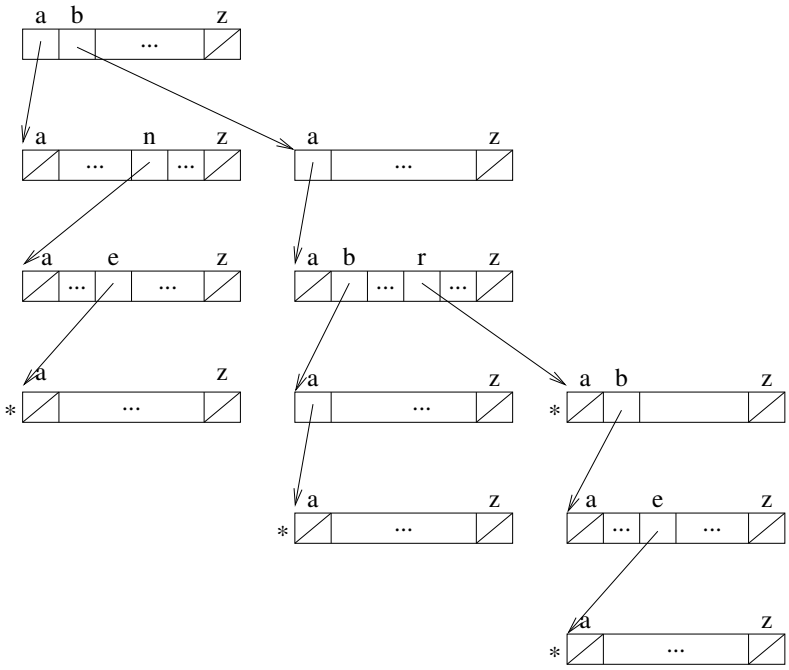
Mini-Projet : un correcteur orthographique

Le but de ce mini-projet est de réaliser un correcteur orthographique. La fonctionnalité est basique : étant donné un texte, vérifier l'orthographe de tous les mots du texte, et reporter les mots mal orthographiés avec une suggestion de correction. Pour simplifier, on supposera tous les mots écrits sur les 26 lettres minuscules de l'alphabet et le texte donné sans ponctuation.

Exercice 1 : Le dictionnaire

La structure que nous allons utiliser pour le dictionnaire est une structure arborescente. Une représentation de la structure contenant les mots 'baba', 'bar', 'barbe' et 'ane' est donnée ci-dessous :



Chaque nœud de la structure contient 26 pointeurs vers des fils. Le nœud n de profondeur p possède un i-ème fils si il existe un mot dans le dictionnaire qui possède la lettre i en position p. Par exemple, il existe un fils de la racine, de profondeur 0, en position 2 parce qu'il existe un mot contenant la lettre b (seconde lettre de l'alphabet) en position 0.

De plus, on marque les nœuds (avec une étoile sur le schéma) si ils correspondent à un mot du dictionnaire. Ainsi les fils de la racine ne sont pas marqués parce que ni 'a' ni 'b' ne sont des mots du dictionnaire. Au contraire, le troisième nœud à une profondeur 3 est marqué parce que 'bar' est dans le dictionnaire.

Q 1.1 Déclarer dans une unité `U_Dico` les types nécessaires à la représentation d'un dictionnaire.

Dans tout le projet, on utilisera un et un seul dictionnaire. Par conséquent, les primitives de manipulation du dictionnaire n'auront pas en paramètre le dictionnaire lui-même. L'initialisation du dictionnaire se fera grâce aux instructions écrites dans la partie `initialization` de l'unité `U_Dico`.

Q 1.2 Ecrire le code de la fonction d'ajout d'un mot dans le dictionnaire :

```
procedure ajouter (mot : MOT);
```

Q 1.3 Ecrire une procédure de construction du dictionnaire à partir d'un fichier :

```
procedure construire (fichier : STRING);
```

Nous supposerons que le dictionnaire est contenu dans un fichier texte, chaque ligne contenant un mot. Vous pourrez utiliser l'unité `U_FichierTexte` fournie pour réaliser la lecture du fichier dictionnaire.

Q 1.4 Afin d'obtenir la liste de tous les mots du dictionnaire, on ajoute la fonctionnalité suivante à notre unité :

```
function tousLesMots : LISTE;
```

Une unité `U_Liste` est fournie, à adapter à vos besoins.

Q 1.5 Afin de vérifier le fonctionnement de l'unité `U_Dico`, écrire un programme `testDico` qui construit le dictionnaire à partir de `dico.txt` (fourni) et qui affiche la liste des mots obtenue avec la fonction `tousLesMots`.

Exercice 2 : La procédure de vérification (version basique)

Afin de vérifier l'orthographe des mots d'un texte, nous allons traiter le texte mot par mot (vous pourrez encore une fois utiliser `U_FichierTexte`).

Pour chaque mot, nous allons comparer ce mot à l'ensemble des mots du dictionnaire. Pour comparer les mots nous nous servirons de la distance d'édition implantée dans un TP précédent.

Si il existe un mot du dictionnaire dont la distance d'édition avec le mot en cours de test est nulle, alors le mot est dans le dictionnaire. Si ce n'est pas le cas, il suffira de conserver le mot dont la distance est la plus petite afin de proposer la correction la plus pertinente.

Q 2.1 En utilisant le code écrit lors du TP sur la distance d'édition, créer une unité `U_Edition` qui contient l'unique fonction :

```
function distance_dynamique (u, v : STRING) : CARDINAL;
```

Q 2.2 Ecrire une procédure qui, étant donné un mot à tester et une liste de mots, indique si le mot est dans la liste et sinon donne le mot le plus proche au sens de la distance d'édition :

```
procedure verifier(const m : MOT; const l : LISTE; out estPresent; out motLePlusProche);
```

Cette procédure prendra place dans le programme principal.

Q 2.3 Ecrire un programme principal `verificateurBasique.pas` qui effectue la vérification d'orthographe d'un texte contenu dans un fichier dont le nom est passé en paramètre[1]. Le programme produira une sortie de ce style :

```
dans -> ane
un -> une
de -> une
trois -> zut
jours -> zut
un -> une
```

Exercice 3 : La procédure de vérification (version évoluée)

L'inconvénient de la méthode proposée ci-dessus est qu'elle ne tire pas partie de la structure arborescente du dictionnaire. Plutôt que d'obtenir la liste des mots du dictionnaire, on préférerait sauter de nœud en nœud dans le dictionnaire pour parcourir tous les mots.

Cela revient à réaliser un parcours en profondeur de l'arbre. Nous rappelons qu'un tel parcours se fait naturellement avec une pile. La différence par rapport à ce que vous avez pu écrire pour les parcours d'arbre est qu'il faut interrompre le parcours lorsqu'on trouve un mot, puis le reprendre lorsqu'on veut passer au mot suivant. Cela paraît compliqué mais il suffit en fait de laisser la pile dans son état lorsqu'on trouve un mot, puis de recommencer avec cette même pile pour obtenir le mot suivant. C'est pourquoi on utilisera une et une seule pile, déclarée dans l'unité `U_Dico`. Le fonctionnement est très proche de celui d'un itérateur : initialisation (voir `initialiserParcours`), passage au suivant (voir `motSuivant`), test de fin de parcours (réalisé ici grâce à l'eception levée par `motSuivant`). Une unité `U_Pile` est fournie, à adapter à vos besoins.

Q 3.1 La pile va donc servir à empiler les nœuds du dictionnaire lors du parcours, mais pour obtenir le mot associé à un nœud, c'est-à-dire la concaténation des lettres parcourues pour aller de la racine à ce nœud, il faut s'en souvenir, et donc l'empiler également. Proposer une déclaration de type pour les éléments de la pile.

Q 3.2 Afin de pouvoir faire plusieurs parcours du dictionnaire, créer une procédure dans `U_Dico` qui initialise la pile pour le parcours :

```
procedure initialiserParcours;
```

Q 3.3 Ecrire une fonction dans `U_Dico` permettant d'accéder au mot suivant dans le dictionnaire. Cette procédure déclenche une exception lorsqu'on a parcouru tous les mots du dictionnaire.

```
function motSuivant : MOT;
```

[1] On rappelle qu'on accède au premier paramètre avec `paramStr(1)`.

Q 3.4 Ecrire un programme principal `verificateurEvolue.pas` qui effectue la vérification d'orthographe d'un texte contenu dans un fichier dont le nom est passé en paramètre en utilisant le parcours d'arbre. Le programme produira une sortie de ce style :

```
dans -> ane
un -> une
de -> ane
trois -> ane
jours -> bar
un -> une
```

A rendre

Une archive avec toutes les unités et les trois programmes de test ainsi que les fichiers exemples. On doit pouvoir décomprimer l'archive, compiler et tester sans avoir à ajouter ou modifier de fichier. Comme d'habitude l'archive portera le nom du binôme et les fichiers seront contenus dans un répertoire portant le nom du binôme.