MASTER EN SCIENCES INFORMATIQUES

Traduction automatisée de Répertoires symptomatiques pour la Médecine Hornéopathique

Hogne, Jean-Pierre

Award date:
1986

Awarding institution:
Universite de Namur

Traduction automatisée de Répertoires symptômatiques pour la Médecine Homéopathique

Jean-Pierre Hogne.

Mémoire réalisé sous la
direction du Professeur J. Fichefet
en vue de l'obtention du titre de
Licencié et Maître en Informatique.

Namur — Septembre 1986.

Remerciements.

Je tiens avant tout à remercier Maryline Salmon, qui a effectué l'ensemble des recherches grammaticales et lexicales afférentes à ce mémoire. Aucune difficulté n'a pu venir à bout de sa bonne humeur et de ses compétences, et bon nombre des résultats présentés dans la suite sont issus de ses travaux et de ses conseils judicieux.

Je dois ensuite remercier l'ensemble des membres de la Communauté Espérantiste, et tout particulièrement M^rs^ Pierre Anselme et Germain Pirlot, pour leur disponibilité, leur enthousiasme et leur aide désintéressée.

Merci également à Jacques Paris, qui fut disponible à tout instant et me permit au tout début d'esquisser les contours du système à construire.

Merci aussi à M^rs^ Leroy et Barreto, de l'Institut d'Informatique, à Maria van de Casteele, à M^rs^ Deville, Ostyn et Hantson, de la Faculté de Philosophie et Lettres, ainsi qu'à toutes les autres personnes dont les conseils ont contribué à la réalisation de ce mémoire.

Pour terminer, il me reste à remercier M^r^ J. Fichefet, promoteur de ce mémoire, pour l'intérêt du sujet qu'il m'a proposé ainsi que pour ses interventions discrètes et efficaces.

Plan du Mémoire.

Première partie : Préliminaires.

1. Introduction générale.
   1.1. Homéopathie et répertorisation.
   1.2. Le système RADAR.
   1.3. Traduction des répertoires.
   1.4. Un sous-produit utile : une version des répertoires accessible sémantiquement.

2. Etude préliminaire.
   2.1. Informatique et traduction.
      2.1.1. Situation actuelle.
         2.1.1.1. Traitements de textes avancés.
         2.1.1.2. Bases de données terminologiques.
         2.1.1.3. Autres programmes utiles.
         2.1.1.4. Systèmes de traduction "automatique".
      2.1.2. Les techniques de traduction les plus utilisées.
         2.1.2.1. Traduction par transfert de représentation.
         2.1.2.2. La traduction via un interlangage.
      2.1.3. La traduction avec l'Esperanto comme interlangage.
   2.2. Choix d'une technique de traduction.

3. Première analyse du système à construire.
   3.1. Etude du format des répertoires.
      3.1.1. Le répertoire de Kent.
         3.1.1.1. Contenu du répertoire.
         3.1.1.2. Support du répertoire.
         3.1.1.3. Encodage du répertoire dans le système RADAR.
      3.1.2. Le répertoire de Barthel & Klunker.
      3.1.3. Synthèse : structure générale d'un répertoire.
   3.2. Définition précise de la notion de traduction d'un répertoire.

Première partie :

Préliminaires.

1. Introduction générale.

Ce chapitre présente brièvement le cadre dans lequel s'insère ce mémoire. Il en précise ensuite le but, et les résultats escomptés.

1.1. Homéopathie et répertorisation.[Jacques]

La démarche homéopathique se base sur la loi de la Similitude, selon laquelle toute substance capable de provoquer une série de symptômes chez un homme sain est également capable de guérir un homme malade qui présente ce même ensemble de symptômes.

Tout l'art du praticien consiste donc à dresser un portrait détaillé de son patient et des symptômes qu'il présente, afin de découvrir la substance qui, administrée à un homme sain, le ferait ressembler à ce portrait.

Les résultats de l'expérimentation des différentes substances sur l'homme sain, ainsi que les observations cliniques faites lors de l'administration de chaque remède, sont consignés dans de volumineux "catalogues", qui constituent ce que l'on appelle la Matière Médicale Homéopathique [Hering] [Allen].

Etant donné que le remède, et non le symptôme, est la seule clé d'accès à cette Matière Médicale, on conçoit que très tôt, divers auteurs se soient attachés à organiser ces données de manière plus pratique sous forme de répertoires de symptômes, dont ceux de Kent [Kent] (paru dès 1897) et de Barthel & Klunker [Barthel].

Ces répertoires sont divisés en chapitres[1]. Chaque chapitre contient une série de symptômes, assortis de précisions quant à leur fréquence, le moment où ils se manifestent, etc..., ainsi que des différents remèdes qui peuvent être prescrits. Les énoncés de ces symptômes sont rédigés selon une syntaxe rigide et simplifiée, mettant en évidence le(s) mot(s)-clé(s) du symptôme. Ils sont classés par ordre alphabétique.

---

1 Par exemple : Head, Extremities, Vertigo,...

Les deux répertoires couvrent environ 60.000 symptômes et 1.754 remèdes. Celui de Kent est rédigé en langue anglaise, et celui de Barthel & Klunker, bien qu'organisé selon le texte anglais, est assorti de "sous-titres" en français et en allemand.

1.2. Le système RADAR. [Fich1][Fich2]

Développé depuis plusieurs années par l'ASBL ARCHIMEDE[2], ce système a pour but de faciliter l'accès aux répertoires de Kent et de Barthel & Klunker, et d'aider au diagnostic remédial.

Un interface très souple permet au médecin de parcourir le répertoire de diverses manières (accès séquentiel, par rubrique, par page, par clé), et de sélectionner un certain nombre de symptômes présentés par le patient.

Des outils d'aide à la décision permettent alors de déterminer une liste des remèdes susceptibles de guérir le patient.

RADAR a d'abord été développé sur VAX, et une version tournant sur Wang PC est actuellement commercialisée. Grâce à une programmation rigoureusement modulaire, les problèmes de conversion d'un environnement à l'autre sont réduits au strict minimum, au point que le développement du programme peut se poursuivre sur VAX et que les modifications se transfèrent de l'un à l'autre sans encombre.

1.3. Traduction des répertoires.

La pleine utilisation de RADAR est freinée du fait que le répertoire de Kent est unilingue et celui de Barthel & Klunker traduit uniquement en français et en allemand.

Les homéopathes ne possèdent pas tous la langue anglaise au point de saisir parfaitement les nuances, parfois subtiles, qui émaillent les énoncés des symptômes. Dès

---

[2] Association pour la ReCHerche en Informatique MEDicalE.

lors, leur désir de disposer chacun d'une traduction des répertoires dans leur langue nationale apparaît comme tout-à-fait légitime.

Le but du travail entrepris dans le cadre de ce mémoire est de "sous-titrer" le texte de ces deux répertoires :

- d'abord en français et en allemand pour le Kent,
- ensuite dans d'autres langues européennes comme le néerlandais et l'espagnol, cette fois pour les deux répertoires.

Il s'agit bien de sous-titrer, et non de traduire et réorganiser complètement les répertoires selon la nouvelle langue. En effet, la littérature homéopathique fait souvent référence à ces ouvrages de manière très précise, du fait que leur présentation n'a jamais été modifiée depuis leur première édition, ne fût-ce qu'au niveau de la numérotation des pages.

1.4. Un sous-produit utile : une version des répertoires accessible sémantiquement.

Il est raisonnable de penser que quelque part, dans le processus de traduction, le système devra accéder au sens des phrases qu'il traduit, sous peine de ne pouvoir livrer qu'un résultat boiteux, juxtaposition de mots traduits l'un après l'autre sans considération sémantique.

Dans cette perspective, il serait intéressant de conserver une représentation de ce contenu sémantique, qui pourrait être exploitée dans la suite par le système RADAR.

Ainsi, l'accès par clé à un symptôme oblige l'utilisateur à connaître exactement les termes composant l'énoncé de ce symptôme dans le répertoire. Or, rappelons que la base de données contient environ 60000 symptômes différents... Une indexation sur le contenu sémantique des répertoires faciliterait donc considérablement l'accès à ceux-ci.

On tiendra compte de ces considérations dans les points suivants lorsqu'il s'agira de choisir une technique de traduction pour ce système.

2. Etude préliminaire.

Ce chapitre présente brièvement diverses tendances actuelles en matière de traduction automatique, ainsi que les techniques mises en oeuvre par les systèmes existants. Il présente ensuite la technique que nous avons adoptée.

2.1. Informatique et traduction.

2.1.1. Situation actuelle.

Il est bon de se rappeler que la traduction est avant tout l'affaire des traducteurs professionnels. Cependant, l'utilisation de l'informatique peut alléger un certain nombre d'aspects fastidieux et peu intéressants de leur travail. En voici quelques exemples:

2.1.1.1. Traitements de textes avancés.

Un traitement de texte peut être utile aux traducteurs à condition d'être adapté à leurs exigences particulières : écran suffisamment large, facilité de manipulation simultanée de plusieurs textes, fonctions puissantes d'usage facile, jeu de caractères étendu immédiatement accessible, ...

De toute façon, le traitement de textes est un préalable obligé si on veut utiliser d'autres outils informatiques.

2.1.1.2. Bases de données terminologiques.

Un des aspects les plus lourds de la traduction est la recherche de la traduction exacte d'un terme (souvent fortement dépendante du contexte). Si le traducteur n'a pas le bon dictionnaire sous la main, la recherche peut prendre du temps. Un programme donnant accès à une base de données terminologique suffisamment riche peut être très utile.

Un exemple de système de ce type est Eurodicautom, un programme développé par les Communautés Européennes depuis 1969, et utilisé tous les jours par leurs services de traduction [Eurodic].

2.1.1.3. Autres programmes utiles.

Un certain nombre d'autres programmes peuvent aider le traducteur, allant du "simple" vérificateur orthographique de textes, jusqu'au programme capable de signaler qu'une phrase est d'une complexité grammaticale trop importante par rapport au niveau de complexité moyen des phrases habituellement rencontrées dans un certain type de textes.

2.1.1.4. Systèmes de traduction "automatique".

Ces systèmes sont capables de traduire eux-mêmes les textes qu'on leur propose. On les détaillera au point suivant, mais il faut préciser ici pourquoi on les considère comme des systèmes d'aide à la traduction, et pas comme des systèmes traducteurs au sens habituel du terme.

En fait, à l'heure actuelle - et après plus de trente ans de recherche dans ce domaine - aucun système construit pour travailler sur des textes quelconques n'est capable de livrer un produit fini fiable, exempt d'erreurs de traduction.

A titre d'exemple, les Communautés Européennes disposent d'un système de traduction appelé Systran [Systran]. Pour la période du premier avril 1981 au 31 octobre 1982, il n'a été utilisé que pour 2,7% du nombre total des traductions réalisées. De l'avis (ou l'aveu) d'un des responsables, *"après des années de mise au point, la qualité est loin d'être suffisante pour une publication immédiate. Le texte contient encore des erreurs grossières donnant lieu à des non-sens ou à des contresens. Ces erreurs doivent être corrigées par le traducteur, dont la fonction demeure indispensable"*. [Lavorel]

La seule procédure opérationnelle actuellement consiste à faire pré-traduire le texte par le programme, et à le faire corriger (*post-éditer*) par un traducteur humain. Si cette procédure est un échec pour les informaticiens, elle a cependant le mérite de simplifier la tâche des traducteurs en les débarrassant de toute une série de phrases triviales dont la traduction est évidente, mais prend du temps manuellement. [Eurodic] [Systran] [Melby] [Lavorel].

2.1.2. Les techniques de traduction les plus utilisées. [Tucker] [Witkam]

On peut répartir ces techniques en deux grandes familles : la traduction par transfert de représentation et la traduction via un interlangage.

2.1.2.1. Traduction par transfert de représentation.

Les systèmes utilisant cette technique analysent le texte-source pour en découvrir la structure profonde, tant syntaxique que sémantique. Ils élaborent une représentation interne sophistiquée de cette structure, puis procèdent à un transfert de représentation pour obtenir une représentation interne du texte-cible, qui est alors généré dans la langue-cible (figure 2.1).

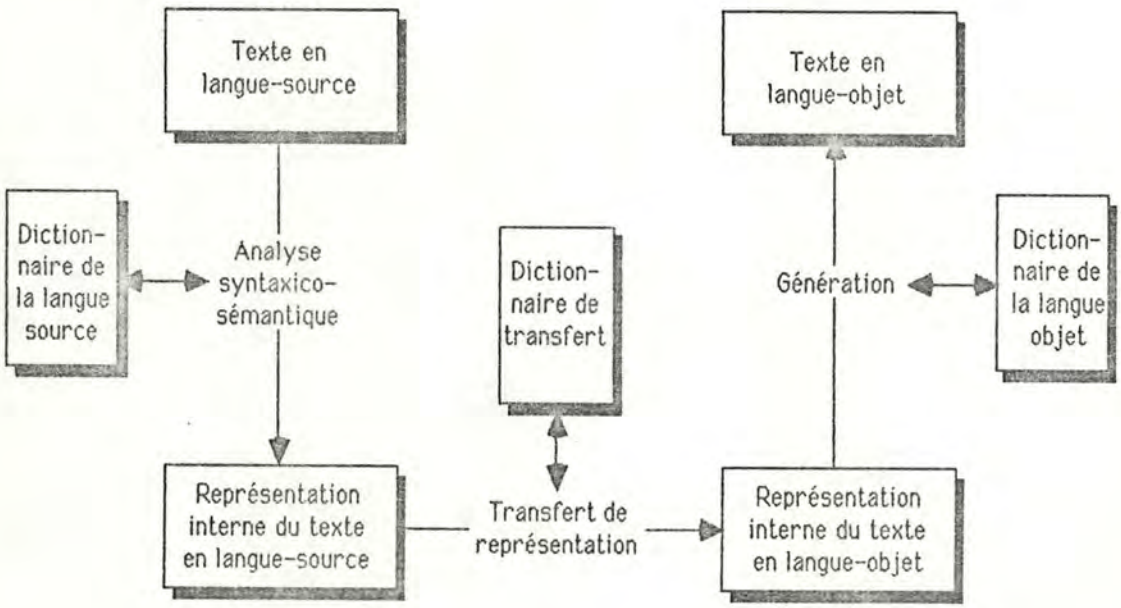


Figure 2.1 : Traduction par transfert de représentation.

La traduction est ici exclusivement bilingue, et si on veut traduire des textes dans **m** langues-cibles à partir de **n** langues-sources, il faut **n*m** composants de transfert de représentations internes, ainsi que **n** analyseurs, et **m** générateurs.

Pour faciliter l'implémentation d'applications multilingues, on doit donc simplifier au maximum la tâche des composants de transfert, en reportant la complexité sur les composants d'analyse et de génération.

2.1.2.2. La traduction via un interlangage.

Le principe est ici de générer, lors de l'analyse du texte-source, une représentation interne dans un langage intermédiaire universel, qui exprime le sens du texte et les dépendances entre ses divers composants. Cette représentation sert alors de base à la génération directe du texte-cible (figure 2.2).

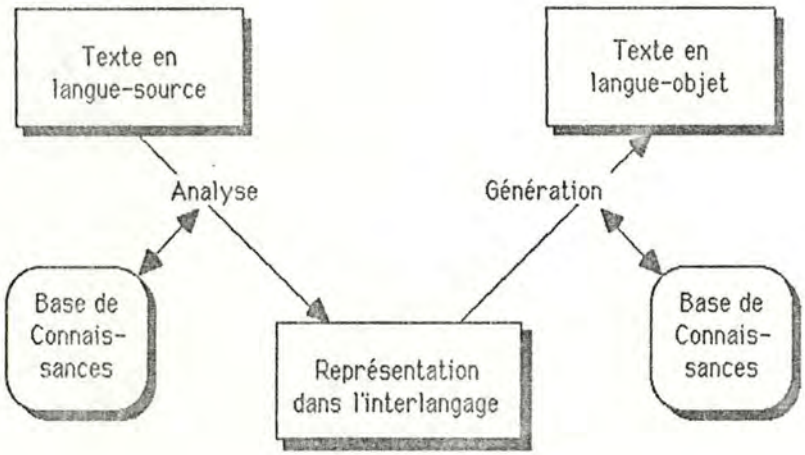


Figure 2.2 : la traduction via un interlangage.

Ces systèmes sont plus adaptés à la traduction multilingue : une fois la représentation interne élaborée, il "ne reste plus" qu'à en tirer un équivalent dans la langue-cible. Il suffit alors d'implémenter, pour **n** langues-sources et **m** langues-cibles, **n** analyseurs et **m** générateurs.

Il semble de plus en plus que la capacité représentative d'un interlangage soit strictement liée à la taille de l'univers du discours envisagé : plus cet univers est vaste, plus il devient difficile de créer un interlangage valable.

2.1.3. La traduction avec l'Esperanto comme interlangage.

Ce point présente une technique de traduction qui, si elle se rapproche des techniques de traduction via un interlangage, se distingue cependant résolument de toutes les

techniques existantes par un certain nombre d'aspects, dont les principaux sont les suivants :

- L'interlangage utilisé est une langue "presque naturelle" existante, et non un langage formel complexe et créé de toutes pièces pour la circonstance;

- La philosophie de traduction rompt définitivement avec le principe de "post-édition" des textes traduits qui règne actuellement en la matière.

Cette nouvelle approche originale a été proposée en 1983 par le Buro voor Systeemontwikkeling (Pays-Bas), à l'occasion du démarrage d'un vaste projet concernant un système de traduction automatique intégrable dans des réseaux de stations de travail (à l'horizon 1990) [Witkam].

Tout utilisateur de ce système pourrait entrer un texte au clavier de sa station, et l'envoyer aux autres personnes connectées sur le réseau, chacune d'elles recevant le texte traduit dans sa propre langue.

Le texte entré serait traduit en Esperanto au fur et à mesure de la frappe, et si nécessaire, le système pourrait lancer un dialogue avec l'utilisateur pour éclaircir les différentes ambiguïtés rencontrées lors de cette traduction. Une fois disponible une traduction esperanto dénuée d'ambiguïtés, c'est cette version du texte qui, après compactage, serait distribuée sur le réseau pour être traduite par la station de travail de chaque destinataire dans sa propre langue. Les erreurs potentielles dues aux ambiguïtés ayant été corrigées à la source, les traductions obtenues seraient d'une qualité suffisante que pour éviter la "post-édition".

Cette approche soutient que l'Esperanto, pour peu qu'on lui apporte quelques modifications destinées à débarrasser les phrases d'ambiguïtés syntaxiques, peut être considéré comme un réel interlangage, au même titre que les autres modes de représentation habituellement utilisés (réseaux sémantiques, arbres étiquetés syntaxiquement et sémantiquement, etc... [Rich] [Sowa]).

En effet, les mots Esperanto sont composés de racines, exprimant leur sens, et d'affixes qui viennent préciser ce sens. Ces affixes, invariables quant à leur signification, fournissent un jeu d'"étiquettes sémantiques" tout trouvé, de même nature que, par exemple, les jeux de "primitives conceptuelles" proposés par Schank [Schank]. De plus, sa structure grammaticale est absolument régulière, et le nombre de règles est restreint à 16. L'analyse d'une phrase est ainsi grandement facilitée.

De plus, la représentation intermédiaire d'un texte en Esperanto est directement accessible via un éditeur classique, contrairement à une représentation complexe qui nécessite des procédures d'accès spécialisées.

2.2. Choix d'une technique de traduction.

Après ce rapide tour d'horizon des techniques de traduction automatique, il ne reste plus qu'à en choisir une pour construire notre système.

Les répertoires devant être traduits dans plusieurs langues, il est clair qu'on retiendra l'approche interlinguale. Reste à décider quel interlangage utiliser.

Les interlangages "classiques" ont l'avantage d'avoir déjà été utilisés, testés, mis au point par d'autres. La littérature ne manque pas d'informations à leur sujet. Cependant, nous avons vu qu'actuellement, aucun système "classique" n'a de résultats satisfaisants.

L'idée d'utiliser l'Esperanto est nouvelle, et à part l'étude de faisabilité de BSO [Witkam], aucun document n'existe à son sujet. Bien plus, elle est déjà décriée par des auteurs [Tucker] et des traducteurs professionnels, qui lui reprochent d'exister, sans fournir d'autres arguments que le danger de sortir des sentiers battus.

Cette idée est pourtant séduisante, et à plus d'un titre :

- D'abord, justement, elle sort des sentiers battus, et propose une approche de la traduction assistée par ordinateur à laquelle personne jusqu'ici n'avait pensé.

- Ensuite, elle est proposée par un groupe de personnes dynamiques, à l'esprit ouvert et optimiste, résolument tournés vers l'avenir. La qualité de l'étude de faisabilité réalisée par BSO est révélatrice à cet égard [Witkam].

- De plus, par sa nature même, cette idée permet d'utiliser les ressources et les qualités de la Communauté esperantiste, dont les membres forment un véritable réseau mondial où naissent et se transmettent une quantité impressionnante d'informations utiles, tant du point de vue informatique, que du point de vue linguistique et même homéopathique.

De plus, cette approche de la traduction est avantageuse dans le cas qui nous préoccupe :

- Elle semble apte à autoriser une traduction de bonne qualité de nos répertoires, dont la syntaxe n'est pas trop compliquée, et qui contiennent surtout un grand nombre de phrases stéréotypées dont la traduction pourrait être faite une fois pour toutes avec l'aide du traducteur, puis confiée à la machine.

- Elle produira une version des répertoires en Esperanto, qui constituera une "représentation accessible sémantiquement" toute trouvée, pour peu que l'on prenne la précaution de séparer les différentes racines qui constituent les mots.

- Vu la simplicité de l'Esperanto par rapport à d'autres modes de représentation, la mise en oeuvre du système sera facilitée, puisque les résultats intermédiaires de la traduction seront simplement des fichiers de texte, accessibles immédiatement tant par le linguiste que par l'informaticien (s'il connaît l'Esperanto).

Pour toutes ces raisons, on adoptera les principes proposés par BSO :

- Traduction en Esperanto du texte-source, via un programme fortement interactif qui utilise les compétences du traducteur humain autant de fois que nécessaire, pour livrer une version du texte dénuée d'ambiguïtés syntaxiques.

- Traduction de l'Esperanto vers d'autres langues, via une série de programmes les moins interactifs possibles, et à la limite complètement indépendants du traducteur humain.

Le processus général de traduction sera le même dans les deux cas : sur base d'un lexique suffisamment complet, le programme essaiera de traduire lui-même chaque phrase qu'il rencontrera, en recourant à l'utilisateur autant de fois que nécessaire. Ce dernier, à l'aide d'un mini-traitement de textes, pourra apporter toutes les corrections qu'il désirera à la traduction proposée, avant de la valider.

Au fur et à mesure, le système devra être capable de tenir une trace de ses raisonnements pour proposer une mise à jour de la grammaire et/ou du lexique lorsque cela sera utile. Par exemple, les règles de grammaire systématiquement refusées par l'utilisateur devront pouvoir être mises en évidence; les petites phrases du genre "reading, while" ou "bed, in" qui apparaissent des centaines de fois dans les répertoires devront pouvoir faire l'objet de règles spécifiques afin d'accélérer le traitement, etc...

Ainsi, de façon incrémentale, on peut espérer aboutir à une traduction correcte des répertoires, tout en allègeant au maximum la tâche du traducteur qui ne devra plus se concentrer que sur des phrases non triviales, laissant à la machine le soin de s'occuper des autres.

3. Première analyse du système à construire.

Le but de ce chapitre est d'aboutir à une première décomposition du système en phases homogènes, ainsi qu'à une brève spécification de chacune de ces phases.

Sur base de ces informations, il faudra ensuite décider quelles phases pourront être programmées dans le cadre de ce travail, en fonction des diverses contraintes existantes.

3.1. Etude du format des répertoires.

Il nous semble utile, avant d'aller plus loin, d'étudier précisément la structure des répertoires que le système à construire devra traduire. Nous détaillerons d'abord le répertoire de Kent, puis celui de Barthel & Klunker.

3.1.1. Le répertoire de Kent.

3.1.1.1. Contenu du répertoire.

Fondamentalement, le répertoire de Kent est une collection d'énoncés de symptômes, structuré selon une hiérarchie de mots-clés.

Tous les énoncés contenant un même mot-clé principal sont regroupés dans un même chapitre (Mind, Vertigo, Head, Eye, etc...). A l'intérieur de chaque chapitre, tous les énoncés contenant un même mot-clé secondaire sont regroupés dans une même rubrique (inflammation, intoxication, pain, weakness, etc...). A l'intérieur de chaque rubrique, tous les énoncés contenant un même mot-clé sont regroupés dans une même sous-rubrique (morning, afternoon, in bed, etc...).

Reportant ce processus autant de fois que nécessaire, on obtient une structure arborescente (voir figure 3.1) dont la racine unique est le répertoire, et dans laquelle chaque branche distincte joignant cette racine à un sommet terminal contient l'énoncé

complet d'un symptôme : partant de la racine, et collectant les énoncés contenus dans les noeuds successifs d'une même branche, on élabore peu à peu cet énoncé complet.

Rien n'empêche d'ailleurs d'arrêter le parcours de la branche à un noeud quelconque de celle-ci : on obtient ainsi un énoncé partiel, qui peut se suffire à lui-même. Les noeuds inférieurs peuvent alors être considérés comme contenant des précisions quant à ce symptôme.

A chaque noeud de l'arborescence, peut être rattachée une liste de remèdes. Cette liste se rapporte au symptôme dont l'énoncé est contenu dans les noeuds allant de la racine au noeud considéré.

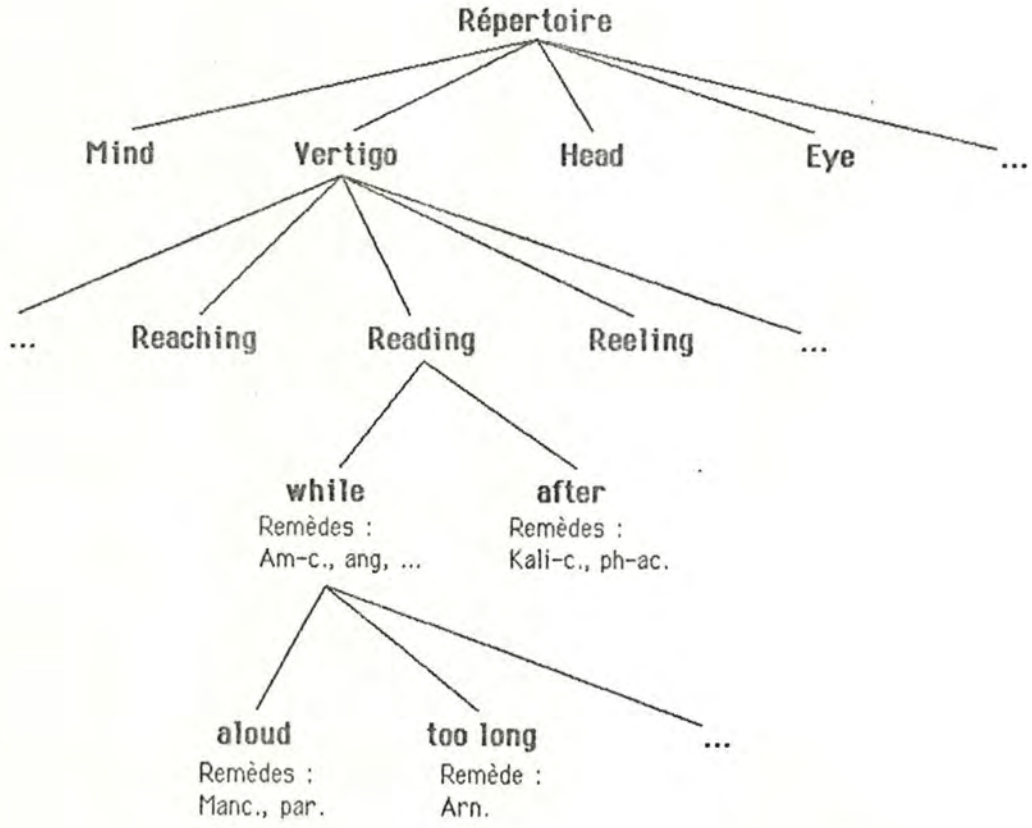


Figure 3.1 : Structure d'un répertoire : exemple tiré du Kent.

3.1.1.2. Support du répertoire.

Le support initial du répertoire était un livre, dans lequel on peut considérer que Kent a écrit page après page le contenu de l'arborescence.

L'ordre de transcription dans le livre correspond à une visite "en profondeur d'abord" des noeuds de l'arborescence, avec transcription au fur et à mesure de l'énoncé et de la liste de symptômes éventuelle de chaque nouveau noeud visité. Pour faciliter la lecture, les énoncés de niveau inférieur sont décalés vers la droite.

L'ordre dans lequel les descendants immédiats d'un noeud sont visités est normalement l'ordre alphabétique de l'énoncé qu'ils contiennent. Cependant, certains énoncés comme les circonstances temporelles, la localisation dans le corps, etc..., peuvent être visités avant les autres pour des raisons de commodité propres aux utilisateurs.

Le support étant un livre, il faut régulièrement passer à la page suivante. Si cette opération a lieu lors de la transcription des sous-arbres rattachés à un noeud qui en possède beaucoup, un mot-clé identifiant ce noeud peut être répété en début de page, afin de dispenser le lecteur de feuilleter les pages précédentes pour voir où il se trouve.

3.1.1.3. Encodage du répertoire dans le système RADAR.

Le répertoire a été encodé tel qu'il se présentait, c'est-à-dire que l'on a transcrit dans l'ordre du livre les différents symptômes et leurs listes de remèdes éventuelles. Chaque symptôme a été précédé d'un nombre identifiant son niveau dans la hiérarchie[3]. A chaque nouvelle page, le numéro de celle-ci a été transcrit, précédé d'un symbole caractéristique. Les répétitions éventuelles ont été recopiées de la même façon.

On se trouve ainsi devant un fichier au contenu hybride, superposition d'une structure purement séquentielle propre au support livresque (n^{os} de page, répétitions), et d'une structure arborescente propre au contenu du Kent (symptômes, remèdes).

---

3 Tout descendant d'un noeud possède un numéro de niveau immédiatement supérieur à celui de son antécédent.

La structure est encore compliquée par la présence de "rubriques doubles", juxtaposition sur une même ligne de deux énoncés appartenant à des niveaux différents de l'arborescence. Ceci arrive lorsqu'un symptôme partiel n'a pas de liste de remèdes associée. Le premier énoncé du niveau suivant est alors écrit sur la même ligne, plutôt qu'à la ligne suivante.

Exemple :

Business, averse to : Agar, am-c,...
incapacity for : Agn,...
talks of : Ars,...

au lieu de :

Business,
averse to : Agar, am-c,...
incapacity for : Agn,...
talks of : Ars,...

3.1.2. Le répertoire de Barthel & Klunker.

A la base, l'organisation de ce répertoire est similaire à celle du Kent. Cependant, trois éléments viennent s'ajouter :

- L'énoncé d'un symptôme peut être accompagné d'une référence (un nombre).
- Un énoncé est généralement complété d'une traduction en français et en allemand.
- Un symptôme peut être également complété par une série de mots-clés faisant référence à d'autres rubriques, ces mots-clés étant également traduits.

La structure est encore compliquée par rapport au cas du Kent, puisque les "rubriques doubles" sont accompagnées de "traductions doubles".

3.1.3. Synthèse : structure générale d'un répertoire.

En réunissant les caractéristiques des répertoires de Kent et de Barthel, on obtient la structure de la figure 3.2, avec les conventions :

- un arc [a,b] signifie "a a pour composant b", composant obligatoire si l'arc est barré, facultatif sinon;
- une astérisque "*" signale que l'élément considéré peut avoir plusieurs occurences;
- l'indication "(Barthel)" signifie que l'élément considéré n'apparaît que dans le répertoire de Barthel et Klunker.

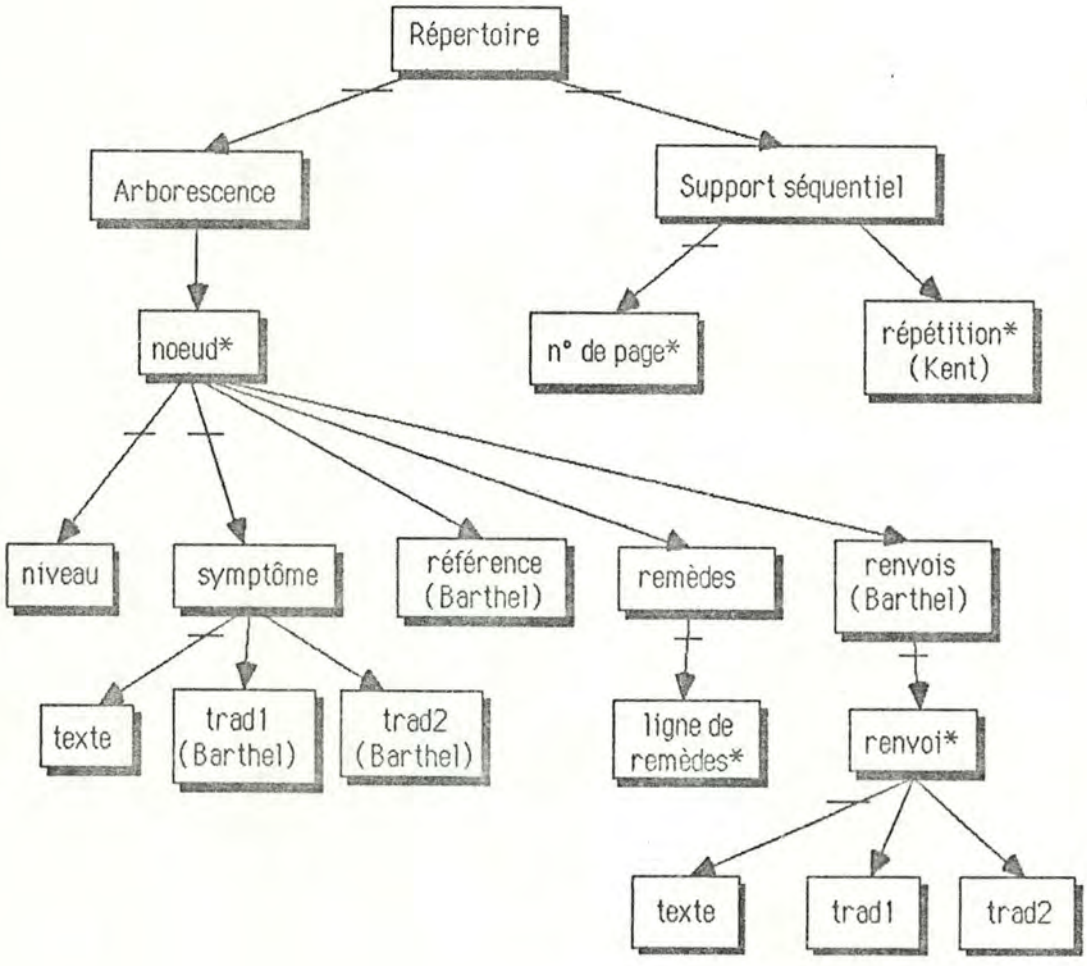


Figure 3.2 : Structure générale d'un répertoire

3.2. Définition précise de la notion de traduction d'un répertoire.

Maintenant que l'on dispose d'une définition plus claire de ce qu'est un répertoire, il est temps de préciser ce que signifie l'affirmation "un répertoire est la traduction d'un autre".

Si on se reporte à la structure de la figure 3.2, on peut dire qu'un répertoire **b** est une traduction d'un répertoire **a**, si **b** a le même contenu que **a**, tant du point de vue du support séquentiel que du point de vue du contenu de l'arborescence, sauf pour ce qui est des composants trad1 et trad2 des symptômes et des renvois, qui peuvent être modifiés comme suit :

- Les composants trad1 et trad2 des renvois et des symptômes sont vides :

  → ce répertoire est une version non traduite du répertoire original

- Les composants trad2 des renvois et des symptômes sont vides, mais pas les composants trad1 :

  → ce répertoire est une version du répertoire original traduite dans une seule langue (l'Esperanto).

- Les composants trad1 et trad2 des renvois et des symptômes sont non vides :

  → ce répertoire est une version du répertoire original traduite dans deux langues.

L'affirmation "Les composants trad1 (trad2) du répertoire sont non vides" n'implique pas que tous ces composants aient une valeur : certaines phrases ne nécessitent pas de traduction[4]. La seule convention qui sera respectée est que si dans un répertoire traduit en deux langues, une phrase n'a qu'une traduction, alors cette traduction est en Esperanto.

Voyons maintenant de plus près en quoi consistera le processus de traduction.

[4] Par exemple les locutions latines, courantes en médecine.

3.3. Etapes de la traduction d'un répertoire.

Vu la taille des répertoires, il sera impossible de les traduire en une seule session interactive. Il faut donc trouver un moyen de décomposer le travail en étapes constituant des unités de traitement homogènes. Les points suivants amorcent cette décomposition.

3.3.1. La conversion Kent-Barthel.

Le système doit pouvoir traduire indifféremment un répertoire de Kent ou un répertoire de Barthel & Klunker. Ces répertoires ont une structure différente, mais le Kent une fois traduit devra avoir la structure du Barthel & Klunker. Deux solutions sont dès lors envisageables :

- Cacher les différences Kent - Barthel & Klunker à l'intérieur d'un type abstrait "répertoire", et fournir un accès identique du point de vue de l'extérieur. Du point de vue interne, procéder à une conversion des fichiers Kent en fichiers Barthel & Klunker au fur et à mesure de la traduction

- Ne faire travailler le programme que sur des fichiers de type Barthel & Klunker, et opérer au préalable une conversion de fichiers dans le cas du Kent.

La première solution oblige à faire cohabiter dans le programme de traduction deux sous-programmes de gestion de fichiers distincts, et nécessite un mécanisme spécial pour détecter quelle procédure utiliser en fonction du fichier traité.

De plus, une session de traduction ne suffisant pas pour traduire complètement un répertoire, cette solution va donner naissance à un répertoire mi-Kent, mi-Barthel & Klunker, pour lequel un troisième programme d'analyse serait nécessaire, à moins de provoquer la conversion complète du fichier à l'issue de la première session de traduction. On voit mal l'utilité d'une telle complication...

On construira donc d'abord un programme de conversion Kent → Barthel & Klunker, et on lui soumettra les fichiers de type Kent avant de les traduire.

3.3.2. La simplification d'un fichier Barthel.

On a vu que suite à l'encodage "fidèle à l'original" des fichiers Barthel & Klunker, on a dû introduire un certain nombre de complications syntaxiques dans ces fichiers, et notamment les "rubriques doubles", ainsi que leurs "traductions doubles".

La prise en compte constante de ces particularités syntaxiques est d'une lourdeur exaspérante, et handicape la mise en oeuvre d'analyseurs syntaxiques "propres".

Les programmeurs du système RADAR se sont d'ailleurs visiblement heurtés eux aussi à ce problème, puisqu'avant de soumettre un fichier à RADAR, ils le confient d'abord à un "optimiseur", dont la tâche consiste notamment à débarrasser ce fichier des rubriques doubles et autres objets ennuyeux du même genre.

Nous suivrons donc leur exemple et transformerons d'abord les fichiers à l'aide d'un programme de simplification syntaxique avant de les livrer au traducteur.

3.3.3. L'enchaînement des étapes de traduction.

On a dit que la traduction d'un répertoire passerait d'abord par une phase de traduction Esperanto, puis par une phase de traduction vers la langue-cible. Précisons les étapes de ce processus.

La première chose à faire avant de traduire un répertoire est de le débarrasser des éventuelles traductions qu'il pourrait contenir. Il faut ensuite le traduire en Esperanto, et enfin le traduire dans une langue-cible.

Ces diverses opérations ont des points en commun :

- vider un répertoire de ses traductions revient à le traduire dans 0 langue;
- traduire un répertoire en Esperanto revient à doter un répertoire sans traductions d'éléments tradl;

- traduire un répertoire dans une langue-cible revient à doter un répertoire traduit en Esperanto d'éléments trad2.

On remarquera que moyennant un paramétrage judicieux, le même programme général de traduction peut resservir dans les trois cas ci-dessus. A la réserve près que le module de traduction qui sera utilisé sera différent dans chaque cas, et que c'est en fait ce module qui sera le plus complexe à réaliser...

3.3.4. Les sessions de traduction interactives.

Il faut maintenant trouver une méthode permettant de fractionner aisément une étape de la traduction en sessions interactives.

Une méthode possible consiste à considérer que traduire un répertoire revient à le parcourir séquentiellement à la recherche des phrases qui ne sont pas encore traduites, à traduire ces phrases, et à itérer le processus jusqu'à ce que l'utilisateur décide de stopper la session, ou que le répertoire soit complètement traduit.

Cette méthode à l'avantage de permettre une utilisation souple du programme : si une phrase pose problème, rien n'empêche de postposer sa traduction; elle sera de nouveau soumise à l'analyse lors de la session interactive suivante.

De plus, sa mise en oeuvre est facile : le programme parcourt séquentiellement le répertoire à traduire, et le recopie au fur et à mesure dans un autre fichier. Lorsqu'il trouve une phrase qui n'a pas de traduction (lors de la traduction en Esperanto) ou qui n'a qu'une seule traduction (lors de la traduction dans une langue-cible), il la transmet au module traducteur qui la lui renvoie avec une traduction; il les recopie alors dans le fichier-résultat, et répète le processus jusqu'à la fin du fichier-source, ou jusqu'à une interruption par l'utilisateur.

3.3.5. Schéma récapitulatif.

Le schéma de la figure 3.3 synthétise les différentes étapes du processus de traduction, et met en évidence les programmes nécessaires.

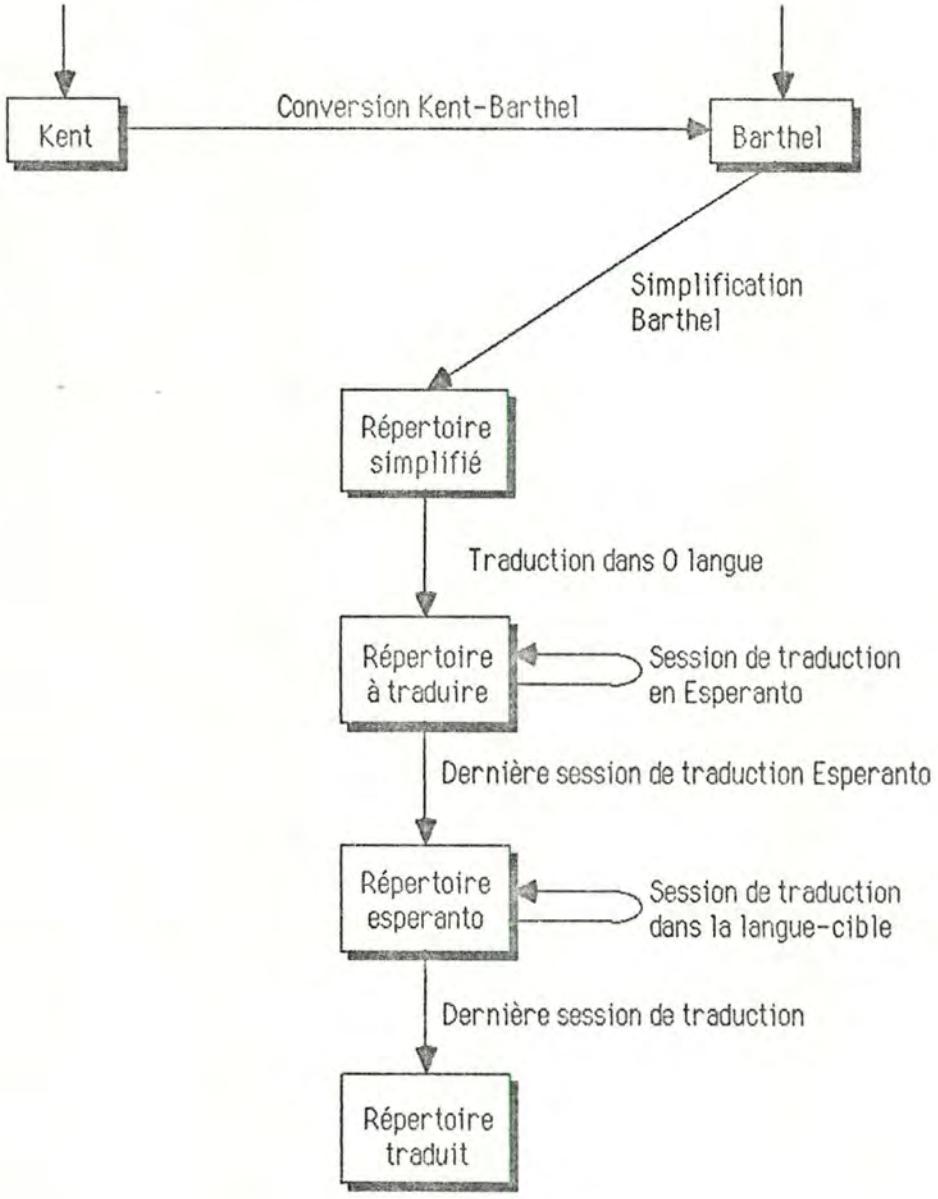


Figure 3.3 : Etapes de la traduction d'un répertoire.

3.4. Les programmes à construire.

Au point précédent, nous avons dégagé les différentes étapes du processus de traduction d'un répertoire, et suggéré l'utilisation d'un certain nombre de programmes. Nous allons maintenant établir une première spécification pour chacun d'eux.

3.4.1. Programme de conversion Kent-Barthel & Klunker.

Ce programme est un simple programme de conversion de fichiers. Etant donné un fichier contenant un répertoire de Kent avec sa syntaxe particulière, il doit en créer un autre qui contienne les mêmes informations que lui, mais dont la syntaxe soit celle du répertoire de Barthel & Klunker.

3.4.2. Programme de simplification d'un répertoire de Barthel & Klunker.

Ce programme est chargé d'éliminer les rubriques doubles et leurs traductions doubles d'un fichier Barthel & Klunker. Il consiste donc en une recopie séquentielle de tout le fichier, avec éclatement des rubriques doubles au fur et à mesure.

3.4.3. Programme de traduction d'un répertoire.

Ce programme constitue évidemment le noyau du système. On a suggéré que son fonctionnement consiste en la recopie séquentielle d'un fichier Barthel & Klunker, lors de laquelle on assortira les symptômes et les renvois de traductions.

On a vu qu'une exigence supplémentaire était que ce programme puisse être réutilisé aux différents stades de la traduction. Il devra donc être paramétré pour être adaptable aux diverses situations possibles.

De plus, sa structure devra être d'une modularité poussée, afin que des composants tels que le traducteur, le dialogue avec l'utilisateur, etc..., puissent être modifiés considérablement dans l'avenir sans entraîner aucune modification supplémentaire que l'écriture d'un nouveau composant.

Sans entrer prématurément dans les détails de l'analyse de ce programme, on peut toutefois déjà dénombrer quelques composants essentiels :

- Un composant "répertoire", qui débarrasse le programme des détails concernant la gestion des fichiers, et permet de considérer le répertoire à traduire comme un

livre qu'on peut ouvrir, où on peut prendre une phrase, remettre la dernière phrase que l'on a prise, et que l'on peut fermer lorsqu'on n'en a plus besoin.

- Un composant "traducteur", à qui on peut donner une phrase (et sans doute aussi son contexte), et qui renvoie la phrase traduite. Ce module lui-même devra être décomposé en sous-modules les plus indépendants possibles, vu le caractère expérimental de son fonctionnement qui exige une modifiabilité poussée.

- Un composant gérant le dialogue avec l'utilisateur, qui débarrassera le programme de tous les détails techniques de gestion d'écran, ainsi que des stratégies de dialogue qui sont elles aussi extrêmement changeantes et imprécises à l'heure actuelle.

On peut dès maintenant établir un premier scénario précis de l'agencement de ces composants lors d'une session interactive de traduction.

La figure 3.4 propose une représentation sous forme d'automate, avec les conventions suivantes :

- Le parcours d'un arc peut être soumis à une ou des conditions. Si c'est le cas, ces conditions sont notées entre deux points d'interrogation : ?xxx?.

- Le parcours d'un arc peut exiger l'accomplissement de certaines actions. Si c'est le cas, ces actions sont notées entre deux points d'exclamation : !xxx!.

- Lorsque plusieurs arcs partent du même état, il faut les prendre en compte successivement dans le sens des aiguilles d'une montre jusqu'à ce qu'il soit possible d'en emprunter un.

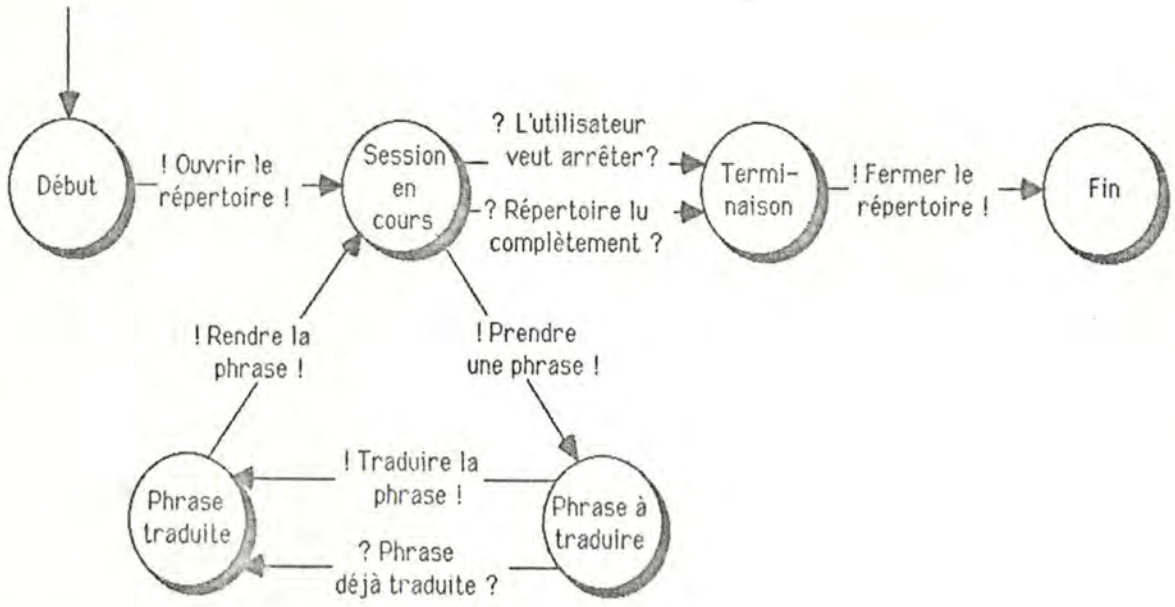


Figure 3.4 : Graphe des états d'une session de traduction.

4. Conclusion de la première partie.

Pour en terminer avec cette partie préliminaire, il faut maintenant définir les résultats qui peuvent être attendus de ce travail.

Il est évident que le programme de conversion syntaxique Kent - Barthel & Klunker et le programme de simplification d'un fichier Barthel & Klunker seront réalisés, parce qu'ils sont un préalable au fonctionnement du programme de traduction, et que leur programmation ne doit poser aucun problème particulier.

Reste à décider de ce qui peut être fait pour le programme de traduction lui-même.

Ce programme a deux aspects : un aspect linguistique (techniques d'analyse grammaticale, méthode de traduction, etc...) et un aspect non linguistique (gestion de fichiers, dialogue avec l'utilisateur, etc...).

Il est clair que l'aspect linguistique ne pourra pas être traité complètement au terme de ce mémoire. On ne dispose actuellement ni des règles grammaticales, ni de l'entièreté du lexique nécessaires pour pouvoir espérer fournir une traduction complète d'un répertoire en Esperanto - a fortiori dans une langue-cible. Tout au plus pourra-t-on effectuer une analyse critique des techniques généralement utilisées, et proposer une voie de recherche pour les travaux futurs.

L'aspect non-linguistique, par contre, est tout-à fait réalisable puisqu'il revient à élaborer un certain nombre de modules spécialisés dont la fonction est connue.

La suite de ce travail se décomposera donc comme suit :

- Construction des programmes "utilitaires" de conversion et de simplification de fichiers.
- Constuction d'un programme gérant les aspects non linguistiques du programme de traduction, doté d'une architecture modulaire claire et simple à utiliser, dans lequel il suffira par la suite d'insérer un composant capable de traduire les phrases des répertoires.
- Réflexions linguistiques plus théoriques sur l'analyse grammaticale et la traduction.

Deuxième partie :

Programmes utilitaires.

5. Programme de conversion Kent-Barthel & Klunker.

5.1. Présentation.

Etant donné un fichier contenant un répertoire de Kent, ce programme doit créer un second fichier contenant les mêmes informations, mais sous la forme syntaxique d'un répertoire de Barthel & Klunker.

On supposera que le fichier reçu est correct syntaxiquement. La gestion d'erreurs sera donc réduite à sa plus simple expression : arrêt du programme, envoi d'un message explicatif, et affichage du numéro de la ligne où l'erreur a été découverte.

Avant de construire le programme proprement dit, nous allons présenter la syntaxe des deux répertoires.

5.2. Définition syntaxique d'un fichier Kent.

5.2.1. Introduction.

Nous avons vu que la syntaxe du Kent était compliquée, parce que sa structure est une superposition de deux structures distinctes (arborescence et support livresque) qui n'ont pas de rapport logique entre elles. Une conséquence directe en est l'apparition de numéros de pages et de répétitions de mots-clés "n'importe où", quelquefois au milieu de la liste de remèdes d'un symptôme. Une définition syntaxique complète du répertoire de Kent devrait pouvoir en rendre compte.

Cependant, puisque le système à construire n'est pas concerné par les listes de remèdes, on peut tourner la difficulté en découplant les symptômes de leurs remèdes. On obtient alors la syntaxe présentée ci-dessous, adaptée de celle qui figure dans la documentation du système RADAR.

5.2.2. Notations utilisées.

On emploie une notation du genre de celle de Backus. Ce formalisme étant suffisemment connu, on se contentera de deux exemples de règles syntaxiques :

Exemple 1 : la règle :

<REPERTOIRE> ::= <SEQUENCE>$^{1..\infty}$ $

peut s'interpréter :

"Un répertoire est constitué d'une suite non vide de séquences, de taille potentiellement infinie, suivie d'un caractère '$' ".

Exemple 2 : la règle :

<NIVEAU-RUBRIQUE> ::= <NIVEAU-SYMPTOME> | <NIVEAU-RENVOI>

peut s'interpréter :

"Un niveau-rubrique est soit un niveau-symptôme, soit un niveau-renvoi".

5.2.3. Règles syntaxiques.

<REPERTOIRE> ::= <SEQUENCE>$^{1..\infty}$ $

<SEQUENCE> ::= <PAGINATION> | <REPETITION> | <NIVEAU-RUBRIQUE> | <LIGNE-MEDICAMENTS>

<PAGINATION> ::= # <chiffre>$^{1..4}$ <caractère 'r' ou 'l'>$^{0..1}$ CR

<REPETITION> ::= = <caractère>$^{1..\infty}$ CR

<NIVEAU-RUBRIQUE> ::= <NIVEAU-SYMPTOME> | <NIVEAU-RENVOI>

<NIVEAU-SYMPTOME> ::= <ENONCE>$^{1..3}$: CR $^{0..1}$

<NIVEAU-RENVOI> ::= <ENONCE> . (<caractère>$^{1..\infty}$) CR

<ENONCE> ::= & <chiffre> <caractère sauf ':', '.' et '&'>$^{1..\infty}$

<LIGNE-MEDICAMENTS> ::= <LIGNE-QUELCONQUE-LISTE> | <DERNIERE-LIGNE-LISTE>

<LIGNE-QUELCONQUE-LISTE> ::= <caractère>$^{1..\infty}$, CR

<DERNIERE-LIGNE-LISTE> ::= <caractère>$^{1..\infty}$ CR

5.3. Définition syntaxique d'un fichier Barthel & Klunker.

La définition ci-dessous utilise les mêmes notations que pour le Kent. Ici aussi, on a découplé les symptômes de leurs remèdes pour simplifier la syntaxe.

Règles syntaxiques :

<REPERTOIRE> ::= <SEQUENCE>$^{1..\infty}$ =

<SEQUENCE> ::= <PAGINATION> | <NIVEAU-RUBRIQUE> | <RENVOIS> | <LIGNE-MEDICAMENTS>

<PAGINATION> ::= # <chiffre>$^{1..4}$ CR

<NIVEAU-RUBRIQUE> ::= <ENONCE-NIVEAU>$^{1..2}$ <LISTE-TRADUCTIONS>$^{0..1}$: CR

<ENONCE-NIVEAU> ::= <ENONCE> <REFERENCE>$^{0..1}$

<ENONCE> ::= & <chiffre> <caractère sauf ':', '[', CR et '&'>$^{1..\infty}$

<REFERENCE> ::= [<caractère>$^{1..\infty}$]

<LISTE-TRADUCTIONS> ::= <LIGNE-TRADUCTIONS>$^{1..2}$

<LIGNE-TRADUCTIONS> ::= CR <TRADUCTION> <TRADUCTION-DOUBLE>$^{0..1}$

<TRADUCTION> ::= <caractère sauf '*', CR et ':'>$^{0..\infty}$

<TRADUCTION-DOUBLE> ::= * <TRADUCTION>

<RENVOI> ::= + <LIGNE-NORMALE-RENVOI>$^{0..\infty}$ <DERNIERE-LIGNE-RENVOI> + CR

<LIGNE-NORMALE-RENVOI> ::= <CORPS-RENVOI> CR

<DERNIERE-LIGNE-RENVOI> ::= <CORPS-RENVOI>

<CORPS-RENVOI> ::= <caractère>$^{1..\infty}$ / <caractère>$^{0..\infty}$ / <caractère>$^{0..\infty}$

<LIGNE-MEDICAMENTS> ::= <LIGNE-NORMALE-LISTE> | <DERNIERE-LIGNE-LISTE>

<LIGNE-NORMALE-LISTE> ::= <caractère y-compris ','>$^{1..\infty}$, CR

<DERNIERE-LIGNE-LISTE> ::= <caractère sauf ':'>$^{1..\infty}$: CR

5.4. Elaboration du programme de conversion.

5.4.1. Variables globales :

FichKent : Fichier Kent à convertir

FichResult : Fichier résultat.

InSeqCour : Chaîne de caractères contenant la séquence courante du fichier d'entrée.

InCarCour : Caractère courant du fichier d'entrée.

LCour : Entier contenant le numéro de la ligne courante du fichier Kent (pour l'afficher en cas d'erreur).

5.4.2. Procédures utilitaires :

OuvrirFichiers : Ouvre FichKent en lecture et FichResult en écriture.

FermerFichiers : Ferme FichKent et FichResult.

Lire(car) : Lit le caractère *car* dans le fichier FichKent.

Ecrire(car) : Ecrit le caractère *car* dans le fichier FichResult.

Erreur(message) : Affiche la chaîne de caractères *message* sur l'écran, affiche LCour, et stoppe l'exécution du programme.

5.4.3. Algorithme principal :

◇ Conditions invariantes au point d'itération :

- La position courante dans FichKent est sur le premier caractère qui n'a pas été traité (la prochaine exécution de lire(car) aura pour effet de lire ce caractère).
- Le fichier FichResult contient la conversion syntaxique de la partie de FichKent allant du début à la position courante non comprise.
- LCour a la valeur du nombre de caractères ‹CR› déjà lus dans FichKent, plus 1 (ce qui représente bien le numéro de la ligne courante).

◇ Initialisation :

```
OuvrirFichiers;
LCour:=1;
```

◇ Itération :

```
Lire(InCarCour);
si(InCarCour='$') Ecrire('='); terminer l'exécution;
sinon si(InCarCour=EOF) Erreur("Le fichier Kent ne finit pas correctement");
sinon si(InCarCour='#') TraiterPagination;
sinon si(InCarCour='=') TraiterRépétition;
sinon si(InCarCour='&') TraiterNiveau;
sinon TraiterLigneMedic;
```

◇ Terminaison :

FermerFichiers;

◇ Spécification commune aux procédures "Traiter..." :

- Elles recopient dans FichResult la conversion syntaxique de la séquence qui leur correspond, séquence dont :
  - le premier caractère est dans InCarCour
  - les caractères suivants sont dans FichKent à partir de la position courante.
- Leur exécution met la position courante de FichKent sur le premier caractère qui suit la séquence qu'elles ont convertie.
- Elles incrémentent LCour chaque fois qu'elles lisent un caractère <CR> dans FichKent.

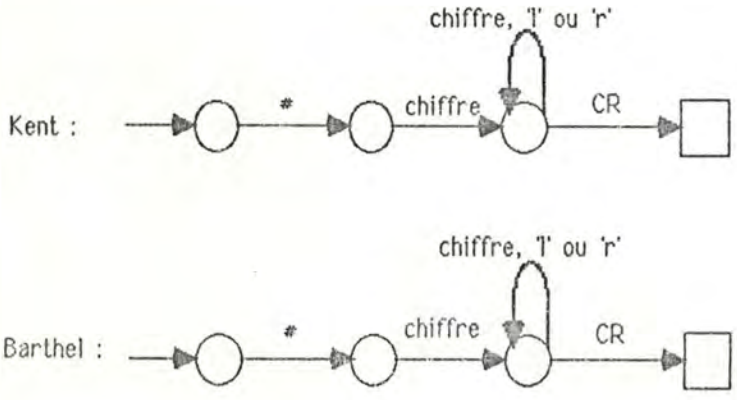
On vérifiera aisément que ces spécifications assurent le respect de l'invariant de la procédure principale.

5.4.4. Algorithme de la procédure TraiterPagination:

◇ Syntaxe des paginations sous forme d'automates :

Conventions: - Un état représente un stade du parcours d'une chaîne de caractères.
- Les transitions entre états définissent les caractères qui sont admis par la syntaxe aux différents stades de la lecture d'une chaîne.
- Si une chaîne peut être lue complètement en suivant les transitions d'un automate, et que l'automate est dans l'état final lorsque la chaîne est lue, alors la chaîne est correcte du point de vue de la syntaxe définie par l'automate.

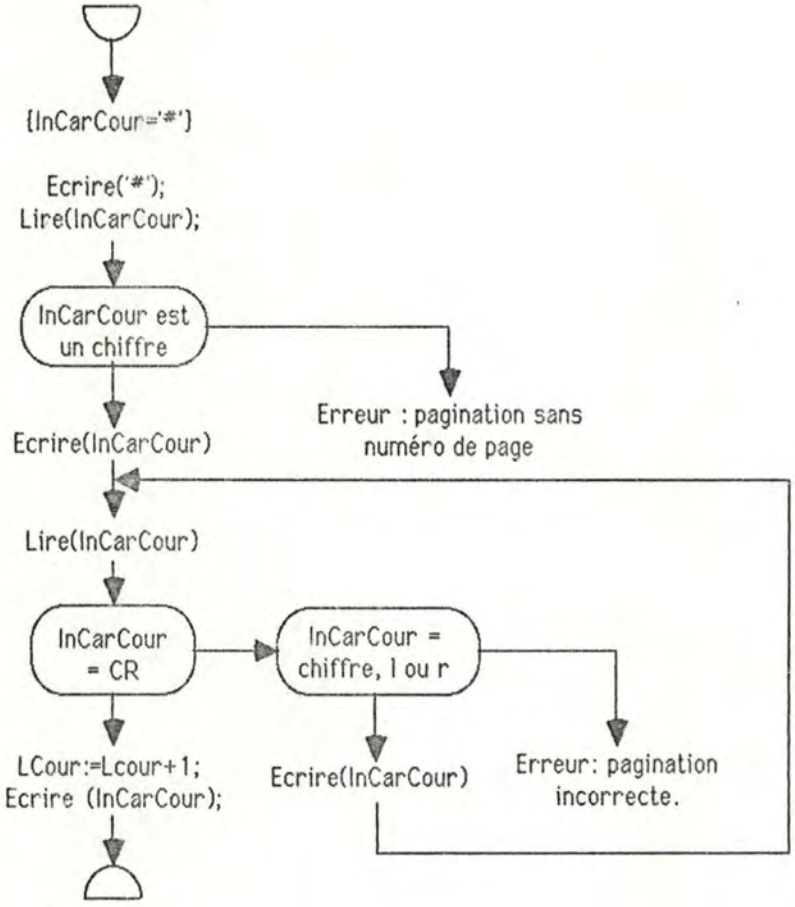
Automates:



◇ Principe de l'algorithme :

La syntaxe des paginations Kent et Barthel est identique. Il faut donc simplement recopier les caractères de FichKent dans FichResult jusqu'à la découverte du CR de fin de pagination, en vérifiant que les caractères lus correspondent bien à la syntaxe.

◇ Organigramme :

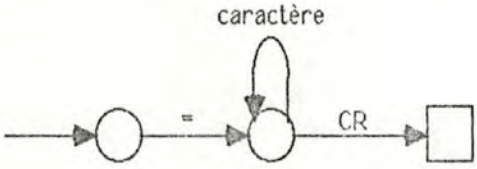


5.4.5. Algorithme de la procédure TraiterRépétition:

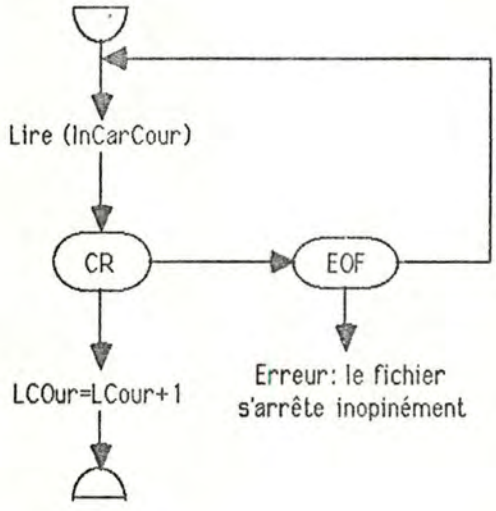
◇ Problème posé :

Les répétitions n'existent pas chez Barthel. Il faut donc seulement lire les caractères pour "sauter" la répétition courante.

◇ Syntaxe d'une répétition sous forme d'automate :



◇ Organigramme :



5.4.6. Algorithme de la procédure TraiterNiveau:

◇ Principe :

Le répertoire de Barthel & Klunker ne possède pas de "niveaux-renvois" comme celui de Kent. On doit traiter ces "niveaux-renvois" comme des énoncés de symptômes.

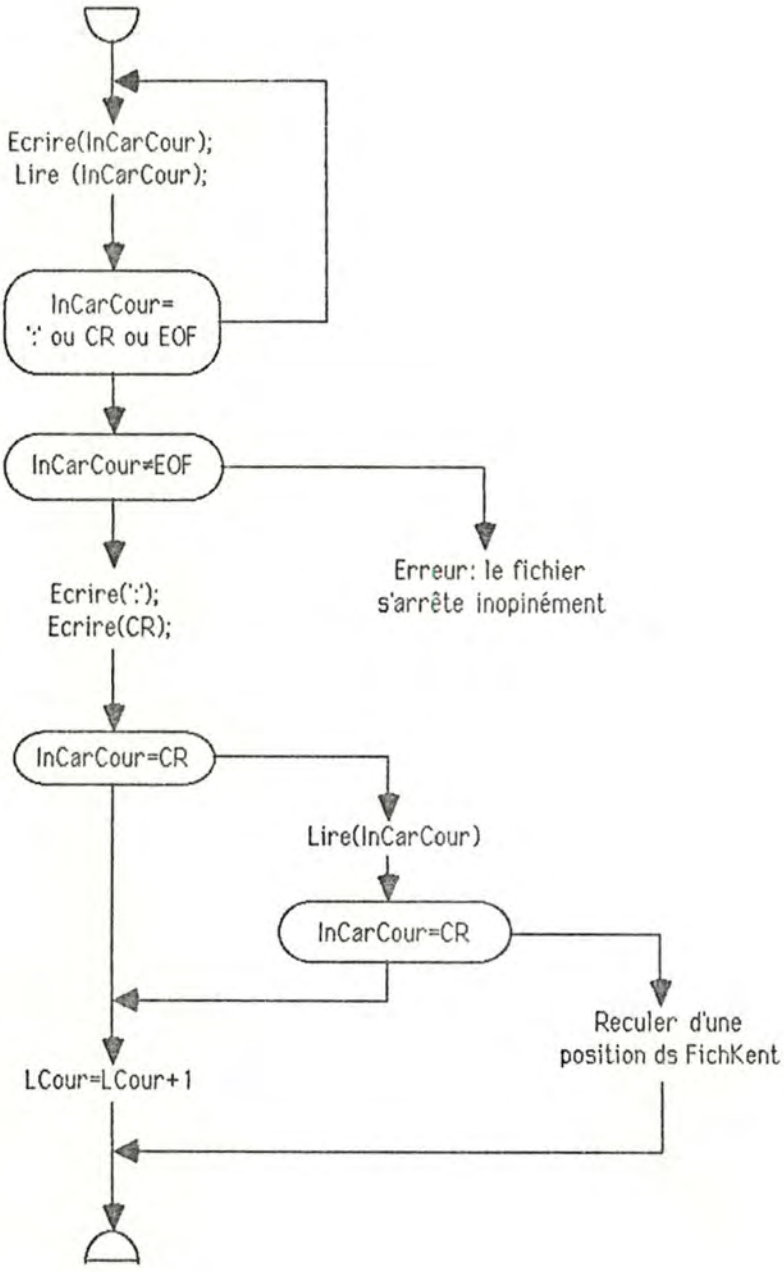
Il faut donc, dans tous les cas :

- recopier tous les caractères jusqu'à ce qu'on trouve un ':' ou un CR, que l'on ne recopie pas,
- écrire ':' puis CR.

Puis, si le dernier caractère lu était un ':', c'est qu'on était dans le cas d'un "niveau-rubrique". Le caractère suivant est donc peut-être un CR, qu'il faudrait sauter pour positionner le fichier sur le premier caractère de la séquence suivante.

Il faut alors lire un caractère, et s'il s'agit de CR, incrémenter LCour. Sinon, il faut reculer de 1 la position courante dans le fichier d'entrée.

◇ Organigramme :



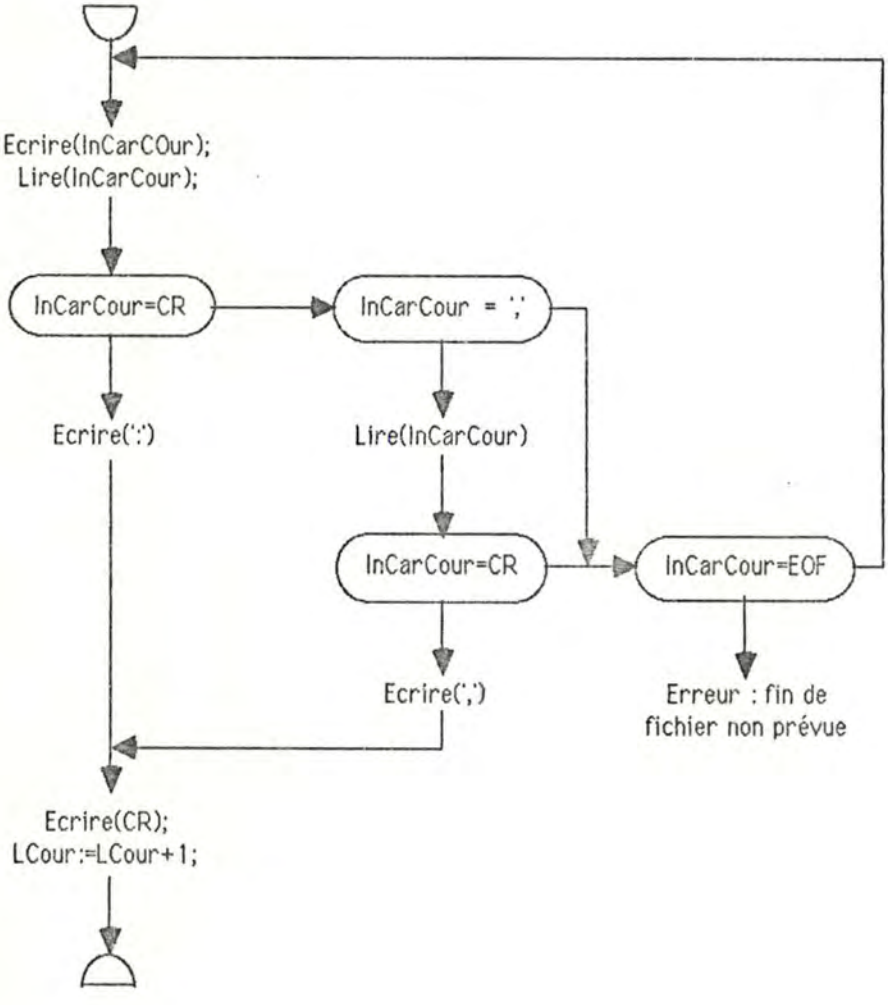
5.4.7. Algorithme de la procédure TraiterLigneMédic:

◇ Principe de l'algorithme :

Il y a 2 sortes de lignes de médicaments : les "lignes normales", terminées par < , CR >, et les "dernières lignes", seulement terminées par <CR>.

Les premières sont à recopier telles quelles, tandis que dans le cas des secondes, il faut insérer un ':' avant le CR.

◇ Organigramme :



5.5. A propos du codage C du programme.

Ce codage consiste a créer les fonctions utilitaires du point 5.4.2., puis à transposer sous forme de fonctions C les différents organigrammes des points précédents.

5.6. Test du programme.

Le test du programme consistera à l'exécuter sur un fichier Kent contenant toutes les séquences possibles (paginations, répétitions, niveaux-rubriques, lignes-médicaments), et de vérifier que toutes ces séquences auront bien été traduites dans la syntaxe de Barthel & Klunker.

Pour tester le fonctionnement du mécanisme de gestion d'erreurs, on exécutera également le programme sur un fichier Kent contenant diverses erreurs syntaxiques (rubriques sans numéro de niveau, pagination sans numéro de page, fin de fichier au milieu d'une séquence, etc...).

6. Programme de simplification d'un fichier Barthel.

6.1. Présentation.

Le programme reçoit en entrée un fichier Barthel & Klunker, qui contient peut-être certaines rubriques doubles, accompagnées éventuellement d'une ou deux traductions doubles (cfr 5.3. : syntaxe du répertoire de Barthel et Klunker).

Il doit créer un nouveau fichier qui a le même contenu que le premier, sauf pour ce qui est des rubriques doubles, qui doivent être "éclatées", c'est-à-dire recopiées l'une à la suite de l'autre comme deux rubriques simples.

En toute généralité, ce processus se schématise comme suit. Soit la rubrique double suivante :

&<Niv 1> <Enoncé 1> <Réf 1> &<Niv 2> <Enoncé 2> <Réf 2>
<Trad1 Enoncé 1> * <Trad1 Enoncé 2>
<Trad2 Enoncé 1> * <Trad2 Enoncé 2> :

où <Niv 1>, <Enoncé 1>, <Réf 1>, <Trad1 Enoncé 1>, <Trad 2 énoncé 1> représentent respectivement le numéro de niveau, l'énoncé , la référence (facultative), la première et la deuxième traductions (facultatives) du premier symptôme; de même pour <Niv 2>, <Enoncé 2>, <Réf 2>, <Trad1 Enoncé 2>, <Trad 2 énoncé 2> par rapport au second symptôme.

Cette rubrique doit se transformer en la suite de rubriques :

&<Niv 1> <Enoncé 1> <Réf 1>
<Trad1 Enoncé 1>
<Trad2 Enoncé 1> :

et :

&<Niv 2> <Enoncé 2> <Réf 2>
<Trad1 Enoncé 2>
<Trad2 Enoncé 2> :

6.2. Elaboration du programme.

6.2.1. Variables globales :

FichIn : Fichier d'entrée.

FichOut : Fichier résultat.

InCarCour : Caractère courant du fichier d'entrée.

LCour : Entier contenant le numéro de la ligne courante du fichier Kent (pour l'afficher en cas d'erreur).

6.2.2. Procédures utilitaires :

OuvrirFichiers : Ouvre FichKent en lecture et FichResult en écriture.

FermerFichiers : Ferme FichKent et FichResult.

Lire(car) : Lit le caractère *car* dans le fichier FichKent.

Ecrire(car) : Ecrit le caractère *car* dans le fichier FichResult.

Erreur(message) : Affiche la chaîne de caractères *message* sur l'écran, affiche LCour, et stoppe l'exécution du programme.

6.2.3. Algorithme principal :

◇ Conditions invariantes au point d'itération :

- La position courante dans FichIn est sur le premier caractère qui n'a pas été traité (la prochaine exécution de lire(car) aura pour effet de lire ce caractère).
- Le fichier Fichout contient la simplification de la partie de FichIn allant du début à la position courante non comprise.
- LCour a la valeur du nombre de caractères ‹CR› déjà lus dans FichIn, plus 1 (ce qui représente bien le numéro de la ligne courante).

◇ Initialisation :

```
OuvrirFichiers;
LCour:=1;
```

◇ Itération :

```
Lire(InCarCour);
si(InCarCour='=') Ecrire('='); terminer l'exécution;
sinon si(InCarCour=EOF) Erreur("Le fichier ne finit pas correctement");
sinon si(InCarCour='&') Reculer d'un caractère dans FichIn; TraiterNiveau;
sinon si(InCarCour=CR) Incrémenter LCour; Ecrire(InCarCour);
sinon Ecrire(InCarCour);
```

◇ Terminaison :

FermerFichiers;

◇ Spécification de la procédure TraiterNiveau :

- Elle recopie dans Fichout la séquence de type niveau-rubrique qui commence à la position courante de FichIn, et, s'il y a lieu, éclate cette séquence en deux séquences distinctes.
- Son exécution met la position courante de FichIn sur le premier caractère qui suit la séquence qu'elle a traitée.
- Elle incrémente LCour chaque fois qu'elle lit un caractère ‹CR› dans FichKent.

On vérifiera aisément que cette spécification assure le respect de l'invariant de la procédure principale.

6.2.4. Algorithme de la procédure TraiterNiveau:

◇ Variables utilisées :

Enoncé1, Enoncé2 : Chaînes de caractères contenant les énoncés du 1er et du 2ème symptômes de la rubrique courante.

Réf1, Réf2 : Chaînes de caractères contenant les références éventuelles du 1er et du 2ème symptômes de la rubrique courante.

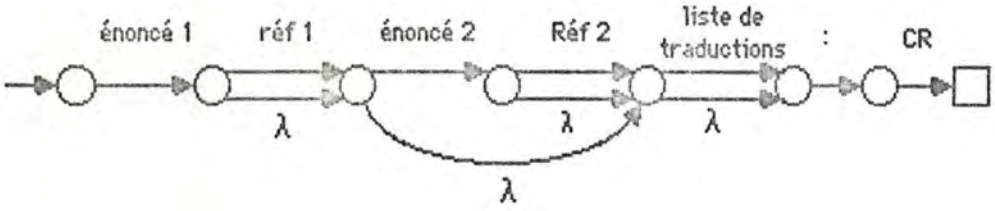
Trad1E1, Trad1E2 : Chaînes de caractères contenant les 1ères traductions éventuelles du 1er et du 2ème symptômes de la rubrique courante.

Trad2E1, Trad2E2 : Chaînes de caractères contenant les 2èmes traductions éventuelles du 1er et du 2ème symptômes de la rubrique courante.

NB : Tous les éléments concerant le 2ème symptôme de la rubrique courante sont vides si la rubrique courante n'est pas une rubrique double.

◇ Définition d'une rubrique sous forme d'automate :

NB: Le symbole "λ" signifie que l'on peut emprunter l'arc correspondant sans effectuer de lecture dans le fichier.



◇ Algorithme de lecture d'une rubrique :

Cet algorithme est un simple codage de l'automate ci-dessus :

```
Enoncé1:=Enoncé2:=Réf1:=Réf2:=Trad1E1:=Trad1E2:=Trad2E1:=Trad2E2:=VIDE;
si LireEnoncé(Enoncé1) renvoie "raté" : Erreur("Rubrique sans énoncé");
sinon :
    LireRéférence(Réf1);
    si LireEnoncé(Enoncé1) renvoie "ok" : LireRéférence(Réf2);
    LireListeTrad(Trad1E1,Trad2E1,Trad1E2,Trad2E2);
    Lire (InCarCour);
    si(InCarCour ◇ ':') : Erreur("Rubrique non terminée par ':'");
    sinon
        lire(InCarCour);
        si(InCarCour◇CR) : Erreur("Rubrique non terminée par CR");
        sinon  Incrémenter LCour;
                RecopierRubrique(Enoncé1,Enoncé2,Réf1,Réf2,Trad1E1,
                                 Trad2E1,Trad1E2,Trad2E2);
```

◇ Spécification des procédures auxiliaires :

NB: Toutes ces procédures doivent incrémenter LCour si elles lisent un CR dans FichIn.

- LireEnoncé(Enoncé)

Fonction qui lit une séquence de type Enoncé dans FichIn, et renvoie cette séquence dans *Enoncé*. Si tout s'est bien passé, prend la valeur "ok", sinon, prend la valeur "raté". Dans ce dernier cas, l'argument est une chaîne vide. Positionne FichIn sur le 1er caractère qui suit la séquence lue.

- LireRéférence(Réf)

Fonction qui lit une séquence de type Référence dans FichIn, et renvoie cette séquence dans *Réf*. Si tout s'est bien passé, prend la valeur "ok", sinon, prend la valeur "raté". Dans ce dernier cas, l'argument est une chaîne vide. Positionne FichIn sur le 1er caractère qui suit la séquence lue.

- LireListeTrad(Trad1E1,Trad2E1,Trad1E2,Trad2E2)

Fonction qui lit dans FichIn une séquence de type "liste de traductions", et renvoie:

- dans *Trad1E1*, la 1ère traduction (éventuellement vide) de la 1ère ligne,
- dans *Trad2E1*, la 1ère traduction (éventuellement vide) de la 2ème ligne,
- dans *Trad1E2*, la 2ème traduction (éventuellement vide) de la 1ère ligne,
- dans *Trad2E2*, la 2ème traduction (éventuellement vide) de la 2ème ligne.

Si tout s'est bien passé, prend la valeur "ok", sinon, prend la valeur "raté". Dans le premier cas, il y a au moins un argument non vide. Dans le second cas, tous les arguments sont vides. Positionne FichIn sur le 1er caractère qui suit la séquence lue.

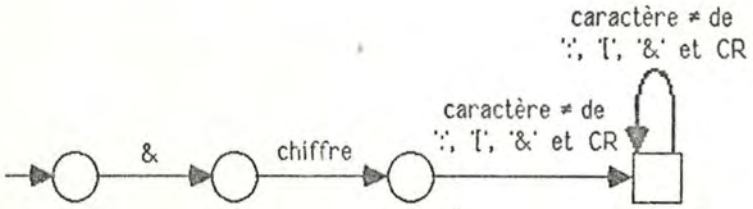
- RecopierRubrique(Enoncé1,Enoncé2,Réf1,Réf2,Trad1E1, Trad2E1,Trad1E2,Trad2E2)

La procédure considère que *Enoncé1*, *Réf1*, *Trad1E1* et *Trad2E1*, constituent les composants d'une rubrique simple à recopier dans FichOut selon la syntaxe du Barthel & Klunker. Elle effectue donc cette recopie.

Si *Enoncé2* est non vide, elle considère de la même façon que *Enoncé2*, *Réf2*, *Trad1E2* et *Trad2E2*, sont les composants d'une deuxième rubrique simple, qu'elle recopie également dans FichOut.

6.2.5. Algorithme de la procédure LireEnoncé(Enoncé).

◇ Définition de la syntaxe d'un énoncé sous forme d'automate:

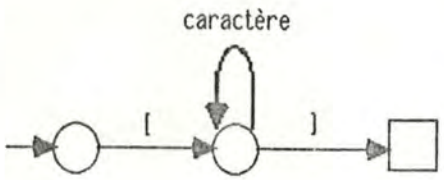


◇ Algorithme:

```
Enoncé:=VIDE;
lire(InCarCour);
si (InCarCour <> '&') :  reculer d'une position dans le fichier;
                         renvoyer "raté";
sinon :
        ajouter InCarCour à l'énoncé;
        lire(InCarCour);;
        si InCarCour n'est pas un chiffre : Erreur("Rubrique sans n° de niveau");
        sinon :
                ajouter InCarCour à l'énoncé;
                lire(InCarCour);
                tant que InCarCour est différent de ':', '[', '&' et CR :
                        si (InCarCour = EOF) : Erreur("fin de fichier inopinée");
                        sinon :
                                ajouter InCarCour à l'énoncé;
                                lire(InCarCour);
                fintant;
                reculer d'une position dans le fichier;
                renvoyer "ok".
```

6.2.6. Algorithme de la procédure LireRéférence(Réf):

◇ Définition de la syntaxe d'une référence sous forme d'automate:



◇ Algorithme:

```
Référence:=VIDE;
lire(InCarCour);
si (InCarCour ◇ '['):   reculer d'une position dans le fichier;
                        renvoyer "raté";
sinon :
        ajouter InCarCour à Référence;
        lire(InCarCour);
        tant que (InCarCour ◇ ']'):
                si (InCarCour = 'EOF') : Erreur("fin de fichier inopinée");
                sinon :
                        ajouter InCarCour à Référence;
                        si (InCarCour = CR) : incrémenter LCour;
                        lire(InCarCour);
        fintant;
        ajouter InCarCour à Référence;
        renvoyer "ok".
```

6.2.7. Algorithme de la procédure LireListeTrad(Trad1E1,Trad2E1,Trad1E2,Trad2E2):

◇ Algorithme principal :

Une liste de traductions est constituée de 0, 1 ou 2 lignes de traductions. On peut les lire comme suit :

```
Trad1E1:=Trad2E1:=Trad1E2:=Trad2E2:=VIDE;
si LireLigneTrad(Trad1E1, Trad1E2) renvoie "raté" : renvoyer "raté";
sinon LireLigneTrad(Trad2E1,Trad2E2).
```

◇ Spécification de la fonction LireLigneTrad(TradE1, TradE2) :

Lit une ligne de traductions dans FichIn à partir de la position courante, et range dans TradE1 et TradE2 les traductions lues (éventuellement vides).

Si elle a réellement lu une ligne de traductions, renvoie la valeur "ok". Sinon, renvoie "raté". Dans ce dernier cas, la valeur des arguments est non significative.

Laisse le fichier positionné sur le 1er caractère qui suit la ligne de traductions.

◇ Elaboration de l'algorithme de LireLigneTrad :

D'après la syntaxe du Barthel & Klunker, après le ou les Enoncés-Niveaux d'une rubrique, ainsi qu'après une ligne de traductions, on peut trouver dans le fichier :

- soit un ':', délimitant la fin de la rubrique,
- soit un 'CR', délimitant le début d'une ligne de traductions.

Dans le premier cas, il faut reculer de 1 la position courante de FichIn, et renvoyer "raté". Dans le second cas, il faut lire les caractères du fichier et les ranger correctement dans les arguments, jusqu'à ce qu'on trouve un caractère ':' ou CR. Il faut alors reculer de

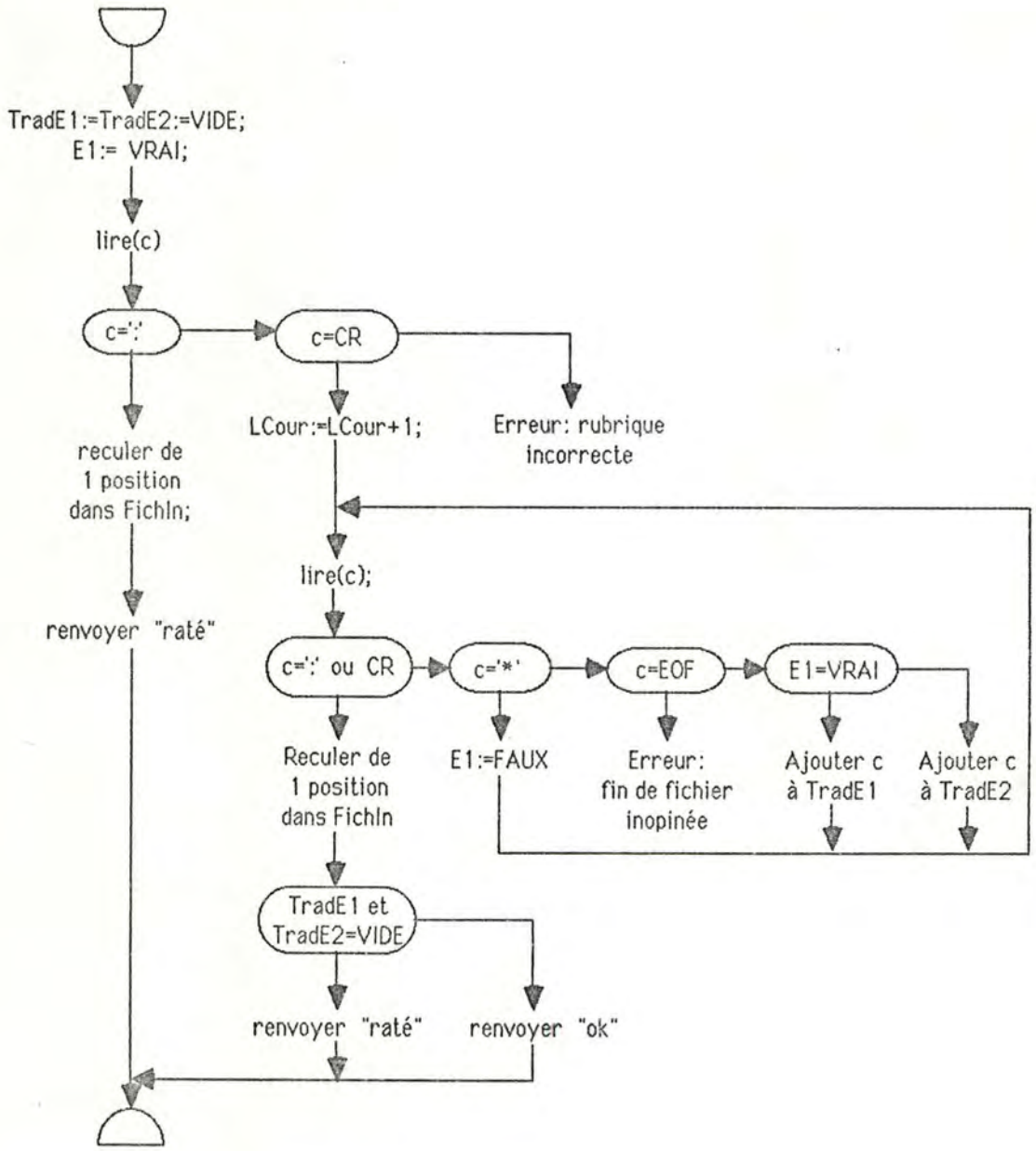
1 la position courante du fichier, et renvoyer "ok" si au moins un argument est non vide, "raté" sinon.

"Ranger correctement les caractères dans les arguments" signifie :

- ranger tous les caractères qui précèdent un éventuel '*' dans TradE1;
- ranger tous les autres dans TradE2.

Pour cela, on utilisera un booléen "E1", vrai si le caractère courant doit être rangé dans TradE1, faux sinon. On l'initialisera à "vrai", et on le mettra à "faux" si on rencontre un '*'.

◇ Organigramme de LireLigneTrad :



6.2.8. Algorithme de la procédure RecopierRubrique(Enoncé1, Enoncé2, Réf1, Réf2, Trad1E1, Trad2E1, Trad1E2, Trad2E2):

◇ Algorithme principal :

Par hypothèse, les éléments de la 1ère rubrique à recopier ont une valeur. Les arguments de la 2ème n'ont une valeur que si Enoncé2 est non vide. On peut écrire l'algorithme :

```
RecopierEnoncé(Enoncé1, Réf1, Trad1E1, Trad2E1);
si (Enoncé2 ◇ VIDE) : RecopierEnoncé(Enoncé2, Réf2, Trad1E2, Trad2E2);
```

◇ Spécification de la procédure RecopierEnoncé(Enoncé, Réf, Trad1, Trad2) :

Copie dans FichOut la rubrique consituée de Enoncé, Réf (éventuellement vide), Trad1 et Trad2 (éventuellement vides), conformément à la syntaxe de Barthel & Klunker.

◇ Algorithme de RecopierEnoncé :

La procédure utilise une sous-procédure Copier(chaine), qui copie dans FichOut les caractères de la chaîne de caractères *chaine*:

```
Copier(Enoncé);
si(Réf◇VIDE) : Copier(Réf);
si(Trad1 ◇ VIDE) : Ecrire(CR); Copier(Trad1);
si(Trad2 ◇ VIDE) : Ecrire(CR); Copier(Trad2);
Ecrire(':');
Ecrire(CR).
```

6.3. A propos du codage C du programme.

Ce codage consiste à transposer en C les diférents algorithmes du point 6.2. Les procédures utilitaires créées pour le programme de conversion syntaxique Kent - Barthel & Klunker peuvent être réutilisées telles quelles.

6.4. Test du programme.

Le programme doit être capable :

- de recopier exactement (au caractère près) un fichier qui ne contiendrait pas de rubriques doubles,

- d'éclater les rubriques doubles d'un fichier qui en contient, sans altérer le reste de son contenu.

Il y aura donc 2 types de tests :

- Recopie fidèle d'un fichier Barthel & Klunker sans rubriques doubles (un petit programme C vérifiant l'égalité de 2 fichiers peut être construit à cet effet).

- Recopie d'un fichier Barthel & Klunker contentant des rubriques doubles (la correction du résultat sera vérifiée "manuellement" sur listing).

Troisième partie :

Eléments non linguistiques du programme de traduction.

7. Principes généraux concernant l'implémentation du programme.

7.1. Objets et primitives.

On sait qu'il est impossible de maîtriser la complexité d'un logiciel si on l'aborde dans son entier, sans le décomposer en éléments les plus indépendants possibles, que l'on peut construire, tester et modifier séparément sans devoir sans cesse remettre le reste du système en question [AVL].

L'analyse d'un système passe donc d'abord par sa décomposition en modules, unités homogènes renfermant des données et fournissant des primitives de manipulation de ces données qui forment un tout du point de vue conceptuel.

Une fois les différents modules déterminés, il faut définir leurs relations, les types de données qu'ils échangeront, la forme de ces données, etc... On peut alors structurer l'ensemble des modules selon une loi telle que *"utilise les services de"*.

Ce n'est qu'après ce travail préparatoire que l'on pourra envisager de construire les différents modules.

Des langages de programmation tels que Smalltalk proposent un certain nombre de concepts qui facilitent le passage des spécifications abstraites d'un système modulaire à son implémentation concrète.

En Smalltalk, les modules sont appelés "objets", et, lorsqu'on les invoque par un "message" (primitive d'accès), ils sont capables d'agir en modifiant leur propre contenu ou en faisant appel à d'autres objets - sans devoir connaître les détails de leur implémentation interne -. Le fonctionnement du programme résulte alors de ces relations inter-objets.

Une autre caractéristique fondamentale des objets Smalltalk est que les objets de même nature se regroupent en "classes", et que de nouvelles classes peuvent être définies sur base des classes existantes, en gardant certaines propriétés, et en en créant d'autres.

L'intérêt de cette approche est qu'elle correspond assez bien avec la façon habituelle que l'on a d'aborder les problèmes : il semble tout "naturel" de considérer la réalité comme

un ensemble d'"objets" qui communiquent et interagissent, en utilisant les services d'"objets" spécialisés pour effectuer certaines tâches compliquées.

On peut ainsi arriver à construire des systèmes de taille importante, que l'on maîtrise aisément parce que :

- du point de vue externe, les objets peuvent être définis de façon claire et facilement compréhensible, et les détails concernant les structures de données, les traitements, etc... peuvent être cachés par des primitives d'accès de haut niveau.

- du point de vue interne, l'objet est vu comme un système fermé, autosuffisant, et sa construction ne doit pas tenir compte du contenu des autres objets.

Le revers de la médaille est que le langage Smalltalk se caractérise par une inefficacité congénitale, due à la gestion complexe de l'héritage des propriétés entre classes supérieures et inférieures, etc...

Il n'en reste pas moins que la vision des choses proposée par Smalltalk est très attrayante, et que bon nombre des concepts qu'elle propose peuvent être utilisés avec d'autres langages de programmation. On ne se privera donc pas de les utiliser dans la suite de ce travail.

7.2. Choix du langage de programmation.

Il peut sembler prématuré de parler dès maintenant du choix d'un langage de programmation alors que les spécifications du système sont loin d'être complètes.

Cependant, ce choix nous semble opportun, pour les raisons suivantes :

- On va devoir définir et désigner un certain nombre d'objets et de primitives. Le formalisme d'un langage de programmation est tout aussi indiqué qu'un autre à cet effet.

- De toute façon, toutes les définitions écrites dans un autre formalisme que le langage de programmation devront être traduites le jour où on choisira quel langage utiliser. Le choix "prématuré" du langage évite tout ce travail de recopiage.

- Finalement, si on veut construire un prototype rapidement, il faut pouvoir aboutir très vite à quelque chose de programmable. Autant connaître l'outil dont on disposera pour adapter tout de suite les algorithmes en fonction de ses possibilités.

Reste à déterminer quel langage utiliser.

Le langage LISP est très prisé par les tenants de l'"Intelligence" Artificielle, pour ses capacités de traitement de données symboliques ainsi que pour sa puissance d'expression élégante d'algorithmes complexes. Il permet d'implémenter aisément des algorithmes d'analyse syntaxique, exprimés par exemple sous forme d'Augmented Transition Networks [Winston].

Cependant, son utilisation sur VAX ne peut être que redoutablement inefficace. De plus, LISP ne fournit aucune facilité concernant la construction de systèmes modulaires : ainsi, toutes les variables internes aux différents objets devraient être implémentées sous forme de variables globales accessibles par tous.

Le langage PROLOG est également très utilisé actuellement. Il permet lui aussi d'implémenter aisément des analyseurs syntaxiques [Clock].

Cependant, comme LISP, il est très peu efficace sur des matériels inadaptés, et ne se prête que difficilement à la construction d'un système modulaire où chaque objet doit pouvoir disposer de ses propres variables, plutôt que d'une base de faits fourre-tout où chacun modifie ce qu'il veut.

Finalement, si on se tourne vers les langages procéduraux et que l'on cherche un langage efficace qui autorise la construction d'un système modulaire, on est obligé d'admette qu'actuellement, c'est le langage C qui s'impose tout naturellement [Kern].

On adoptera donc ce langage, d'autant plus qu'on dispose en librairie de toutes les routines qui ont été écrites par les concepteurs de RADAR, réalisé lui aussi en langage C.

8. Architecture générale du programme.

8.1. Décomposition en objets

En 3.4.3., on a établi un premier scénario du fonctionnement du programme, schématisé par la figure 3.4.:

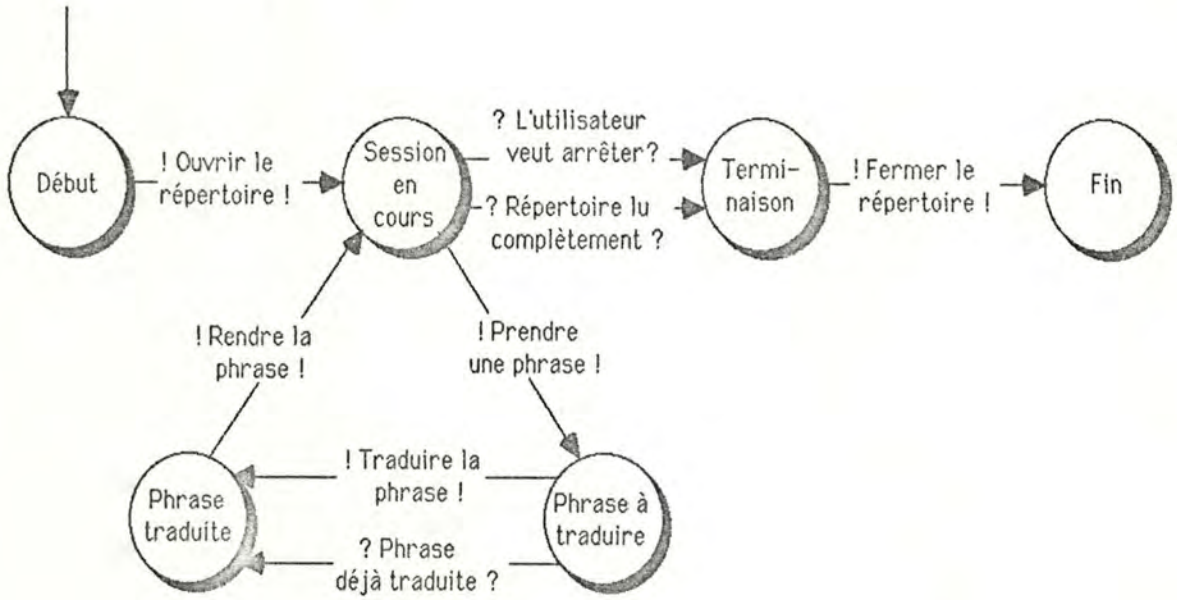


Figure 3.4 : Graphe des états d'une session de traduction.

La mise en oeuvre de ce scénario nécessite la construction des objets suivants :

- *Programme principal* : objet qui implémente l'automate de la figure 3.4.

- *Répertoire* : objet dans lequel on prend et remet les phrases et leurs traductions.

- *Traducteur* : objet à qui on donne une phrase et qui la rend traduite.

D'autres objets seront cependant nécessaires :

- *chaînes de caractères* : dans un programme manipulant le langage, les chaînes de caractères sont des objets fondamentaux. On élaborera un certain nombre de primitives standardisées pour faciliter leur manipulation.

- *éléments du répertoire* : le répertoire contient des phrases, assorties de traductions éventuelles. De plus, une phrase peut avoir un contexte, ensemble de phrases (et de traductions) qui ont un rapport sémantique et/ou grammatical avec elle. Un objet spécifique regroupera les structures de données et les primitives d'accès de ces éléments.

- *dialogue interactif* : si on désire pouvoir adapter facilement l'interface-utilisateur au fur et à mesure de l'évolution du programme, il faut localiser en un seul endroit les définitions et les primitives qui le concernent. De plus, cette localisation permettra de construire un interface homogène, donc sécurisant et convivial pour l'utilisateur.

8.2. Structuration des objets.

Une fois les différents objets définis, il faut les structurer pour pouvoir construire un programme. On peut proposer l'architecture de la figure 8.1, dans laquelle tout objet de niveau supérieur peut *utiliser les services* des objets de niveau inférieur.

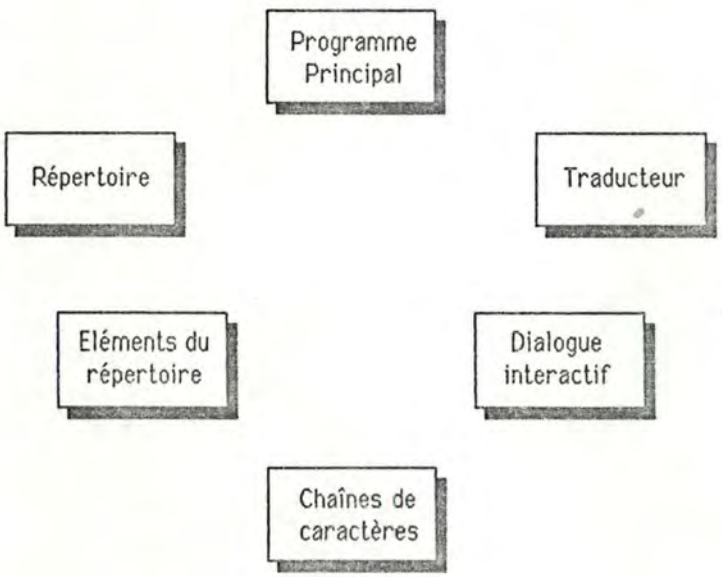


Figure 8.1 : Architecture générale du programme.

Reste maintenant à spécifier précisément ces différents objets, puis à les implémenter complètement en C.

On commencera par les objets de base de l'architecture, puis on remontera vers les objets de haut niveau. De cette façon, on disposera toujours des outils nécessaires à la construction de l'objet courant.

Les chapitres suivants exposent la définition externe de ces objets (structures de données et primitives). La définition interne, quant à elle, se trouve dans un fascicule séparé qui ne sera communiqué qu'aux membres du jury.

9. L'objet *chaînes de caractères*

9.1. Position du problème.

Le langage C propose une représentation très simple pour les chaînes de caractères : un tableau de longueur fixe dans lequel on met les caractères de la chaîne, suivis du caractère '\0' (code ASCII nul) servant de délimiteur de fin de chaîne.

Cette représentation a les avantages suivants :

- Les procédures qui lisent des chaînes de caractères ne doivent pas connaître la longueur maximum du tableau : elles reçoivent un pointeur vers le premier caractère, et lisent les caractères jusqu'à ce qu'elles rencontrent un '\0'.

- La gestion est facile et rapide : par exemple, pour mettre à blanc une chaîne de caractères, il suffit d'y ranger un '\0' en première position.

Elle présente cependant un inconvénient majeur : si on veut écrire dans une chaîne de caractères, on risque de dépasser ses bornes. Pour éviter cela, deux solutions sont possibles :

- Transmettre les chaînes de caractères et leur longueur maximum : cette solution, outre sa lourdeur et son inconfort d'utilisation, va à l'encontre de l'idée qu'une chaîne de caractères est potentiellement infinie.

- Définir des chaînes de caractères de taille "suffisemment grande" pour qu'on ne dépasse "jamais" leurs bornes : cette solution provoque un gaspillage conséquent de place-mémoire, sans garantir de toute façon qu'on ne dépassera vraiment jamais les bornes en question. Elle est donc extrèmement dangereuse, même si elle est très souvent utilisée par les programmeurs C.

Il est donc nécessaire de trouver une autre représentation des chaînes de caractères, qui garde les avantages de la représentation "standard", sans en avoir les inconvénients. Pour laisser les autres objets indépendants de cette représentation, on devra aussi construire des primitives d'accès spécialisées.

9.2. Définition externe de l'objet.

Dans ce point comme dans tous les points suivants, on utilisera les notations de C chaque fois qu'il faudra définir formellement une structure de données ou une en-tête de procédure. Bien que ces notations soient faciles d'accès, le lecteur qui n'est pas familiarisé avec elles peut se reporter à [Kern] pour en trouver une définition complète.

9.2.1. Structure de données.

Le type qui implémente la chaîne de caractères s'appellera *CdC*. Une CdC est une suite de caractères de longueur potentiellement infinie, et dont la longueur réelle vaut le nombre de caractères qu'elle contient effectivement.

On accède à ces caractères par leurs indices, allant de 0 à la longueur de la chaîne-1.

Le dernier caractère est toujours suivi d'un '\0', qui n'est pas compté dans les caractères de la chaîne pour ce qui est de la longueur de celle-ci. La seule façon d'indiquer la fin d'une chaîne est d'insérer un '\0' après son dernier caractère.

Exemple : la chaîne : "t o t o \0" est de longueur 4; son caractère d'indice 0 est 't', son caractère d'indice 1 est 'o', et ainsi de suite.

9.2.2. Primitives d'accès.

9.2.2.1. Initialisation de l'objet.

```
InitCdC();
```

Procédure qui initialise l'objet. A utiliser avant toutes les autres.

9.2.2.2. Longueur d'une chaîne de caractères.

```
LongChaine( ptChaine)
CdC *ptChaine;
```

Primitive qui renvoie la longueur de la CdC d'adresse *ptChaine*, c'est-à-dire le nombre de caractères de cette chaîne.

9.2.2.3. Initialisation d'une variable de type chaîne de caractères.

```
InitChaine( ptChaine)
CdC *ptChaine;
```

Procédure dont l'utilisation est obligatoire avant l'emploi de la CdC d'adresse *ptChaine*. Aucune autre procédure ne donnera de résultat cohérent sinon.

9.2.2.4. Abandon d'une variable de type chaîne de caractères.

```
AbandonnerChaine( ptChaine)
CdC *ptChaine;
```

Procédure à exécuter lorsqu'on ne désire plus utiliser la CdC d'adresse *ptChaine*. Après son exécution, toute référence à cette CdC sera interdite (sauf pour sa réinitialisation).

9.2.2.5. Prendre le ième caractère d'une chaîne.

```
GetCarChaine( ptChaine,indice)
CdC *ptChaine;
int indice;
```

Prend la valeur du *indice* ième caractère de la CdC d'adresse *ptChaine*. Si ce caractère n'existe pas, prend la valeur '\0'.

9.2.2.6. Mettre un caractère dans une chaîne à la position i.

```
PutCarChaine(car,ptChaine,indice)
char car;
CdC *ptChaine;
int indice;
```

Met le caractère *car* à la position *indice* dans la CdC d'adresse *ptChaine*. Si *indice* est supérieur ou égal à la longueur de la chaîne, la fonction insère des blancs aux positions intermédiaires.

Car peut avoir la valeur '\0'; insérer ce caractère à l'indice i signifie que l'on décide que la chaîne a une longueur i (ses indices vont de 0 à i-1).

La fonction prend la valeur du caractère *car* après son exécution.

9.2.2.7. Copier le contenu d'une chaîne dans une autre.

```
CopierChaine(ptSource,ptDest)
CdC *ptSource,*ptDest;
```

Remplace le contenu de la CdC d'adresse *ptSource* par celui de la CdC d'adresse *ptDest*.

9.2.2.8 Affecter une valeur "en extension" à une chaîne de caractères.

```
AffecterChaine(ValeurSource,ptChaineDest)
char ValeurSource[];
CdC *ptChaineDest;
```

Remplace le contenu de la CdC d'adresse *ptChaineDest* par celui du tableau ValeurSource (dont le dernier caractère significatif doit être suivi de '\0').

Exemples d'utilisation :

```
AffecterChaine("Ceci est une chaîne", &Chaine1);
```

Met dans *Chaine1* les caractères entre guillemets.

```
AffecterChaine("",&Chaine2);
```

Vide *Chaine2* de son contenu (sans toutefois abandonner la variable, qui est toujours référençable).

10. L'objet *éléments du répertoire*: définition externe.

10.1. Structures de données.

L'élément de base du répertoire est la phrase et ses traductions. Cette structure s'appellera *PhraseTrad* et contiendra trois chaînes de caractères : *Phrase*, *Trad1* et *Trad2*, respectivement la phrase et ses deux traductions.

Les PhraseTrad peuvent avoir un contexte, suite finie de PhraseTrad qui ont un rapport sémantique et/ou grammatical avec la PhraseTrad considérée. Cette structure s'appellera *ContextePhrase*, et contiendra un nombre maximum de *MaxContexte* éléments de type PhraseTrad. On accèdera à ces éléments par le biais d'un indice. Les indices des élémens d'un *ContextePhrase* vont de 0 à *MaxContexte*-1.

10.2. Primitives d'accès.

10.2.1. Primitives d'accès à une PhraseTrad.

10.2.1.1. Initialisation d'une PhraseTrad.

```
InitPhraseTrad(ptPhraTra)
PhraseTrad *ptPhraTra;
```

Initialise une variable de type PhraseTrad. Aucune autre fonction ne peut traiter correctement cette variable si elle n'a pas été initialisée.

10.2.1.2. Abandon d'une PhraseTrad.

```
AbandonnerPhraseTrad(ptPhraTra)
PhraseTrad *ptPhraTra;
```

Libère les emplacements-mémoire occupés par une variable de type PhraseTrad. L'utilisation ultérieure de cette variable est fortement déconseillée (sauf après réinitialisation).

10.2.1.3. Prendre la phrase d'une PhraseTrad.

```
GetPhrase(ptPhraTra,ptPhrase)
PhraseTrad *ptPhraTra;
CdC *ptPhrase;
```

Renvoie dans *ptPhrase la phrase contenue dans *ptPhraTra.

10.2.1.4. Prendre la traduction 1 d'une PhraseTrad.

```
GetTrad1(ptPhraTra,ptTrad1)
PhraseTrad *ptPhraTra;
CdC *ptTrad1;
```

Renvoie dans *ptTrad1 la traduction 1 de la phrase contenue dans *ptPhraTra.

10.2.1.5. Prendre la traduction 2 d'une PhraseTrad.

```
GetTrad2(ptPhraTra,ptTrad2)
PhraseTrad *ptPhraTra;
CdC *ptTrad2;
```

Renvoie dans *ptTrad2 la traduction 2 de la phrase contenue dans *ptPhraTra.

10.2.1.6. Prendre la phrase et les traductions d'une PhraseTrad.

```
GetPhraseTrad(ptPhraTra,ptPhrase,ptTrad1,ptTrad2)
PhraseTrad *ptPhraTra;
CdC *ptPhrase,*ptTrad1,*ptTrad2;
```

Renvoie dans *ptPhrase, *ptTrad1 et *ptTrad2 le contenu de *ptPhraTra.

10.2.1.7. Assigner un contenu à la phrase d'une PhraseTrad.

```
PutPhrase(ptPhrase,ptPhraTra)
CdC *ptPhrase;
PhraseTrad *ptPhraTra;
```

Remplace le contenu de la phrase de *ptPhraTra par celui de *ptPhrase.

10.2.1.8. Assigner un contenu à la traduction 1 d'une PhraseTrad.

```
PutTrad1(ptTrad1,ptPhraTra)
CdC *ptTrad1;
PhraseTrad *ptPhraTra;
```

Remplace le contenu de la traduction 1 de *ptPhraTra par celui de *ptTrad1.

10.2.1.9. Assigner un contenu à la traduction 2 d'une PhraseTrad.

```
PutTrad2(ptTrad2,ptPhraTra)
CdC *ptTrad2;
PhraseTrad *ptPhraTra;
```

Remplace le contenu de la traduction 2 de *ptPhraTra par celui de *ptTrad2.

10.2.1.10. Vider une PhraseTrad de son contenu.

```
ViderPhraseTrad( ptPhraTra)
PhraseTrad *ptPhraTra;
```

Met à blanc la PhraseTrad *ptPhraTra.

10.2.1.11. Copier le contenu d'une PhraseTrad dans une autre.

```
CopierPhraseTrad( ptSource,ptDest)
PhraseTrad *ptSource, *ptDest;
```

Remplace le contenu de *ptDest par celui de *ptSource.

10.2.2. Primitives d'accès à un ContextePhrase.

10.2.2.1. Initialiser un ContextePhrase.

```
InitContextePhrase( ptContexPhra)
ContextePhrase *ptContexPhra;
```

Initialise une variable de type ContextePhrase. Aucune autre fonction ne peut traiter cette variable correctement si elle n'a pas été initialisée..

10.2.2.2. Abandonner un ContextePhrase.

```
AbandonnerContextePhrase( ptContexPhra)
ContextePhrase *ptContexPhra;
```

Libère les emplacements-mémoire occupés par une variable de type ContextePhrase. L'utilisation ultérieure de cette variable est fortement déconseillée (sauf après réinitialisation).

10.2.2.3. Calculer la longueur réelle d'un ContextePhrase.

```
LgContexte( ptContexPhra)
ContextePhrase *ptContexPhra;
```

Renvoie la longueur de *ptContexPhra (nombre de PhraseTrad de ce contexte).

10.2.2.4. Définir la longueur réelle d'un ContextePhrase.

```
SetLgContexte( ptContexPhra,lgr)
ContextePhrase *ptContexPhra;
int lgr;
```

Assigne la valeur lgr à la longueur du contexte de ptContexPhra. Renvoie la valeur lgr après son exécution.

10.2.2.5. Prendre un élément d'un ContextePhrase.

```
GetEltContexte( ptContexPhra,pos,ptPhraTra)
ContextePhrase *ptContexPhra;
int pos;
PhraseTrad *ptPhraTra;
```

Range dans *ptPhraTra le contenu du pos^{ième} élément de *ptContexPhra.

10.2.2.6. Mettre un élément dans un ContextePhrase.

```
PutEltContexte( ptPhraTra,pos,ptContexPhra)
PhraseTrad *ptPhraTra;
int pos;
ContextePhrase *ptContexPhra;
```

Remplace le contenu du pos^{ième} élément de *ptContexPhra par celui de *ptPhraTra

11. L'objet *Dialogue*

11.1. Principes généraux.

Cet objet est nécessaire pour deux raisons :

- Facilité pour les autres objets : l'existence de procédures de dialogue de haut niveau les rendront indépendants des détails physiques propres au terminal utilisé, et leur fourniront des procédures "toutes faites" pour créer des menus et autres outils de dialogue du même genre.

- Uniformisation de l'interface et convivialité : toutes les procédures de gestion d'écran étant localisées dans le même objet, on pourra les implémenter de façon intégrée, afin de fournir un interface facile à apprendre et sécurisant pour l'utilisateur.

La méthode suivie sera la suivante : pour que l'interface soit facile à apprendre, il faut que la forme des messages apparaissant à l'écran soit standardisée, et que les messages de même nature aient la même présentation. Pour cela, on opèrera d'abord une classification des messages possibles selon leur nature. On créera alors des primitives propres à chaque type de message, et on imaginera une présentation suggestive pour chacun.

11.2. Classification des messages.

Ce point définit brièvement les différentes "familles" de messages possibles.

- Communication : Le programme communique un message à l'utilisateur et attend un accusé de réception avant de continuer. Ce type de message est ponctuel, ce qui signifie que l'écran ne doit pas nécessairement en garder de trace une fois que l'utilisateur a accusé réception du message.

  Exemple : "Bonjour. Ce programme traduit des répertoires homéopathiques."

- Passe-temps : Le programme affiche un message pour occuper l'utilisateur pendant qu'il effectue un calcul assez long. Le message reste affiché jusqu'à ce que le programme décide de l'effacer.

  Exemple : "Veuillez patienter..."

- Alerte : Le programme a détecté une erreur, un accident. Il le signale, et attend un accusé de réception avant de continuer. Le message est ponctuel, et l'écran ne doit pas en garder de trace dans la suite.

  Exemple : "Le fichier d'entrée se termine de façon anormale."

- Information : Le programme affiche une information quelconque : résultat d'un calcul, etc... Cette information est permanente, et l'écran doit pouvoir en garder la trace, afin que l'utilisateur puisse la consulter éventuellement dans la suite afin de prendre certaines décisions.

- Question : Le programme pose une question à l'utilisateur, et attend une réponse sous forme de chaîne de caractères. Ce type de message est ponctuel.

- Alternative : Le programme pose une question dont la réponse est "oui" ou "non". Ce type de message est ponctuel.

- Menu : Le programme propose plusieurs options, et l'utilisateur doit en choisir une. Ce type de message est ponctuel.

11.3. Définition externe de l'objet.

11.3.0. Introduction.

On va créer un sous-objet pour implémenter chacune des "familles" de messages définies ci-dessus. L'objet Dialogue lui-même sera constitué de ces différents sous-objets, ainsi que d'un certain nombre de primitives utilitaires dont ses composants hériteront.

11.3.1. L'objet *dialogue*.

Il disposera de deux primitives : InitDialogue( ) et ClôturerDialogue( ), qui initialiseront et clôtureront son utilisation. InitDialogue effectuera entre autres l'initialisation de tous les sous-objets.

D'autre part, on construira les procédures EcrireChaine(ptChaine) et LireChaine(ptChaine), afin de pouvoir respectivement écrire et lire une chaîne de caractères au terminal.

Enfin, on construira également la procédure EffacerEcran( ), qui nettoie l'écran du terminal.

11.3.2. L'objet *communication*.

On peut construire les primitives suivantes :

- InitCommunication( );

  Primitive qui initialise l'objet.

- Communiquer(ptMessage)

  CdC *ptMessage;

  Communique la chaîne de caractères d'adresse *ptMessage* à l'utilisateur, et attend un accusé de réception. Une fois l'accusé de réception reçu, efface toute trace du message à l'écran.

- StrCommuniquer(Message)

  Char Message[ ];

  Version alternative de Communiquer dont l'argument est un tableau de caractères terminé par '\0'[1]. Utile pour communiquer un message constant, comme par exemple : StrCommuniquer("Bonjour !").

[1] On appellera "string" un tel tableau.

11.3.3. L'objet *passe-temps*

Primitives :

- InitPasseTemps( );

  Initialise l'objet.

- AfficherPasseTemps(ptMessage)

  CdC *ptMessage;

  Affiche la chaîne de caractères d'adresse *ptMessage* et rend la main au programme appelant. Si un autre Passe-temps était déjà affiché, le détruit avant d'afficher le nouveau.

- StrAfficherPasseTemps(Message)

  Char Message[ ];

  Version de AfficherPasseTemps dont l'argument est un string.

- SupprimerPasseTemps( );

  Supprime le Passe-temps couramment affiché (s'il y en a un).

11.3.4. L'objet *alerte*

Primitives :

- InitAlerte( );

  Initialise l'objet.

- Alerter(ptMessage)

  CdC *ptMessage;

  Affiche la chaîne de caractères d'adresse *ptMessage*, et attend un accusé de réception. Une fois l'accusé de réception reçu, efface toute trace du message à l'écran.

- StrAlerter(Message)

  Char Message[ ];

  Version de Alerter dont l'argument est un string.

11.3.5. L'objet *information*.

Primitives :

- InitInformation( );

  Initialise l'objet.

- AfficherInfo(ptMessage)

  CdC *ptMessage;

  Affiche la chaîne de caractères d'adresse *ptMessage* et rend la main au programme appelant.

- StrAfficherInfo(Message)

  Char Message[];

  Version de AfficherInfo dont l'argument est un string.

11.3.6. L'objet *question*.

Primitives :

- InitQuestion( );

  Initialise l'objet.

- PoserQuestion(ptQuestion, ptReponse)

  CdC *ptQuestion, *ptReponse;

  Affiche la chaîne de caractères d'adresse *ptQuestion*, et renvoie dans * *ptReponse* la réponse de l'utilisateur. En fin d'exécution, efface toute trace de la question à l'écran.

- StrPoserQuestion(Question, ptReponse)

  Char Question[];

  CdC *ptReponse;

  Version de PoserQuestion dont le premier argument est un string.

- Str2PoserQuestion(Question, Reponse)

  Char Question[];

  Char Reponse[];

Version de PoserQuestion dont les deux arguments sont des strings.

11.3.7. L'objet *alternative*.

◇ Constantes :

```
#define OUI TRUE          /* Réponse positive ...                    */
#define NON FALSE         /* ... et négative à une alternative.      */
```

◇ Primitives :

- InitAlternative( );

  Initialise l'objet.

- ReponseAlternative(ptQuestion)

  CdC *ptQuestion;

  Affiche la chaîne de caractères d'adresse *ptQuestion*, et prend la valeur OUI si l'utilisateur répond "oui" à la question, NON sinon. En fin d'exécution, efface toute trace de la question à l'écran.

- StrReponseAlternative(Question)

  Char Question[ ];

  Version de ReponseAlternative dont l'argument est un string.

11.3.8. L'objet *menu*.

◇ Structure de données :

```
#define MaxMenuOptions ???      /* Nombre maximum d'options d'un menu    */
typedef struct MenuOptions      /* Un menu et ses options contient :     */
    {
        CdC Titre;              /* - le titre du menu,                   */
        CdC Options[MaxMenu];   /* - les options du menu,                */
        int NbOptions;          /* - le nombre d'options du menu         */
    } MenuOptions;              /* ... et s'appelle MenuOptions.         */
```

◇ Primitives :

- InitMenu( );

  Initialise l'objet.

- InitMenuOptions(ptMenuOptions)

  MenuOptions * ptMenuOptions;

  Initialise une variable de type MenuOptions.

- AbandonnerMenuOptions(ptMenuOptions)

  MenuOptions * ptMenuOptions;

  Abandonne une variable de type MenuOptions. Toute référence ultérieure à cette variable est illicite.

- ChoixOptionMenu(ptMenuOptions)

  MenuOptions * ptMenuOptions;

  Affiche les options du menu **ptMenuOptions* et renvoie la valeur de l'indice de l'option choisie (comprise entre 0 et ptMenuOptions->NbOptions). Une fois le choix effectué, efface toute trace à l'écran.

12. L'objet *Repertoire*.

12.1. Présentation intuitive.

Nous allons donner du répertoire à traduire une vue extérieure simplifiée et stable, via des primitives d'accès spécifiques. Il y a deux raisons à cela :

- d'abord, cette façon de faire permet de débarrasser les autres objets des détails de gestion des fichiers;
- ensuite, en cas d'utilisation ultérieure du programme sur d'autres textes-sources, les adaptations seront limitées à réécrire les primitives d'accès.

Le répertoire est un objet qui contient une suite de PhraseTrad, dotées d'un ContextePhrase (éventuellement vide). Les PhraseTrad d'un ContextePhrase sont ordonnées, dans l'ordre croissant des indices, de celle qui a le moins de rapports avec la PhraseTrad correspondante, à celle qui en a le plus.

Avant d'utiliser un répertoire, il faut l'ouvrir. Quand on a fini de l'utiliser, il faut le fermer. Cette utilisation consiste à le parcourir séquentiellement de la façon suivante :

- y prendre une PhraseTrad et son ContextePhrase
- y remettre cette PhraseTrad, éventuellement modifiée
- répéter ce couple d'opérations jusqu'à ce qu'il n'y ait plus de PhrasTrad, ou que l'on désire arrêter.

La figure 12.1 propose une représentation des différents états que peut prendre un répertoire, sous forme d'automate.

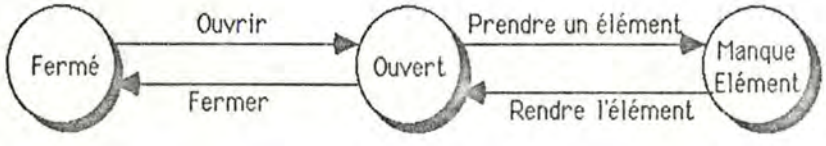


Figure 12.1 : Les différents états d'un répertoire.

12.2. Définition externe de l'objet.

On peut définir les primitives suivantes :

- OuvrirRepertoire( );

  Fonction qui initialise le répertoire. Renvoie les valeurs :

  - 0 si tout s'est bien passé,
  - 1 si le répertoire était déjà ouvert,
  - 2 si on n'a pas pu l'ouvrir à cause d'un problème de fichiers.

- FermerRepertoire( );

  Fonction qui clôture le traitement du répertoire. Renvoie les valeurs :

  - 0 si tout s'est bien passé,
  - 1 si le répertoire était déjà fermé,
  - 2 si le répertoire était ouvert, mais qu'il y manquait un élément,
  - 3 si on n'a pas pu le fermer à cause d'un problème de fichiers.

- PrendreElt(ptPhraTra,ptContexte)
  PhraseTrad *ptPhraTra;
  ContextePhrase *ptContexte;

  Fonction qui lit dans le répertoire l'élément suivant, et range ses divers composants dans les variables d'adresse ptPhraTra et ptContexte. Renvoie les valeurs :

  - 0 si tout s'est bien passé,
  - 1 si le répertoire ne contenait plus d'élément,
  - 2 si le répertoire était fermé,
  - 3 s'il était ouvert, mais que la dernière PhraseTrad prise n'y avait pas été remise
  - 4 si on a rencontré un problème de gestion de fichiers.

- RendreElt(ptPhraTra)
  PhraseTrad *ptPhraTra;

  Fonction qui remet dans le répertoire la PhraseTrad d'adresse ptPhraTra. Renvoie les valeurs :

  - 0 si tout s'est bien passé,
  - 1 si le répertoire était fermé,
  - 2 s'il était ouvert mais que l'on n'avait pas pris d'élément avant de vouloir l'y remettre,
  - 3 si on a rencontré un problème de gestion de fichiers.

13. Spécification externe de l'objet *traducteur*

Ce point sera évidemment un des plus brefs de tout ce travail, puisque le traducteur ne sera pas construit à l'issue du mémoire.

Du point de vue externe, le traducteur sera accessible par trois primitives : InitTraducteur, CloturerTraducteur et Traduire(ptPhraTra, ptContexte). On peut spécifier ces primitives comme suit :

InitTraducteur()

Procédure qui initialise l'utilisation du traducteur.

CloturerTraducteur()

Procédure qui clôture l'utilisation du traducteur.

```
Traduire(ptPhraTra,ptContexte)
PhraseTrad *ptPhraTra;
ContextePhrase *ptContexte;
```

Etant donné ptPhraTra, l'adresse d'une PhraseTrad à traduire, et ptContexte, l'adresse d'un ContextePhrase qui lui est associé, cette procédure renvoie dans *ptPhraTra une traduction de son contenu initial.

Afin de pouvoir tester le fonctionnement global des différents objets construits jusqu'ici, on effectuera une implémentation fictive du traducteur : les procédures InitTraducteur et CloturerTraducteur n'exécuteront aucune action, et la procédure Traduire renverra ses arguments inchangés.

14. L'objet *programme principal*

Ce point sera lui aussi très bref. Le programme principal est en effet très simple: d'abord, initialisation des autres objets, ensuite, session de traduction, et enfin, clôture des différents objets une fois la session terminée.

On peut détailler comme suit l'algorithme d'une session de traduction :

```
prendre une PhraseTrad et son contexte dans le répertoire;
s'il ne restait plus de PhraseTrad, terminer;
sinon :
   si l'utilisateur désire continuer la session :
         traduire la PhraseTrad;
         la rendre au répertoire;
         retourner à la première instruction;
   sinon :
         rendre la PhraseTrad au répertoire
         terminer.
```

Outre ces différents éléments, le programme principal fournit une fonction appelée StopErreur(message), dont l'argument est un tableau de caractères, et dont l'appel consiste à afficher son argument à l'écran, puis à provoquer l'arrêt du programme, après fermeture de tous les fichiers ouverts. Cette fonction permet de stopper tout traitement en cas d'incident grave au cours du fonctionnement du programme. Tous les objets héritent de la définition de cette fonction, et peuvent donc l'utiliser à leur guise.

Quatrième partie :

Eléments linguistiques.

15. Introduction de la quatrième partie.

Dans les chapitres qui suivent, nous allons présenter les principaux résultats des recherches linguistiques qui ont été menées dans le cadre de ce mémoire.

Après une classification des différents stades du processus de traduction d'un texte, nous détaillerons les problèmes posés par le premier de ces stades : l'analyse syntaxico-sémantique du texte-source. Nous montrerons les limites d'un modèle grammatical traditionnel, et proposerons l'adoption d'un autre modèle pour les travaux futurs.

Nous terminerons cette quatrième partie par un chapitre un peu "à part", concernant la construction et l'encodage d'un lexique anglais-esperanto adapté à la traduction.

Il faut ici attirer l'attention sur le fait que bien peu de ce qui est exposé dans cette partie n'aurait pu l'être sans la collaboration irremplaçable de Maryline Salmon. Les résultats qui vont être présentés sont en grande partie le fruit de ses patientes recherches.

16. Les différents stades de la traduction des répertoires.

Nous avons vu que pour traduire nos répertoires, nous utiliserions une technique relevant de l'*approche interlinguale*. Dans ce type d'approche, le processus de traduction peut se schématiser comme suit [Witkam] [Schubert] :

1. Analyse du texte-source, fournissant une description intermédiaire de sa structure.

2. Elaboration d'une représentation de cette structure en termes de l'interlangage.

3. Conversion de cette représentation en un texte exprimé dans l'interlangage.

4. Vérification de la correction grammaticale du texte ainsi obtenu.

→ On dispose alors d'une représentation fidèle du texte-source dans l'interlangage. Reste à effectuer, pour chaque langue-cible :

5. Analyse du texte en interlangage pour obtenir une description intermédiaire de sa structure.

6. Génération du texte en langue-cible à partir de cette description.

Dans le cadre de ce travail, la seule chose que nous ayons pu faire est d'entamer l'étude du stade 1 : "analyse du texte-source". Ce sera l'objet des points suivants.

17. Généralités sur l'analyse du langage.

17.1. Deux aspects complémentaires : syntaxe et sémantique. [Ruwet] [Dik]

La fonction essentielle du langage est de servir de support aux messages échangés par différents locuteurs. Tout langage possède nécessairement deux éléments :

- Un composant *syntaxique*, regroupant des règles d'utilisation, qui permet de représenter un message dans le langage.
- Un composant *sémantique*, regroupant des règles d'interprétation, qui permet de faire le lien entre la représentation d'un message et sa signification.

Bien que ces composants soient étroitement liés et qu'on ne puisse pas réduire la notion de langage à un seul d'entre eux, il est cependant possible de les étudier séparément afin de se simplifier la tâche. Dans les points suivants, nous nous limiterons à l'étude de l'aspect syntaxique.

17.2. Définition d'un langage. [Chom1] [Chom2] [Chom3] [Dik] [Leroy] [Ruwet].

17.2.1. Notions générales.

Si on fait abstraction de l'aspect sémantique des choses, on peut définir les notions suivantes :

- Une *chaîne de symboles* est une suite finie de symboles appartenant à un alphabet donné.
- Un *langage* est un ensemble de chaînes de symboles définies sur le même alphabet.

Un langage se caractérise en outre par des *lois*, qui définissent la façon dont les symboles de l'alphabet doivent être combinés pour obtenir des chaînes appartenant au langage. Ces lois peuvent être exprimées de façon formelle ou informelle.

La façon informelle est souvent utilisée pour définir les langues naturelles : il s'agit des grammaires traditionnelles, du genre de celle de Grévisse pour le français. Ces grammaires sont essentiellement destinées aux êtres humains. Elles consistent en de nombreux exemples et éxercices destinés à faire acquérir des automatismes qui, peu à peu, procurent une connaissance pratique de la langue. Mais elles ne sont en aucun cas adaptables au traitement automatisé des langages, et on ne s'en préoccupera donc pas dans la suite.

Du point de vue formel, il existe différentes façons de définir un langage :

- Définition *en extension* : liste exhaustive des chaînes du langage.
- Définition *grammaticale* : ensemble de règles dont l'application permet de construire toutes les chaînes du langage (et rien qu'elles). Cette définition est aussi appelée définition *générative* du langage.
- Définition par le *test d'appartenance* : algorithme permettant de décider si une chaîne quelconque appartient ou non au langage considéré.

Nous allons détailler brièvement ces différentes définitions.

17.2.2. Définition en extension.

Cette définition, par sa nature, est très limitée , et n'est que rarement utilisée. Il y a cependant une exception de taille, dans le domaine des langages naturels.

Une caractéristique de ces langages est de contenir des *mots*, chaînes de caractères particulières en nombre (relativement) limité qui entrent dans la construction de chaînes plus larges telles que les phrases et les textes.

Le sous-ensemble des chaînes de caractères constituant les mots d'une langue naturelle est défini en extension : c'est le *dictionnaire* de la langue.

On peut remarquer que cette façon de faire facilite l'analyse des langues naturelles : le dictionnaire peut être considéré comme un alphabet un peu particulier (ses symboles sont des mots); les chaînes du langage se définissent alors sur des mots, plutôt que sur des caractères. On travaille ainsi à un niveau d'abstraction plus élevé, mais toujours en termes d'alphabets et de chaînes de symboles.

17.2.3. Définition grammaticale.

Elle consiste à mettre en évidence un certain nombre de sous-ensembles des chaînes du langage (par exemple : groupes verbaux, prépositionnels, etc...), et à définir la façon dont ces composants peuvent se combiner entre eux.

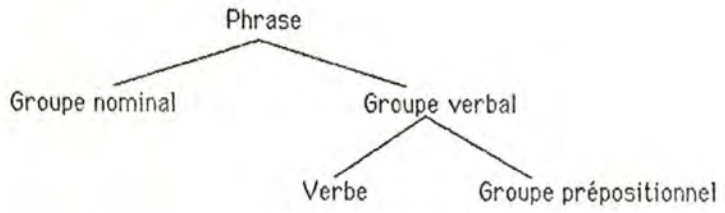
Exemple 1 :

"une phrase est un groupe nominal suivi d'un groupe verbal";

"un groupe verbal est composé d'un verbe et d'un groupe prépositionnel";

etc...

Ces règles permettent de décrire les chaînes du langage comme des structures hiérarchiques, que l'on peut représenter par des arborescences. Ainsi, les règles de l'exemple 1 donnent l'arborescence :



Ces règles sont en fait des *règles de production*, dont l'application permet de générer des chaînes de symboles.

Exemple 2 : soit la grammaire suivante (λ désigne une chaîne vide) :

GN = Art GAdj N

Art = le

GAdj = λ | Adj GAdj

N = monsieur chien train

Adj = joli gentil vieux méchant petit

Elle permet de générer des chaînes du genre :

le monsieur,

ou le train,

ou le gentil chien,

ou le méchant vieux monsieur,

ou le joli petit train.,

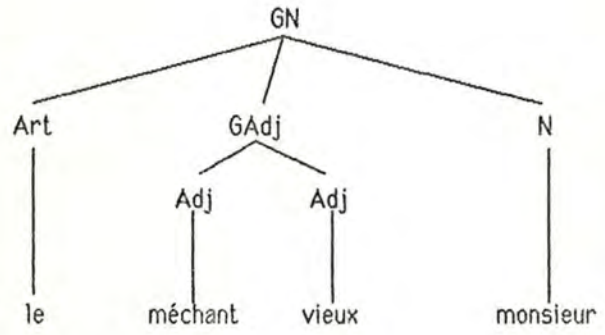
Notons qu'elle permet en outre de générer des chaînes telles que

le gentil méchant train,

ou le joli gentil vieux méchant petit joli monsieur.

Ces deux dernières phrases, sémantiquement dérangeantes, n'en sont pas moins syntaxiquement correctes du point de vue de la grammaire ci-dessus.

La phrase "Le méchant vieux monsieur" peut être représentée par l'arborescence suivante :



17.2.4. Définition par le test d'appartenance.

Elle est évidemment utilisée quand il s'agit d'analyser des chaînes afin de vérifier si elles appartiennent ou non à un langage. Elle doit en fait être dérivée de la définition grammaticale du langage.

Malheureusement, on peut démontrer qu'il n'existe aucun moyen systématique qui permette de passer de l'une à l'autre [Calc]. Tout au plus existe-t-il un certain nombre de techniques partielles, que l'on peut espérer appliquer avec plus ou moins de bonheur dans chaque cas particulier. On trouvera une description de ces techniques, et des algorithmes d'analyse correspondants, dans [Leroy], [Aho], [Gross].

Nous allons maintenant présenter les deux approches grammaticales que nous avons successivement étudiées dans le cadre de ce travail.

18. Choix d'un modèle linguistique.

18.1. L'approche de Chomsky.

Que l'on soit linguiste ou informaticien, on associe presqu'automatiquement le nom de Chomsky à l'idée de traitement automatique du langage naturel [Aho][Chom1] [Chom2] [Chom3] [Dik] [Ruwet]. Dans un premier temps, nous nous sommes nous aussi inspirés de la grammaire générative pour créer l'analyseur syntaxique des répertoires.

Nous disons bien *créer*; car, comme nous l'avons déjà mentionné, les répertoires sont rédigés d'une manière particulière, qui ne respecte en rien l'ordre "normal" des phrases anglaises.

Partant des catégories de mots définies par la grammaire traditionnelle (et que Chomsky adopte), nous avons observé quels mots se combinent pour former des *syntagmes*, et quels syntagmes se combinent pour former des *phrases*[1].

Nous nous sommes cependant heurtés à quelques problèmes, dont celui de la ponctuation, qui occupe une place capitale dans les répertoires. Celle-ci n'est pas toujours utilisée de manière conséquente. Les virgules remplissent plusieurs fonctions : celle de séparateur, de "coordinateur" ou encore de marqueur de dépendance entre éléments de niveaux différents dans l'arborescence. De plus, une même relation de dépendance peut être exprimée avec ou sans virgule...

Etant donné la difficulté d'intégrer cette ponctuation dans nos arborescences, qui sont fort tributaires de l'ordre dans lequel les mots apparaissent dans la phrase, nous nous sommes résolus à toujours considérer les virgules comme des séparateurs, et donc, à travailler sur des "phrases" minuscules.

Cette manière de procéder n'était en aucun cas gênante tant que nous analysions des "phrases" anglaises. De plus, cette analyse nous permettait de bâtir la description de la phrase mot à mot, au fil de la lecture. Cela représentait un certain gain de temps.

---

[1] On parle habituellement de *propositions*; mais nous préféré ne pas utiliser ce terme à cause de notre syntaxe particulière.

Mais les problèmes commencèrent à apparaître au moment du transfert vers l'Esperanto. Cette langue est flexionnelle : les adjectifs s'accordent avec les noms, les pronoms connaissent deux cas (nominatif et accusatif), l'accusatif est utilisé pour marquer l'idée de déplacement,... Bon nombre des informations nécessaires pour réaliser ces accords grammaticaux étaient totalement absentes de nos descriptions grammaticales. Il fallait donc encore découvrir - et écrire formellement - toute une série de règles qui pourraient pallier à ces inconvénients.

Ceci entraîna un autre problème non négligeable : malgré la concision des phrases prises en compte, nous aboutissions à une liste impressionnante de règles syntaxiques, dont l'utilisation devenait difficile du fait de leur nombre et des ambiguités qu'elles contenaient.

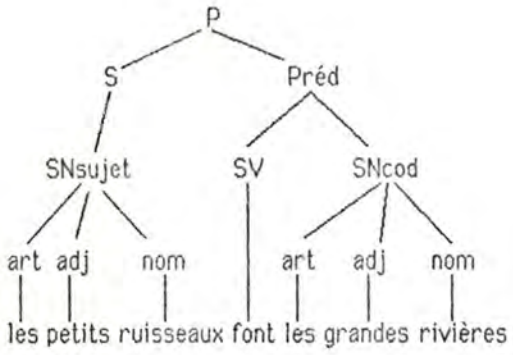
18.2. L'approche de Tesnière.

Nous en étions là de nos réflexions lorsque le Buro voor SysteemOntwikkeling (BSO) publia le rapport du Dr Klaus Schubert [Schubert] consacré à l'analyse syntaxique de l'Esperanto "amélioré" actuellement mis au point dans le cadre de leur projet DLT. Ce rapport nous mit sur la piste d'un autre linguiste, nettement moins connu que Chomsky et ses émules, mais dont les vues nous sont apparues d'emblée comme très intéressantes.

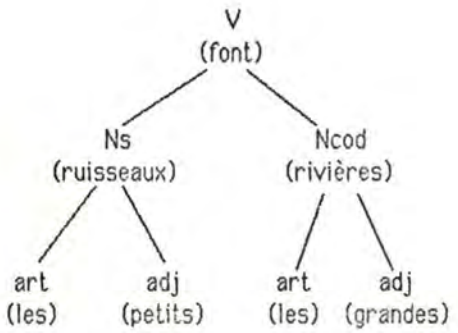
Ce linguiste est Lucien Tesnière, professeur (français) de langues slaves, qui a résumé sa théorie du langage dans un livre intitulé "analyse syntaxique" [Tesn]. Le principe de base de sa "syntaxe structurale" est sans doute mieux suggéré par la traduction anglaise : "dependency grammar".

L'analyse de Tesnière, symbolisée elle-aussi par des structures arborescentes, repose non sur les mots qui composent une phrase (analyse "morphologique"), mais sur les relations qui existent entre ces mots.

Ainsi, la phrase "Les petits ruisseaux font les grandes rivières", que Chomsky aurait décomposée comme ceci :



peut être représentée chez Tesnière de la façon suivante :



Sans vouloir nier la valeur des théories de Chomsky, il nous est apparu dès le début que la manière dont Tesnière appréhende le langage est beaucoup mieux adaptée au but que nous nous sommes fixé : non pas simplement traiter des textes anglais, mais surtout véhiculer leur contenu dans des langues multiples, dont les structures sont différentes.

En effet, comme le fait remarquer le Dr Klaus Schubert, la grammaire générative a surtout été pensée *en anglais* (langue maternelle de la plupart de ses promoteurs). Les arborescences qu'elle produit voient leurs feuilles presque toujours rangées dans l'ordre exact de la chaîne parlée anglaise.

Nous nous étions déjà posé la question de savoir comment respecter les flexions de l'Esperanto à partir de ces arbres anglais. A celà venait inévitablement s'ajouter la question de savoir comment respecter l'ordre des mots imposé par les diverses langues-cibles. Car si l'Esperanto, langue agglutinante, nous laisse relativement libres à ce sujet, les langues-cibles, elles, ont des principes rigides dans ce domaine[2].

---

2 Par exemple, rejet du verbe en néerlandais et en allemand.

Or, Tesnière, en considérant non pas l'ordre des mots, mais les relations de dépendance, s'affranchit de l'ordre de la chaîne parlée pour dégager des relations fondamentales, illustrées par le même schéma dans les langues source, intermédiaire et cible. La remise en ordre dans la langue-cible peut alors être effectuée au dernier moment, grâce à quelques règles de réécriture. On a en effet constaté que les différentes langues humaines, si elles diffèrent dans leur manière d'exprimer les choses, utilisent cependant toutes plus ou moins les mêmes relations de dépendance. Ainsi, une action connaît toujours un "agent" et un "subissant".

Un autre argument en faveur des théories de Tesnière est leur utilisation féquente dans les recherches en *linguistique contrastive*. Cette discipline linguistique relativement jeune étudie justement les problèmes que suscite le passage, lors de la traduction, d'une manière d'exprimer les choses à une autre.

Enfin, la préface à la deuxième édition du livre de Tesnière atteste que le succès de l'ouvrage est dû non pas tant aux linguistes, qu'à l'intérêt croissant des informaticiens pour les problèmes de traitement du langage naturel, intérêt qui les a amenés à apprécier l'applicabilité de la syntaxe structurale au traitement automatique du langage.

Tout ceci, ajouté à notre parti-pris de suivre de nouvelles voies de recherche plutôt que celles déjà explorées maintes fois -pour aboutir à des systèmes qui ne satisfont pleinement ni le traducteur, ni l'informaticien-, nous a poussés à reprendre à zéro le travail d'analyse des textes-sources. D'autant plus que nous étions soutenus par la perspective, si nous aboutissions à des résultats acceptables, de disposer avec le rapport du Dr Schubert d'une analyse déjà complète de la syntaxe Esperanto dans le même formalisme.

Dès le début de l'analyse des textes-sources selon les principes établis par Tesnière, nous avons entrevu une solution à l'un de nos principaux problèmes : la virgule.

En considérant les relations d'abord, et les mots qui prennent place aux noeuds de la structure ensuite, il est apparu que la virgule, quelle que soit sa fonction, pouvait tout simplement apparaître à un noeud de l'arborescence, facilitant ainsi la transposition

Esperanto et les accords nécessaires entre les mots. Il suffit d'établir les structures de dépendance possibles de la virgule.

Qui plus est, nous avons remarqué rapidement que ce type d'analyse pourrait s'appliquer avantageusement à l'analyse des mots Esperanto eux-mêmes. Nous l'avons dit, l'Esperanto est une langue agglutinante, c'est-à-dire qu'elle a tendance à accoler en un seul mot des éléments complémentaires plutôt que de les articuler autour de prépositions, verbes, etc... On peut raisonnablement postuler que les racines sémantiques des mots entretiennent avec leurs affixes et terminaisons des relations de dépendance de même nature que celles existant entre le mot principal d'une phrase et les autres. Ceci prendra toute son importance lorsque, dans l'avenir, on envisagera sérieusement de pouvoir sélectionner dans RADAR les symptômes des répertoires non plus sur leur forme, mais sur leur signification.

Les structures de Tesnière représentent ainsi pour nous une possibilité d'harmoniser les diverses approches analytiques qu'il nous faudra développer, tant du point de vue syntaxique (dans tous les cas envisagés) que du point de vue sémantique (sur les racines Esperanto).

Encouragés par ces premiers résultats, nous sommes allés plus avant dans l'étude de la syntaxe structurale. Tesnière, comparatiste chevronné, remet en cause la plupart de nos notions grammaticales, cherchant à établir ce que les linguistes appellent des *universaux de langage*, traits communs à toutes les langues naturelles. Sa façon d'appréhender le langage nous force à remettre en question bon nombre de nos acquisitions en matière de grammaire. L'assimilation de sa théorie prendra probablement un certain temps, mais nous croyons que ce retard sera comblé largement dans la deuxième phase de la réalisation du système, concernant le passage de l'Esperanto vers le néerlandais, le français ou l'italien, par exemple.

D'un point de vue technique, nous pouvons retenir les avantages et inconvénients suivants.

Le seul inconvénient réel décelé jusqu'à présent est que la méthode d'analyse de Tesnière nous impose de ne commencer l'élaboration d'une structure qu'après lecture de la phrase entière, sous peine d'entreprendre un trop grand nombre d'analyses parallèles consécutives à des hypothèses posées au fur et à mesure de la lecture. Ceci représente une perte de temps par rapport à une analyse "de gauche à droite", mais reste néanmoins négligeable si on songe à la brièveté de nos phrases.

En ce qui concerne les avantages, nous pouvons retenir l'harmonie de la représentation et de la technique d'analyse. Le fait aussi que le nombre de règles grammaticales sera beaucoup moins élevé que dans l'approche précédente : par exemple, pour l'Esperanto développé au BSO, le Dr Schubert a dénombré 10 types de mots différents, et 8 arborescences de base. Lorsqu'on songe aux dizaines de schémas grammaticaux différents que nous avions retenus comme représentatifs des répertoires lors de notre première analyse, on peut raisonnablement espérer un gain de compacité appréciable.

Avant de clôturer ce travail, il nous reste à dire quelques mots concernant le lexique anglais-esperanto qui devra être construit.

19. Lexique Anglais-Esperanto.

19.0. Introduction.

Lorsque la technique de traduction aura été suffisamment étudiée que pour être automatisable, il faudra disposer d'un lexique contenant les traductions esperanto des mots anglais. Afin de pouvoir commencer dès maintenant l'encodage de ce lexique, nous allons définir son contenu et des règles syntaxiques d'encodage.

On veillera à ce que ces règles permettent la construction aisée d'un analyseur syntaxique efficace.

19.1. Généralités.

Le lexique contient un ensemble de *rubriques* qui concernent chacune un mot anglais.

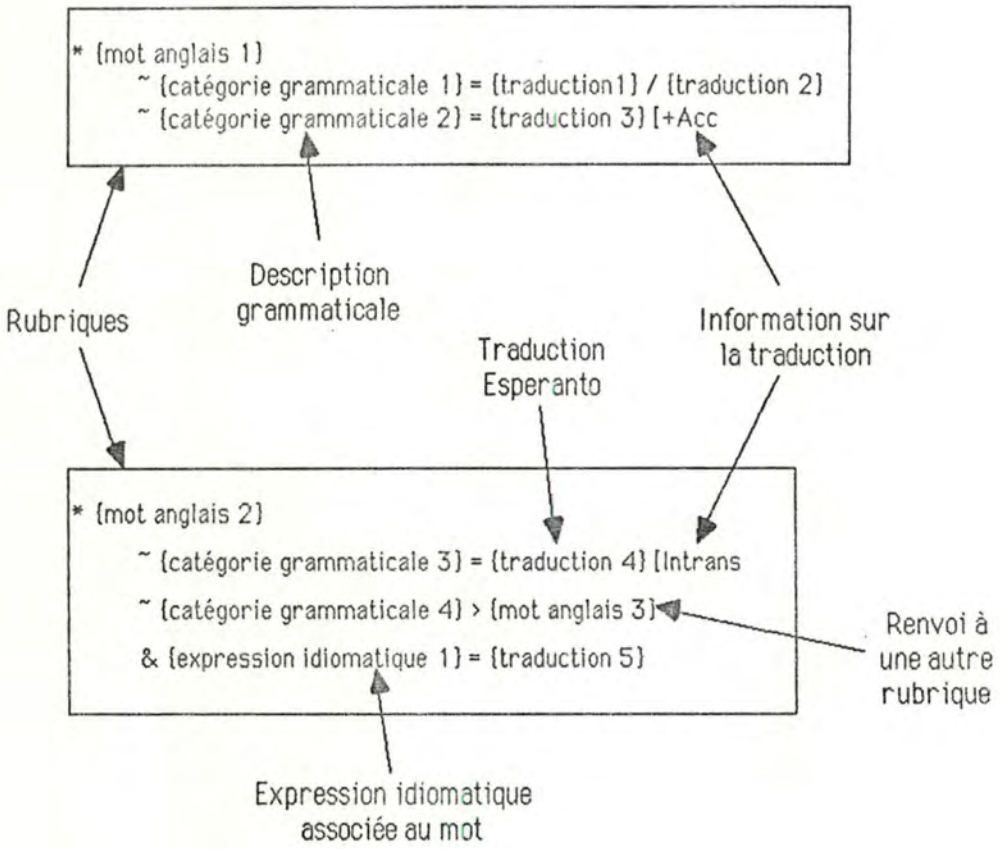
En général, un mot anglais peut avoir plusieurs *catégories grammaticales*. A chacune de ces catégories, correspondent une ou plusieurs *traductions* esperanto.

D'autre part, un mot peut intervenir dans un certain nombre d'*expressions idiomatiques*. Pour chacune de ces expressions que l'on associe à un mot, il faut connaître la traduction esperanto.

De plus, certains *mots* sont *dérivés* d'autres mots : par exemple, les formes conjuguées des verbes irréguliers. Il est inutile de reprendre dans le lexique la traduction de ces mots : une référence au mot dont ils sont dérivés est suffisante.

Enfin, certaines traductions esperanto nécessitent un *ajout explicatif* pour signaler, par exemple, que ce qui suit est obligatoirement un mot à l'accusatif, etc... On ajoutera dans ce cas l'information à la traduction concernée.

19.2. Exemples de rubriques.



19.3. Conventions typographiques.

Pour marquer...	Utiliser le caractère...
un début de rubrique	*
un début de description grammaticale	~
un début de traduction esperanto	=
la séparation entre deux traductions	/
un début d'information sur une traduction	[
un début de renvoi à une autre rubrique	>
le début d'une expression idiomatique	&

19.4. Format d'encodage.

Le contenu du dictionnaire peut être entré en format "libre", c'est-à-dire que l'on peut insérer tous les blancs, les tabulations, les sauts de lignes que l'on veut entre les différents éléments des rubriques, et aussi entre les rubriques elles-mêmes.

19.5. Remarque sur la typographie des mots Esperanto.

Afin de distinguer aisément les différentes racines constitutives des mots esperanto utilisés, sans perdre de place, on mettra en majuscule la première lettre de chaque racine:

Exemple : enterigi -> EnTerIgI.

19.6. Définition syntaxique précise du dictionnaire.

19.6.1. Notations utilisées.

- Un élément entre parenthèses est un élément non terminal de la grammaire, qui sera défini par une règle appropriée.
- Le symbole "|" signifie "**ou bien**"; il sert à séparer deux définitions grammaticales qui peuvent s'appliquer indifféremment mais pas en même temps.
- Le symbole "<--" signifie "**se définit comme**"; il sert à séparer les parties gauche et droite d'une règle.
- L'élément **(chaine)** désigne une chaîne de caractères non vide et qui ne contient aucun des caractères réservés : * ~ = /[> &.
- Dans la partie droite d'une règle, l'ordre des éléments est celui dans lequel ces derniers doivent se trouver dans le texte.

19.6.2. Exemple de règle et interprétation.

La règle :

(DICTIONNAIRE)<-- (RUBRIQUE) | (RUBRIQUE)(DICTIONNAIRE)

Peut s'interpréter :

"Un dictionnaire est constitué soit d'une rubrique, soit d'une rubrique suivie d'un nouveau dictionnaire" - càd : "Un dictionnaire est une suite non vide de rubriques".

19.6.3. Règles grammaticales.

(DICTIONNAIRE)<-- (RUBRIQUE) | (RUBRIQUE)(DICTIONNAIRE)

(RUBRIQUE)<-- (ENTREE)(PARTIE LITTERALE) | (ENTREE)(PARTIE LITTERALE)(PARTIE IDIOMATIQUE)

(ENTREE)<-- *(CHAINE)

(PARTIE LITTERALE)<-- (DESCRIPTION LITTERALE) | (DESCRIPTION LITTERALE)(PARTIE LITTERALE)

(DESCRIPTION LITTERALE)<-- ~(CATEGORIE GRAMMATICALE)(DESCRIPTION)

(CATEGORIE GRAMMATICALE)<-- ADV | PREP | VO | VED1 | VED2 | ...

(DESCRIPTION)<-- (RENVOI) | (DESCRIPTEUR)

(RENVOI)<-- > (CHAÎNE)

(DESCRIPTEUR)<-- =(TRADUCTION) | =(TRADUCTION)[(INFORMATION)

(TRADUCTION)<-- (CHAINE) | (CHAINE)/(TRADUCTION)

(INFORMATION)<-- + ACC | + NOM | INTRANS | ...

(PARTIE IDIOMATIQUE)<-- (DESCRIPTION IDIOMATIQUE)(PARTIE IDIOMATIQUE)

(DESCRIPTION IDIOMATIQUE)<-- &(CHAÎNE)(DESCRIPTEUR)

Conclusion

Ce mémoire était consacré à la traduction automatisée des répertoires homéopathiques du système RADAR.

Une étude de l'état de l'art en matière de traduction "automatique" nous a amené à choisir une technique de traduction *via un interlangage*, puis à décider que notre interlangage serait l'Esperanto. Cette décision, peu orthodoxe, nous paraissait cependant justifiée à plus d'un titre. De plus, nous choisissions avec elle une voie de recherche nouvelle, espérant par là obtenir de meilleurs résultats que les techniques plus classiques utilisées actuellement.

Notre travail se divisait en deux parties relativement distinctes :

- La programmation des éléments non linguistiques du programme de traduction.

- Une étude linguistique préparatoire à la traduction.

La première partie a consisté à réaliser complètement deux programmes utilitaires permettant d'uniformiser les différents formats que peuvent avoir les répertoires à traduire, puis à entamer la construction du programme de traduction proprement dit.

Pour ce programme, on a d'abord élaboré une architecture d'une modularité poussée, isolant au maximum les éléments dont la manipulation est fastidieuse, ainsi que les éléments susceptibles d'être modifiés dans l'avenir. Cette façon de travailler devrait faciliter -comme le souhaite Archimède- une adaptation future du programme pour la traduction d'autres textes que les répertoires, moyennant la réécriture de composants tels que le Répertoire et le Traducteur.

Dans son état actuel, ce programme fonctionne avec les restrictions suivantes :

- Son interface-utilisateur est implémenté de façon très rudimentaire. Cependant, on y accède via les primitives d'un composant spécialisé; les adaptations seront donc localisées, et limitées à la réécriture de ces primitives.

- Le composant de traduction n'est pas réalisé, et on se contente actuellement de simuler son fonctionnement.

Du point de vue linguistique, le travail préparatoire à la traduction s'est décomposé en deux parties : une étude lexicale et une étude syntaxique des répertoires à traduire.

L'étude lexicale a donné lieu à la création d'un lexique Anglais-Esperanto, actuellement partiellement encodé selon une syntaxe aisément exploitable dans la suite.

Pour ce qui est de l'étude syntaxique, nous avons d'abord tenté de mettre en pratique les théories de Chomsky afin de construire une grammaire de nos répertoires. Cela nous semblait possible vu la simplicité apparente des phrases à traduire. De plus, des algorithmes d'analyse syntaxique adaptés à ce type de grammaire sont désormais classiques, et nous espérions pouvoir les appliquer.

Cependant, cette approche était inadaptée, et après plusieurs semaines de travail, nous avons dû nous résoudre à l'abandonner pour nous tourner vers le linguiste français Tesnière, dont les théories semblent plus prometteuses. Nous n'avons malheureusement pas eu le temps d'aller très loin dans cette voie, qui reste donc ouverte pour les travaux futurs.

Pour conclure, on peut dire que ce mémoire fut essentiellement un travail préparatoire à la construction d'un système de traduction effectif. Le travail principal de nos successeurs consistera dans l'immédiat, du point de vue programmation, à construire un composant *Traducteur* (dont l'interface est déjà spécifié), et à l'inclure dans le système actuel. Ceci supposera, du point de vue linguistique, une étude plus approfondie des théories de Tesnière, afin de les amener à un niveau de formalisation tel qu'on puisse en tirer des algorithmes d'analyse syntaxico-sémantique. Une fois ce travail réalisé, on pourra alors espérer créer une version Esperanto des répertoires, qu'il s'agira ensuite de traduire vers différentes langues-cibles, en réutilisant au maximum les outils qui auront été construits.

Bibliographie.

Références bibliographiques citées dans le mémoire.

[Aho] : Aho, Ullman : the theory of parsing, translation and compiling. Vol 1 : Parsing. Prentice Hall.

[Allen] : Allen T.F.A. : The Encyclopedia of pure Materia Medica (12 vol.). B.Jain Publishers, New Delhi, 1977.

[AVL] : A. van Lamsweerde : cours de méthodologie de développement de logiciels. FNDP, Institut d'Informatique. Notes personnelles.

[Barthel] : Barthel H. & Klunker W.H. : Synthetic repertory (3 vol.). Karl Haug Verlag, Heidelberg.

[Calc] : Leroy H : Cours de théorie de la calculabilité.FNDP, Institut d'Informatique. Notes personnelles.

[Chom1] : Chomsky N : Structures syntaxiques. Seuil, Paris, 1969.

[Chom2] : Chomsky N : La nature formelle du langage. Seuil, Paris, 1969.

[Chom3] : Chomsky N, Miller G.A. : L'analyse formelle des langues naturelles. Gauthier-Villars, Paris.

[Clock] : W.F. Clocksin, C.S. Mellish : Programming in Prolog. Springer Verlag, Berlin-Heidelberg-New York-Tokyo, 1984 (seconde edition).

[Dik] : Dik S.C., Kooij J.G. : Beginselen van de algemene taalwetenschap. Uitgeverij Het Spectrum, Utrecht, Antwerpen, 1970-1977.

[Eurodic] : Articles sur le système Eurodicautom. Revue TERMINOLOGIE, 1981, n^{os} 38-40.

[Fich1] : Fichefet J., Jacques A., Gardin P., Paris J. : RADAR, un système expert à base de connaissances pour l'homéopathie. Research paper, Institut d'Informatique, Namur.

[Fich2] : Fichefet J., Jacques A., Gardin P., Paris J. : Homéopathie, cybernétique et aide à la décision multicritère. Research paper, Institut d'informatique, Namur.

[Gross] : Gross M. : Notions sur les grammaires formelles. Gauthier-Villars, Paris, 1967.

[Hering] : Hering C. : The guiding symptoms of our Materia Medica (10 vol). B.Jain Publishers, New Delhi.

[Jacques] : Jacques A. : Introduction à l'homéopathie Hahnemanienne. UIHN, Namur, 1983.

[Kent] : Kent J.T. : Repertory of the Homeopathic Materia Medica. Ehrart & Karl, Chicago.

[Kern] : B.W. Kernighan et D.M. Ritchie : Le langage C - Masson, Paris, 1983.

[Lavorel] : Lavorel B. : La traduction automatique à la Commission des Communautés Européennes - Expérience d'un traducteur. Revue LE LINGUISTE/DE TAALKUNDIGE, 1983, nos 4-5.

[Leroy] : Leroy H. : Cours de théorie des langages (matière approfondie).FNDP, Institut d'Informatique. Notes personnelles.

[Melby] : Melby A.K. : Machine translation with post-editing versus a Three-Level Integrated Translator Aid System. Revue TERMINOLOGIE, 1984, n° 45.

[Rich] : Rich E. : Artificial Intelligence. McGraw-Hill, 1984.

[Ruwet] : Ruwet N : Introduction à la grammaire générative. Plon, Paris, 1968.

[Schank] : Schank R.C. & Abelson R.P. : Scripts, Plans, Goals and Understanding. Erlbaum, Hillsdale, N.J., 1977.

[Schubert] : Schubert K. : DLT-Syntactic tree structures. BSO Research, Utrecht, 1986.

[Sowa] : Sowa J.F. : Conceptual structures : information processing in mind and machine. Addison-Wesley, 1984.

[Systran] : Articles sur le système Systran. Revues : TERMINOLOGIE, 1981, nos 38-40; TERMINOLOGIE, 1983, n° 44; TERMINOLOGIE, 1984, n° 45; LE LINGUISTE/DE TAALKUNDIGE, 1983, nos 4-5.

[Tesn] : Tesnière L. : Elements de syntaxe structurale. Ed. Kliniksieck, Paris, 1969 (2).

[Tucker] : Tucker A.B., Nirenburg S. : Machine Translation : a contemporary view. Annual Review on Information Science and Technology, vol 19, 1984.

[Winston] : Winston P.H., Horn B.K.P. : LISP. Addison-Wesley, 1981.

[Witkam] : Witkam A.P.M. : Distributed Language Translation : a multilingual facility for Videotex Information Networks (feasability Study). Buro voor Systeemontwikkeling (BSO), Uthrecht, Nederland, 1983.

Autres références non citées.

Andreyew N.D. : The intermediate language as the focal point of Machine Translation. In Booth, A.D. Ed. : Machine Translation. North Holland Publishing Co, Amsterdam.

Bateman R. : Linguistic tools are incidental to work in Machine Translation. Language Monthly, 1985, n° 21.

Chomsky N. : La linguistique cartésienne. Seuil, Paris, 1969.

Coulon D., Kayser D : Informatique et langage naturel : présentation générale des méthodes d'interprétation des textes écrits. TSI, vol 5, n° 2, 1986.

Garvin P.L. : Natural language and the computer. New-York, 1963.

Grévisse M. : Le bon usage. Grammaire française. Duculot S.A., Gembloux, 1964.

Hays D.G. : Introduction to computational linguistics. New-York, 1967.

Lecharlier B. : Réflexions sur la correction des programmes. Thèse de doctorat. Institut d'Informatique, Namur, 1985.

Lyons J. : Semantics. Cambridge University Press, 1977 paperback (3 vol).

Mounin G. : La machine à traduire; histoire des problèmes linguistiques. Den Haag 1964.

Nida E.A. : Language structure and translation. Stanford University Press.

Pigott J.M., Wheeler P.J. *et al.* : divers articles concernant la traduction automatique à la CEE. Terminologie, 1984, n° 45.

Remy C. : La traduction automatique - Dossier. Micro Systèmes, juin 1985.

Rollinger C.R. : Text understanding as a knowledge-based process. Research paper, Technische Universität Berlin - Institut für Angewandte Informatik. Berlin (RFA).

Schneider T. : Some notes on machine aids for translators. Meta, 28, 1983, n° 4.

Seleskovitch D. : La machine à traduire & la théorie de la traduction. Traduire, 1980, n^{os} 104, 105, 108.

Shannon C.E., Mac Carthy J. : Automation studies (1956).

Vithoulkas G. : La Science de l'Homéopathie. L'Esprit et la Matière, Rocher, Monaco, 1980.