  Le mot tapé est stocké dans le tableau Mot_Entre
■
■  Mot_Entre : Tableau(1..255) de caractères
■  C : caractère ;
■  ch1, ch2 : heure ;
■  i : entier ;

Lire_car(Mot_Entre(1)) ;
Ch1 := Demarre_Chrono ;

■  Enregistre les saisies successives
■  Jusqu'à l'appui sur la touche Entrée

I:=2;
Tant que Mot_Entre(i) /= ASCII.LF     -- ASCII.LF = Entrée
   Lire_Car(Mot_Entre(i)) ;
   I := i+1 ;
   Affiche_Couleur(Mot(i))            -- Affiche la lettre à taper
Fin TantQue

■  Arrête le chronomètre

Ch2 := Arrête_Chrono ;
```

- Evalue le temps mis par l'utilisateur

```
Temps := (Ch2 – Ch1) ;
```

Calcul du score

A la base, le score est le rapport du temps théorique pour taper le mot, et le temps pratique que l'utilisateur a mis. Le temps théorique est évalué en fonction de la complexité du mot : on définit une unité de complexité, par exemple 3 centièmes de seconde par unité de complexité. Le score est ensuite le rapport du temps théorique sur le temps pratique. Ainsi, plus l'utilisateur est rapide, meilleur sera son score.

Exemple : En utilisant la base de temps définie précédemment et considérant un mot d'une complexité évaluée à 28. Le temps théorique sera de 0.84s. Si l'utilisateur réalise un temps de 1.10s sans aucune faute, son score sera de 76 %. Un taux supérieur à 100% indique que l'utilisateur a été plus rapide que le temps théorique.

Supposons que l'utilisateur fasse une faute en tapant le mot. Si celui-ci la corrige, on estime que la perte de temps occasionnée par la correction est une pénalité suffisante. Si le mot n'est pas corrigé, ou si celui-ci ne correspond pas au mot demandé, le score est nul.

L'ensemble des scores réalisés par l'utilisateur sont stockés dans un tableau nommé Results qui est de type Résultat. Dans ce type personnalisé, on stocke le mot, sa taille et le score de l'utilisateur.

```
Type Resultat = Enreg(str :chaîne, ln : entier, score :entier);
Results : Tableau(1..NB_TESTS) de Resultat ;
```

- Avant tout, vérifions que le mot rentré est exact

```
If VerifieMot (Mot, l, MotEntre, i) alors

         o  Calcule le score pour le mot
         o  Et enregistre le résultat dans le tableau Results

         Temps_Theorique := Complexite_Mot(Mot, l) * UNITE_TEMPS ;

         Score := (Temps_Theorique / Temps) ;

         Results(NumMot) := (Mot, l, score) ;

Sinon

         Score := 0 ;
         Results(NumMot) := (Mot, 1, score) ;

Fin Si
```

Fonction VerifieMot

1) Algorithme

Cette fonction prend en entrée le mot exact, sa taille, le mot entré et sa taille et renvoie un booléen pour indiquer si les deux mots sont identiques ou non.

```
Fonction VerifieMot (MotExact : Chaîne, Lexact : Entier,
                     MotEntre : Chaîne, Lentre : Entier) :booléen
Var      i,j :entier ;
Début
         I := Lexact ;
         J := Lentre ;
         Tant que (i >=0 et j>=0) faire

               Si MotEntre(J) := ASCII.DEL alors   --ASCII.DEL= Del
                     J := j –2 ;
               Fin Si ;

               Si MotEntre(i) /= MotEntre(j) alors retourne faux ;
               I := i –1 ; j :=j –1 ;
         Fin Tant Que
         Si (i/=j) alors retourne faux ;
         Retourne vrai ;
Fin ;
```

2) Test de fiabilité

Imaginons que le mot à taper était : SALUT et que l'utilisateur ait tapé SAM, retour arrière, puis LUT. Les tableaux sont alors constitués de cette manière :

	1	**2**	**3**	**4**	**5**	**6**	**7**
MotExact	S	A	L	U	T		
MotEntre	S	A	M	ASCII.DEL	L	U	T

Au depart, les variables sont initialisées au valeurs suivantes :
I = 5 (longueur de MotExact) et j= 7 (longueur de MotEntre)

- Premier passage : MotEntre(J) /= ASCII.DEL et MotEntre(I) = MotEntre(j) = « T » donc on décrémente i et j. I=4 et J=6.

- Second passage : MotEntre(J) /= ASCII.DEL et MotEntre(I) = MotEntre(J)= « U » donc on décrémente i et j. i=3 et j=5.

- Troisième passage : MotEntre(J) /= ASCII.DEL et MotEntre(i) = MotEntre(J)= « L » donc on décrémente i et j. i=2 et j=4.

- Quatrième passage : MotEntre(J)=ASCII.DEL. On a alors j=j-2, ce qui permet d'inhiber la faute de frappe. On a alors i=2 et j=2. On a bien MotEntre(i)=MotEntre(J)= « A » donc on décrémente i et j.

- Dernier passage : i=1 et j=1. MotEntre(J)/= ASCII.DEL et on a bien MotEntre(i)=MotEntre(J)= « S ». On décrémente i et j. Comme ils sont égaux à 0, on sort de la boucle. Les deux mots sont bien équivalents.

Si jamais MotEntre(7) était égal à « R » au lieu de « T », la procédure détecte immédiatement que les MotEntre(7) /= MotExact(5) et on considère que le mot tapé par l'utilisateur est faux.

Evaluation du score global

Mode débutant

On calcule la variation relative entre les différents scores et on effectue la moyenne. On ne prend pas en compte les scores nuls car la variation serait excessive et incorrecte.

```
Fonction ScoreGlobal (Results : Resultat) retourne Entier
   Somme, MotsTestes : entier := 0;
Début

   Pour i de 2..NB_MOTS_TESTES
         Si (Results(i).Score /= 0 et Results(i-1) /=0) alors

            ■ On calcule la variation et on
            ■ L'ajoute à la somme

               Somme:=Somme+(Results(i).Score–Results(i-1).Score);
               MotsTestes := MotsTestes + 1 ;

         Fin Si ;
   Fin Pour

   ■ On retourne la moyenne

   Retourne (Somme / MotsTestes)
Fin ;
```

Mode expert

On calcule la moyenne des scores.

```
Fonction ScoreGlobal (Results : Resultat) retourne Entier
   Somme, MotsTestes : entier := 0;
Début

   Pour i de 1..NB_MOTS_TESTES
         Somme := Somme + Results(i).Score;
   Fin Pour

   Retourne (Somme / NB_MOTS_TESTES)
Fin ;
```

Evolution de la complexité au cours du jeu

En mode débutant, la complexité varie en fonction du score global obtenu. En effet, celui-ci montre la variation du score et donc l'évolution de la prestation de l'utilisateur. Si ce score est positif, cela signifie que l'utilisateur est en progrès et l'on peut donc augmenter de niveau de complexité. Si le score est nul, cela signifie que l'utilisateur n'a pas progressé et qu'il doit continuer à s'entraîner au même niveau de complexité. Si le score est négatif, l'utilisateur n'a pas réussi à maintenir son niveau et on baisse le niveau de complexité pour s'adapter à ses besoins. Cela se traduit très simplement au niveau algorithmique :

```
Si ScoreGlobal > 0 alors
   Cpx := cpx +1 ;
Sinon
   si ScoreGlobal < 0 alors
        Cpx := cpx –1 ;
   Fin  si
Fin si
```