MASTER EN SCIENCES INFORMATIQUES

Étude et réalisation d'un éditeur de texte

Debacker, Serge; Lestrade, Michel

Award date:
1981

Awarding institution:
Universite de Namur

FACULTES
UNIVERSITAIRES
N.D. DE LA PAIX
NAMUR

INSTITUT D'INFORMATIQUE

RUE GRANDGAGNAGE, 21, B - 5000 NAMUR (BELGIUM)

FACULTES UNIVERSITAIRES NOTRE-DAME DE LA PAIX - NAMUR,

INSTITUT D'INFORMATIQUE.

ETUDE ET REALISATION D'UN EDITEUR DE TEXTE.

Promoteur : C. Cherton.

Serge Debacker

Michel Lestrade

Mémoire présenté en vue de l'obtention du grade de

LICENCIE ET MAITRE EN INFORMATIQUE.

ANNEE ACADEMIQUE 1980 - 1981.

Nous tenons tout d'abord à remercier Monsieur C. Cherton qui a accepté de diriger ce mémoire et qui nous a permis, par ses conseils judicieux, de mener ce travail jusqu'au bout.

Nous exprimons également toute notre gratitude à Messieurs R.E. Buckley et M.S. Paterson qui nous ont accueillis à l'université de Warwick où ils nous ont aidés à réaliser un éditeur de texte.

Nous remercions tous les membres du personnel de l'institut qui nous ont soutenus dans ce travail et, tout particulièrement, Monsieur P. Gardin dont les avis éclairés nous ont constamment guidés.

Enfin, notre reconnaissance va à tous ceux qui nous ont aidés dans l'utilisation du PDP-11 de Namur, tant pour la mise au point des programmes que pour l'édition de ce mémoire. Plus particulièrement, nous remercions Messieurs J.P. Adans, Ph. Van Bastelaer et R. Verhaeghe.

TABLE DES MATIERES.

2. ANALYSE FONCTIONNELLE.

3. PRINCIPES DE REALISATION.

INTRODUCTION.

Ce mémoire aborde le problème de l'édition interactive de textes sur matériel informatique.

Dans une première partie, nous étudions ce domaine d'un point de vue général. Quatre concepts ressortent de la définition que nous y donnons d'un éditeur de texte : le texte, l'éditeur de texte, le dialogue et l'utilisateur. Ils sont analysés via la notion introduite de "potentiel".

A la lumière de cette étude générale, nous exposons dans la deuxième partie, les motifs qui nous ont poussés à choisir l'éditeur de texte implémenté. Comme nous le verrons, ces options sont dictées par les besoins de l' utilisateur concernant deux aspects : la définition du texte et des commandes.

Les trois parties suivantes définissent de manière précise les techniques retenues pour la phase de réalisation.

La première d'entre elles concerne les principes de représentation, d'accès et de gestion du texte en mémoire.

Vient ensuite la partie d'analyse " top-down " qui débouche sur une architecture de modules en plusieurs niveaux.

Enfin, la troisième de ces parties explique les caractéristiques du langage d'implémentation choisi et introduit les programmes repris en annexe.

Nous clôturons notre travail par une évaluation comparative des analyses fonctionnelle et organique ainsi qu'une synthèse des possibilités d'extension.

1. ETUDE GENERALE DE L'EDITION.

(Serge Debacker)

1. ETUDE GENERALE DE L'EDITION DE TEXTE.

1.1. INTRODUCTION.

Dans cette première partie, nous étudions le problème de l'édition de texte d'un point de vue général.

Nous commençons par présenter l'intérêt d'un tel travail (1.2.).

Ensuite, nous définissons la notion d'éditeur de texte (1.3.). Nous en retirons quatre concepts "clés" dont nous introduisons les propriétés en 1.5. .

Nous expliquons alors (1.4.) en quoi la méthode choisie pour cette étude générale se base sur la définition donnée en 1.3. .

Enfin, l'idée de potentiel (1.6.) permet d'analyser les quatre concepts majeurs (1.7. à 1.10) qui ressortent de la définition d'un éditeur de texte.

1.2. INTERET DE L'ETUDE DE L'EDITION DE TEXTE.

Dans un premier temps, nous soulignons l'étendue du problème de l'édition. Ensuite, nous expliquons les raisons pour lesquelles nous nous limitons à une étude de l'édition de texte.

Comme le souligne C.W. Fraser ([2]), les types de données le plus souvent éditées en informatique sont des textes mais peuvent être autre chose. Ainsi, des dictionnaires (directories) sont édités par les utilitaires qui détruisent ou renomment des

fichiers et les "interactive debuggers" éditent des données de forme binaire.

Dans ces exemples, Fraser ne perçoit pas les "directories" et les "données de forme binaire" comme des textes bien qu'on pourrait les considérer comme tels.

Mais il reste vrai que des données non textuelles sont éditées. Par exemple, les industries qui créent le profil de formes plus ou moins complexes peuvent recourir à la conception assistée par ordinateur pour éditer des dessins. Ses domaines d'application sont l'électronique, les industries automobile, aéronautique et navale, l'architecture et l'urbanisme, etc. .

Dans certains systèmes d'exploitation, les liaisons entre différents programmes avant leur chargement pour exécution sont réalisées par l'"éditeur de liens". Ou encore, un programme spécialisé d'un système d'exploitation effectue la sortie de travaux terminés qu'il choisit, sur base de critères, dans une file d'attente. C'est l'"éditeur de sorties".

Tous ces exemples montrent que le problème de l'édition est très vaste. Cependant, trois raisons limitent notre étude au domaine de l'édition de texte.

Tout d'abord, comme nous l'avons dit, le texte est un type de donnée fréquemment édité en informatique.

Ensuite, la littérature concernant l'édition de texte est essentiellement constituée de manuels pour utilisateur. Ce problème est donc intéressant à étudier vu le manque d'ouvrages

plus généraux qui traitent de cette question.

Enfin, l'emploi des éditeurs de texte se généralise plus que d'autres formes d'édition. Par exemple, la conception assistée de dessins par ordinateur est coûteuse et est moins utile pour certains secteurs (tertiaire : banques, assurances, ...) alors que l'édition de texte est financièrement abordable et utile dans les secteurs secondaire et tertiaire.

1.3. DEFINITION D'UN EDITEUR DE TEXTE.

Voici comment nous le définissons :

Un éditeur de texte est un programme qui aide un utilisateur à créer et/ou transformer un texte en temps réel.

Notons quatre concepts clés de cette définition :

1.3.1. Le texte.

Le texte est au moins envisagé comme une suite de caractères.

Cette définition minimale est toujours vraie puisque le caractère est un élément atomique d'un (ou de plusieurs) alphabet(s). Elle peut être étendue si les propriétés du texte sont considérées : structure de fond, structure de forme, etc. . Nous reprenons ces notions plus tard (cfr. 1.7.).

1.3.2. L'éditeur de texte.

L'outil informatique qui aide un utilisateur à construire un texte est communément appelé "éditeur de texte".

Nous ne précisons pas s'il s'agit d'un mécanisme hardware et/ou software car cette question ne se pose que dans le contexte d'une implémentation. Néanmoins, pour simplifier, nous le qualifions de "programme".

1.3.3. Le dialogue.

Un temps de réponse très court est une des conditions pour assurer un dialogue fructueux entre l'utilisateur et l'éditeur de texte. Le temps réel paraît donc adéquat pour le processus de questions-réponses. Celui-ci est possible grâce à une "interface" entre l'homme et la machine.

1.3.4. L'utilisateur.

C'est un homme (ou un groupe d'hommes) qui construit un texte. Il est clair que la notion d'être humain est générale. Il en existe nécessairement une typologie correspondant à l'étude des buts qu'il vise.

Remarque.

Dorénavant, et sauf cas explicite, les termes "éditeur de texte" et "utilisateur" sont vus dans le sens qui vient d'être

précisé.

1.4. DEMARCHE SUIVIE POUR L'ETUDE GENERALE.

L'étude générale de l'édition de texte a pour but de faciliter le choix de l'éditeur de texte que nous réalisons.

Cette étude préliminaire pourrait être abordée à quatre points de vue : l'éditeur de texte, l'utilisateur, le dialogue, le texte. Ce qui signifie que l'étude de chacun d'entre eux pourrait être une méthode d'investigation ~~aux~~ des concepts de l'édition de texte.

Choisir le concept "utilisateur" comme point de départ revient à se poser deux questions : Quelles sont les définitions possibles d'un texte pour l'utilisateur ? Quelles sont les actions possibles sur le texte qui peuvent l'intéresser ?

C'est effectivement pour cette voie que nous optons car l'utilisateur est le principal intéressé dans l'édition de texte. D'ailleurs, d'une part, si nous prenions le texte ou l'éditeur de texte comme point de départ, nous ne nous poserions qu'une des deux questions citées (la première pour le texte, la deuxième pour l'éditeur de texte) et d'autre part, la qualité du dialogue est plus une contrainte à respecter qu'un but à atteindre.

- - -

Notre démarche va donc consister à se demander ce que sont

pour l'utilisateur les aspects statique (qu'est-ce qu'un texte ?) et dynamique (quelles sont les actions sur le texte ?) de l'édition de texte.

La deuxième partie de cette question est ambigue. En effet, quoique leur but final soit la transformation de texte, il faut distinguer deux types d'actions, suivant leur but immédiat.

Certaines (exemples: insertion, suppression, ...) visent directement le contenu du texte dont la transformation est donc rapide.

D'autres (exemples: actions permettant à l'utilisateur de définir de nouvelles commandes) modifient les propriétés d'autres concepts clefs que le texte car ce n'est qu'à plus long terme que leur résultat sert pour transformer le texte. Ainsi, lorsque l'utilisateur se définit de nouvelles commandes (à condition, bien sûr, que l'éditeur le permette), il modifie les propriétés du concept "éditeur de texte" et non du texte qui, lui, n'est modifié que par après, lors de l'utilisation de ces nouvelles commandes.

Par conséquent, la question portant sur l'aspect dynamique de l'édition de texte devient: lors d'un processus d'édition de texte, quelles sont les actions que l'utilisateur peut apporter sur les concepts clés en vue de transformer le texte à court ou à long terme ?

- - -

Pour chaque concept clé, nous donnons une liste introductive de ses propriétés les plus significatives (point 1.5.) et nous disons comment l'utilisateur le voit et quelles sont les actions qu'il peut y porter (points 1.7. à 1.10.).

Bien sûr, quoiqu'étudiés séparément, ces concepts sont liés. Par exemple, la définition par l'utilisateur de la structure qu'un texte doit vérifier (apport au concept "texte") n'a d'intérêt que si elle débouche sur des actions sur le texte (apport au concept "éditeur de texte").

Entre-temps, nous montrons que l'utilisateur peut modifier les propriétés des concepts clés suivant certains niveaux (point 1.6.).

1.5. INTRODUCTION AUX PROPRIETES DES CONCEPTS CLES.

Dès maintenant, nous proposons, pour chaque concept clé, une liste très générale des propriétés les plus significatives en un temps quelconque d'un processus d'édition de texte.

1.5.1. Le texte.

- taille (en nombre de caractères)
- structure de forme
- structure de fond
- contenu
- sécurité

1.5.2. L'éditeur de texte.

- liste des commandes

1.5.3. Le dialogue.

- temps de réponse
- support physique (interface)
- vision du processus d'édition

1.5.4. L'utilisateur.

- droits sur le texte
- identification

1.6. NIVEAUX DE MODIFICATION DES PROPRIETES : POTENTIELS.

1.6.1. Introduction aux potentiels.

Conformément à la démarche proposée, nous pouvons étudier un processus d'édition de texte en nous interrogeant sur l'évolution que l'utilisateur peut apporter aux propriétés de chaque concept clé d'un éditeur de texte.

Nous voulons montrer maintenant que cette évolution est une modification de certains niveaux (que nous appelons "potentiels")

de propriétés.

1.6.2. Définitions.

Les propriétés de chaque concept clé peuvent être définies de sorte qu'à chacune d'elles soient associés, en tout temps t d'un processus d'édition, une valeur courante (valeur effective de la propriété au temps t) et un ensemble de potentiels (un potentiel est un ensemble de valeurs possibles) inclus strictement les uns aux autres.

En effet, si l'utilisateur peut changer directement ou indirectement la valeur d'une propriété, cette valeur appartient à un potentiel (disons "de niveau 1") qui, s'il peut lui-même être modifié par l'utilisateur, est sous-ensemble d'un potentiel strictement plus grand de niveau 2, et ainsi de suite de niveaux en niveaux. Pour généraliser, convenons d'appeler la valeur courante "potentiel de niveau 0".

A chaque propriété correspond, au temps t d'un processus donné d'édition de texte, le numéro de niveau (nombre entier positif ou nul) du potentiel le plus large.

Moins essentielle sans doute est la notion de variabilité de ce numéro. Mais pour être assez général, il faut admettre qu'il peut exister des propriétés pour lesquelles ce numéro est variable au cours d'un tel processus. Par exemple, pour une telle propriété, l'utilisateur qui réduit le potentiel de niveau i à celui de niveau i-1, diminue ainsi le nombre de niveaux d'une unité car le potentiel de niveau i est supprimé (l'inclusion doit

être stricte).

Pour toute propriété, nous avons la situation suivante :

fig. 1.1. : représentation de toute propriété de tout concept clé en termes de potentiels

potentiel de niveau 0 (valeur courante) :	potentiel de niveau 1 :	- - -	potentiel de niveau m :
valeur	ensemble 1	- - -	ensemble m

où m est fonction du temps et de la propriété,

et $\begin{cases} \text{valeur} \subset \text{ensemble 1}, \\ \text{ensemble 1} \subset \text{ensemble 2}, \\ \text{ensemble m-1} \subset \text{ensemble m}. \end{cases}$

1.6.3. Exemple.

Voici un exemple qui aide à mieux comprendre ces notions.

Supposons un éditeur de textes qui reconnait les structures "texte français", "programme ALGOL", "programme FORTRAN". Supposons également qu'au temps t d'une édition, le texte dont la structure est définie "texte français" soit celui-ci : " je suis un texte ".

Au temps t, la propriété "contenu" du concept "texte" se résume comme suit :

valeur courante :

"je suis un texte "

potentiel de niveau 1 :

"texte français"

potentiel de niveau 2 :

"texte français"

"programme ALGOL"

"programme FORTRAN"

1.6.4. Intérêt du concept de potentiel.

Plusieurs motifs prouvent l'intérêt de parler de potentiels.

1.6.4.1. Mise en évidence des niveaux de modification.

Par définition, la notion de potentiel met en évidence le fait qu'au cours de tout processus d'édition, toute action d'un utilisateur est une modification d'une (ou de plusieurs) propriété(s) suivant certains niveaux.

1.6.4.2. Aide à l'étude générale.

Quoique certaines propriétés (ex. : sécurité du texte) des concepts clés se prêtent moins à une étude par potentiels, l'idée de potentiel est malgré tout intéressante pour étudier l'édition de texte d'un point de vue général, c'est-à-dire sans quantification précise. En effet, nous espérons que le caractère ensembliste de la notion de potentiel assure une certaine généralisation.

1.6.4.3. Existence d'éditeurs travaillant par potentiels.

Bien que la plupart des commandes d'un utilisateur modifient des valeurs courantes (ex. : contenu de texte), certains éditeurs de texte permettent à l'utilisateur non seulement de dire avec quels potentiels de niveaux positifs il travaille, mais aussi d'en créer de nouveaux.

Un exemple type est donné par EMACS ([1] et [6]) qui permet à l'utilisateur de spécifier les ensembles (appelés modes) de commandes qu'il désire employer. Nous y reviendrons au point 1.8.3.3.2. .

Le même éditeur permet également de définir des commandes à l'aide du langage TECO ([10]).

1.6.4.4. Propriétés constantes et changeables.

La notion de potentiel permet de différencier deux types de propriétés.

Une propriété est constante si le numéro de niveau du potentiel le plus large est obligatoirement nul pendant toute la durée de tout processus d'édition. S'il peut être positif, la propriété est changeable.

1.6.4.5. Puissance d'un éditeur de texte.

Un autre intérêt du concept de potentiel est l'introduction à la notion de puissance.

Un éditeur de texte est d'autant plus puissant qu'il permet à l'utilisateur de modifier profondément et utilement les potentiels d'un nombre élevé de propriétés.

1.7. ETUDE DU TEXTE.

1.7.1. Structures de forme et de fond : définitions.

Nous savons déjà qu'un texte est une suite de caractères. En considérant ses propriétés, nous pouvons définir davantage.

En effet, que l'utilisateur en ait conscience ou non, tout texte vérifie une structure de forme et une structure de fond.

La structure de forme (ou plus simplement la "forme") est l'agencement physique des caractères du texte. Les problèmes de mise en page (soulignement, justification, etc.) relèvent de cette structure. Elle concerne donc surtout la morphologie du texte.

La structure de fond (ou encore le "fond") est l'agencement logique des caractères du texte. Elle porte donc plus sur les aspects syntaxique et sémantique.

Illustrons ces deux définitions par l'exemple d'un programme ALGOL. Du point de vue de la forme, l'utilisateur peut mettre en évidence la structure de blocs du programme en utilisant, par exemple, le principe de tabulateur. Et pour être correct du point de vue du fond, ce programme doit, entre autres, contenir autant de "begin" que de "end".

1.7.2. Intérêt d'une structure valide.

Pour l'utilisateur, l'intérêt qu'un texte ait une bonne structure de forme (c'est-à-dire ait la morphologie conforme à l'idée qu'il s'en fait) est non seulement la beauté de mise en page, mais aussi l'accentuation de la structure de fond. Ce renforcement peut garantir à l'utilisateur une meilleure perception sémantique et par-là une facilité dans les éventuelles modifications.

Dans l'exemple cité en 1.7.1., il est vraissemblable qu'un programme ALGOL où les blocs sont physiquement mis en évidence par le tabulateur est de prime abord plus facile à comprendre que le même programme totalement justifié à gauche.

Parfois, le respect d'une certaine forme est une contrainte. Ainsi, la disposition d'articles en colonnes dans les journaux nécessite la justification du texte.

- - -

En ce qui concerne la structure de fond d'un texte, l'intérêt pour l'utilisateur qu'elle soit bonne est évident : Ce n'est que si cette structure est correcte que le texte est susceptible de signifier quelque chose.

Pour revenir à l'exemple du point 1.7.1., une condition nécessaire (mais non suffisante) pour qu'un texte soit un programme ALGOL sensé est la présence d'autant de "begin" que de "end".

- - -

Notons enfin la possibilité d'indépendance des deux types de structures du point de vue de leur validité : l'une peut être correcte ou non, indépendamment de l'autre.

1.7.3. Référence et définition de structures.

Comme nous venons de le voir, l'utilisateur a intérêt à ce que son texte vérifie une structure (de forme et/ou de fond) conforme à l'idée qu'il en a.

C'est pourquoi certains éditeurs de texte lui permettent de référencer une structure (c'est-à-dire de nommer celle que le texte édité est sensé avoir) ou même d'en définir de nouvelles.

\- - -

Montrons par un exemple que la référence et la définition de structures modifient des potentiels de niveaux différents.

Supposons qu'au temps t1 d'un processus d'édition de texte, un utilisateur édite un texte dont la structure (qu'elle soit de forme ou de fond) est référencée "s2". Prenons aussi comme hypothèse qu'au même moment, il peut en référencer trois : "s1", "s2" et "s3".

En t1, la situation peut se résumer comme suit (fig. 1.2.).

fig. 1.2. : représentation de la propriété "structure" du concept "texte" au temps t1 de l'exemple :

valeur courante :	potentiel de niveau 1 :
"s2"	"s1" ∪ "s2" ∪ "s3"

Admettons qu'au temps t2 (t2 après t1), l'utilisateur achève de définir une nouvelle structure "s4" qu'il référence au temps t3 (t3 après t2) pour éditer son texte.

Les modifications apportées aux temps t2 et t3 peuvent se schématiser ainsi :

fig. 1.3. : représentation de la propriété "structure" du concept "texte" au temps t2 de l'exemple :

valeur courante :	potentiel de niveau 1 :
"s2"	"s1"∪"s2"∪"s3"∪"s4"

fig. 1.4. : représentation de la propriété "structure" du concept "texte" au temps t3 de l'exemple :

valeur courante :	potentiel de niveau 1 :
"s4"	"s1"∪"s2"∪"s3"∪"s4"

Par le choix de la représentation de la propriété "structure", toute référence modifie la valeur courante et toute définition accroît le potentiel de niveau 1.

Ces deux types d'action sur les structures n'ont de sens que si elles débouchent sur des actions sur le texte : vérification de la structure du texte par l'éditeur (ex. : contrôle de la syntaxe d'un programme ALGOL édité), accès de l'utilisateur au texte conformément à la structure référencée (ex. : accès au texte par mots si le texte est défini comme une suite de mots), ...

1.7.4. Liens entre fond et forme.

A certains égards, les propriétés "structure de fond" et "structure de forme" du concept "texte" sont liées.

D'une part, il existe des notions qui peuvent être difficilement rattachées à une seule des deux propriétés. Sans doute est-ce parce que de telles notions relèvent simultanément des deux structures ?

Par exemple, outre sa fonction physique de découpe du texte, le "paragraphe" n'a-t-il pas aussi une connotation sémantique lorsqu'il est considéré comme véhicule d'une idée homogène ?

D'autre part, quand un utilisateur communique à un éditeur de texte ad hoc la définition d'une structure de fond syntaxique, ne le fait-il pas au moyen d'une définition morphologique ? En d'autres termes, puisque la détermination de la syntaxe d'un texte est l'indication de toutes les séquences de caractères admises, une partie de la structure de fond (la syntaxe) est définie comme structure de forme.

D'ailleurs, un éditeur de texte ne peut vérifier si un texte est correct syntaxiquement qu'en analysant la suite des caractères du texte, c'est-à-dire en s'interrogeant sur l'aspect physique du texte.

1.7.5. Langage de définition de structures.

Il est clair que si l'utilisateur peut définir des structures, c'est qu'il dispose d'un "langage de définition de

structures".

Notons ici le compromis entre, d'une part, la richesse d'un tel langage, c'est-à-dire la possibilité de définir un nombre élevé de structures utiles, et, d'autre part, la rapidité, la simplicité et même la possibilité de l'éditeur de texte acceptant ce langage.

1.8. ETUDE DE L'EDITEUR DE TEXTE.

1.8.1. Avertissement : méthode.

Normalement, l' étude du concept "éditeur de texte" devrait se scinder en deux parties : une étude du texte et une étude des actions de l'utilisateur.

Néanmoins, puisque la première a déjà été vue au point 1.7., nous réduisons l'étude du concept "éditeur de texte" à la deuxième partie, ce qui ne diminue en rien le lien entre ces deux aspects (cfr. 1.4.).

En conséquence, nous nous intéressons maintenant aux actions que l'utilisateur mène lors d'un processus d'édition de texte.

1.8.2. Commandes : définition et types.

Pour exprimer des actions, l'utilisateur dispose d'un ensemble de commandes.

Nous pouvons définir une commande comme un ordre donné par un utilisateur à un éditeur de texte pendant un processus d'édition.

Elle se répercute sur ce processus par la modification d'un ensemble non vide de potentiels. Il est donc possible de classer les commandes suivant les propriétés des concepts clés qu'elles visent.

Les commandes les plus fréquentes portent sur le contenu (modification et recherche) et la sécurité du texte. Parfois, l'utilisateur peut modifier le comportement de l'éditeur, soit en référençant des structures et des modes, soit en définissant des structures, des modes et des commandes.

En toute généralité, nous devons admettre qu'il peut exister d'autres types de commandes. Toutefois, puisque les deux premiers sont très courants et que le troisième est intéressant à voir, nous nous limitons aux trois types cités. Nous les étudions aux points 1.8.4., 1.8.5. et 1.8.6. .

Auparavant, nous expliquons la notion de mode qui intervient dans beaucoup d'éditeurs de texte (1.8.3.).

1.8.3. Regroupements de commandes : modes.

1.8.3.1. Définition.

Un mode est un état du processus d'édition de texte dans lequel un utilisateur peut employer un ensemble donné de commandes.

1.8.3.2. Liens entre modes et potentiels.

Considérons un éditeur de texte avec lequel l'utilisateur peut désigner des modes.

Si, en un instant donné d'un processus d'édition, aucun d'entre eux n'a été explicitement spécifié comme mode courant, nous pouvons dire que l'utilisateur est occupé à travailler en "mode par défaut". EMACS ([1] et [6]) l'appelle "mode fondamental".

Admettant que n modes peuvent être référencés, convenons de la terminologie suivante :

mode 0 : mode par défaut,

mode 1, mode 2, ..., mode n : modes que l'utilisateur peut référencer,

ensmode i : ensemble des commandes dont l'utilisateur peut se servir lorsqu'il travaille en mode i ($0 \leqslant i \leqslant n$).

Il est alors facile de représenter la propriété "commande" du concept "éditeur de texte" en termes de potentiels :

fig. 1.5. : représentation générale de la propriété "commande" du concept "éditeur de texte" :

valeur courante	potentiel de niveau 1	potentiel de niveau 2
valeur	ensmode i	$\bigcup_{j=0}^{n}$ ensmode j

avec, en tout temps t,
$0 \leqslant i \leqslant n$ et valeur $\subset$ ensmode i.

1.8.3.3. Exemples.

Nous présentons ici deux cas d'éditeurs de texte qui concrétisent l'idée de mode.

1.8.3.3.1. Modes de l'éditeur $FSEDIT.

$FSEDIT, orienté écran, est un des trois utilitaires d'édition de programmes sur ordinateur IBM série/1 ([9]). Le raisonnement qui suit peut être simplifié pour les deux autres utilitaires $EDIT1 et $EDIT1N qui, eux, sont orientés lignes (cfr. distinction en 1.9.3.).

En mode par défaut, $FSEDIT présente à l'écran un menu qui offre à l'utilisateur le choix entre neuf commandes.

Les deux premières sont employées pour entrer dans les deux seuls modes possibles : BROWSE (par commande BROWSE) et EDIT (par commande EDIT). Les sept autres, qui ne sont pas des commandes de changement de modes, sont les suivantes : END, HELP, LIST, MERGE, READ, SUBMIT et WRITE.

Le mode BROWSE, qui permet de voyager à travers le texte, offre quatre commandes : END, FIND, LOCATE et MENU.

Quant au mode EDIT, il permet non seulement les mêmes fonctions que le mode BROWSE (d'où la question de l'intérêt de ce dernier mode), mais aussi de créer et de modifier des informations. Au total, 15 commandes sont possibles en mode EDIT.

Convenons des noms suivants :

mode 0 : mode par défaut (mode MENU),

ensmode 0 : {BROWSE, EDIT, END, HELP, ..., WRITE},

mode 1 : mode BROWSE,

ensmode 1 : {END, FIND, LOCATE, MENU},

mode 2 : mode EDIT,

ensmode 2 : {CHANGE, ..., MASK}.

Pour $FSEDIT, la propriété "commande" du concept "éditeur de texte" se schématise alors comme suit :

fig. 1.6. : représentation de la propriété "commande" du concept "éditeur de texte" pour $FSEDIT :

valeur courante :	potentiel de niveau 1 :	potentiel de niveau 2 :
valeur	ensmode 1	$\bigcup_{j=0}^{2}$ ensmode j

avec, en tout temps t :

$$\begin{cases} 0 \leqslant i \leqslant 2, \\ \text{valeur} \subset \text{ensmode } 1. \end{cases}$$

1.8.3.3.2. Modes de l'éditeur EMACS.

EMACS ([1] & [6]) permet à l'utilisateur de travailler avec les notions de modes "majeur" et "mineur".

L'utilisateur édite dans un et un seul mode majeur, qu'il choisit en fonction de la nature du texte : il peut référencer les modes "FUNDAMENTAL" (mode majeur par défaut), "LISP", "TEXT", ...

Les commandes se comportent différemment d'un mode majeur à un autre. Ainsi, puisque la disposition des commentaires dans un

programme dépend du langage de programmation, chaque mode majeur insère des commentaires différemment.

En outre, EMACS permet à l'utilisateur de spécifier des modes "mineurs" appelés ainsi car ils n'apportent que de légères modifications aux modes majeurs.

Tout mode mineur (ex. : mode "Autofill") peut être référencé ou abandonné par l'utilisateur indépendamment du mode majeur courant et des éventuels modes mineurs courants. En d'autres termes, en tout instant d'un processus d'édition, l'utilisateur peut travailler en 0, 1, ou plusieurs modes mineurs.

Nous pouvons donc représenter la propriété "commande" comme suit :

fig. 1.7. : représentation de la propriété "commande" du concept "éditeur de texte" pour EMACS :

valeur	potentiel de niveau 1 :	potentiel de niveau 2 :
valeur	un ensemble parmi ceux du potentiel de niveau 2	(commandes des modes :) - Fundamental - Lisp - Text - - - - - - - - - Text et Auto-fill - - - - - - - -

1.8.3.4. Intérêt des modes.

Le regroupement des commandes par nature, et donc l'existence d'autres modes que le mode par défaut, n'est pas indispensable à une édition de texte.

On pourrait imaginer que toutes les commandes d'édition puissent être appelées dans un même mode unique, qui serait le mode fondamental. Une telle situation se traduirait par la diminution d'une unité du nombre de niveaux de potentiels rattachés à la propriété "commande" du concept "éditeur de texte". La figure 1.5. deviendrait donc la figure 1.8. :

fig. 1.8. : présentation générale de la propriété "commande" du concept "éditeur de texte", avec mode unique :

valeur courante :	potentiel de niveau 1 :
valeur	ensmode

avec, en tout temps t, valeur $\subset$ ensmode.

Or, la pluralité des modes existe pour de nombreux éditeurs de texte. Dès lors, quel peut être l'intérêt pour l'utilisateur de spécifier des modes d'édition ?

Le premier avantage est la non redondance des informations qu'il donne chaque fois qu'il utilise une commande d'un mode donné. En effet, lorsqu'il travaille dans un certain mode, il ne doit pas accompagner chacune de ses commandes d'informations toujours identiques, à savoir celles qui sont inhérentes au mode courant.

Illustrons ce premier point par l'exemple du mode "TEXT & AUTO FILL" prévu par EMACS ([1] et [6]). Lorsque l'utilisateur travaille dans cet état, pour toute insertion de texte, le passage à la ligne suivante et le découpage éventuel des mots de fin de ligne sont automatiques puisque prévus par le mode lui-même. Effectivement, l'utilisateur ne doit plus demander ces actions

chaque fois qu'il commande une insertion.

L'existence de modes peut également rendre le processus d'édition plus clair aux yeux de l'utilisateur. Celui-ci peut être aidé par des regroupements judicieux des commandes suivant des critères qui l'intéressent.

Par exemple, l'utilisateur d'ED ([8]) peut passer en mode visuel s'il trouve plus facile d'éditer en voyant constamment à l'écran une fenêtre sur le texte, fréquemment mise à jour.

Dans certains cas, il peut être agréable pour l'utilisateur que les commandes soient regroupées suivant leur but. On pourrait imaginer un éditeur qui dispose d'un mode de modification de texte, d'un mode de recherche, d'un mode de communication (ex. : fusion de plusieurs textes), etc. .

1.8.4. Commandes portant sur le contenu du texte.

Tous les éditeurs de texte disposent bien sûr de commandes modifiant le contenu. La plupart d'entre eux permettent aussi à l'utilisateur de commander des recherches dans le texte.

Voyons rapidement ces deux types de commandes.

1.8.4.1. Commandes de recherche de contenu.

La plupart des éditeurs de texte disposent de commandes de ce type.

Leur emploi permet à l'utilisateur de demander à l'éditeur de le renseigner sur des occurrences d'une partie de texte.

Quant à leur intérêt, il est double : l'éditeur garantit un travail systématique et quasi instantané.

Un grand nombre de paramètres expliquent la multitude des formes possibles de recherche. Quelle est la définition du texte (paragraphes, lignes, mots, chapitres, strings, caractères, ...) ? L'utilisateur veut-il toutes les occurrences (recherche globale, partielle, ...) ? Comment la partie à trouver et les lieux de recherche sont-ils identifiés (numéros de lignes, strings, ...) ? Quels renseignements l'éditeur doit-il donner (impression des occurrences avec ou sans environnement, impression du nombre d'occurrences) ?

Pour un éditeur "orienté écran" (cfr. distinction en 1.9.3.), les commandes permettant de voyager à travers le texte peuvent être rangées dans les commandes de recherche : passage au caractère se trouvant au dessus, en dessous, à gauche ou à droite de l'endroit ou se trouve le curseur, passage à un autre écran (déplacement de la "fenêtre"), etc. .

Nous considérons que les commandes de recherche auxquelles sont greffées des possibilités de modification du contenu sont du deuxième type (1.8.4.2.).

1.8.4.2. Commandes de modification du contenu.

(par simplification, nous les appelons aussi "commandes

modifiantes")

Toute commande de ce type change la sèquence de caractères du texte et peut donc se définir comme une combinaison de suppressions et d'ajouts de suites de caractères.

Ce qui signifie que si l'on définit un texte comme une suite de lignes, une ligne comme une suite limitée (terminée par un délimiteur spécial) de caractères et un string comme une suite de caractères différents du caractère de fin de lignes, quatre primitives sont nécessaires et suffisantes pour définir toute commande modifiant du texte. Nous présentons ces quatre "modules de définition" ainsi que leur répercussion sur la taille (fig. 1.9.).

fig. 1.9. : primitives de définition des commandes modifiant le texte considéré comme suite de lignes :

	taille augmentée	taille inchangée	taille diminuée
(1) suppression d'un ensemble de lignes		X	X
(2) suppression d'un ensemble de strings		X	X
(3) ajout d'un ensemble de lignes	X	X	
(4) ajout d'un ensemble de strings	X	X	

Remarque : les croix indiquent des alternatives par lignes : la suppression et l'ajout laissent la taille inchangée en cas d'ensemble vide.

1.8.5. Commandes portant sur la sécurité du texte.

Lors d'un processus d'édition, certains effets non voulus par l'utilisateur peuvent entraîner une différence entre le texte tel qu'il aimerait qu'il soit et le texte tel qu'il est connu de l'éditeur.

Deux cas typiques sont la panne d'ordinateur et l'exécution de commandes que l'utilisateur, bien qu'il les ait envoyées, ne désire pas réellement (ex. : commande de suppression).

Un bon éditeur doit donc assurer la sécurité du texte en disposant de mécanismes qui permettent de réparer de tels accidents en garantissant la sécurité du processus d'édition lui-même.

Une méthode consiste à ce que l'utilisateur sauve son texte, c'est-à-dire demande à l'éditeur d'en garder une copie à l'abri de tout incident, après chaque commande modifiante. Ce mécanisme, qui garantit une totale sécurité, est parfois lourd car il engendre un temps de réponse trop long.

C'est pourquoi l'utilisateur préfère des sauvetages plus espacés dans le temps. Cette deuxième technique permet, en cas d'incident, de disposer d'une version du texte relativement récente. Ce seul principe assure une sécurité appréciable mais non absolue puisqu'un incident peut survenir après une commande modifiante qui a été envoyée à l'éditeur après le dernier sauvetage.

C'est la raison pour laquelle certains éditeurs tiennent

trace de la séquence des commandes qui suivent tout dernier sauvetage (il est moins coûteux en temps et en place mémoire de sauver une suite de commandes qu'une suite de textes). Ainsi, en cas d'incident, une édition ultérieure du même texte peut, si l'utilisateur le désire, débuter par l'exécution de la séquence des commandes sur la dernière version conservée du texte.

Parce qu'elle demande du temps, cette régénération peut être difficilement appliquée pour réparer l'effet d'une commande envoyée mais non voulue. Elle n'est d'ailleurs utile dans ce cas que si l'utilisateur a la possibilité de demander la régénération jusqu'à l'avant dernière commande. Pour ce deuxième type d'incident, d'autres techniques sont préférables. Notons à ce propos un principe très intéressant utilisé par EMACS ([1] et [6]) : chaque fois que l'utilisateur tue quelque chose qui est plus grand qu'un caractère, l'éditeur le sauve pour lui.

Remarquons aussi que la sécurité intervient à d'autres points de vue : sécurité par rapport aux autres utilisateurs, existence de différents types d'accès (généralement gérés par l'o.s. plutôt que par l'éditeur lui-même), ...

1.8.6. Commandes portant sur le comportement de l'éditeur.

Lorsque l'utilisateur définit à l'aide d'un éditeur adéquat une structure de texte ou une commande, il modifie le comportement de l'éditeur.

En effet, une telle définition fait que l'outil employé devient capable d'agir en fonction d'elle.

Dans le cas de la structure, il peut par exemple assurer un contrôle de validité du texte. Nous renvoyons le lecteur au point 1.7. qui traite du problème des structures de fond et de forme.

D'autre part, dans le cas de la définition d'une commande, l'éditeur s'enrichit _ et donc aussi son comportement _ puisqu'il devient apte à traiter toute référence à de nouvelles commandes.

Il est particulièrement utile qu'un éditeur permette ce deuxième type de définition car l'utilité des commandes dépend du type d'utilisateur.

Ainsi, l'insertion de commandes de formatage dans un texte est toujours identique pour la spécification des paragraphes du texte. Si ceux-ci sont nombreux, il peut être avantageux pour l'utilisateur de regrouper en une seule fonction la suite des caractères qu'il envoie pour chaque commande d'insertion.

En résumé, la définition de commandes ou de structures modifie le comportement de l'éditeur puisque celui-ci devient, grâce à elles, capable d'interpréter leurs références.

Il est clair que les problèmes concernant les langages de définition de commandes sont très intéressants et mériteraient d'être étudiés plus en détail, ce qui, malheureusement, semble hors de notre propos.

1.9. ETUDE DU DIALOGUE.

Le dialogue concerne les relations entre l'utilisateur et l'éditeur de texte. Plusieurs notions y sont donc associées : la

configuration, le temps de réponse, la vision du processus d'édition, ...

1.9.1. La configuration.

Le dialogue est possible grâce à un outil physique de communication. C'est sûrement l'évolution technologique de cet intermédiaire qui explique l'existence et l'intérêt des éditeurs de texte pour lesquels le temps réel est en effet plus adéquat que le traitement par lots.

Nous pouvons difficilement parler de cette interface en sortant du contexte global de la configuration tout entière.

Il existe par ailleurs une multitude de configurations possibles. Mais la plupart ont au minimum un clavier alphanumérique, un écran, une imprimante, un support de mémorisation et une unité centrale.

Nous ne mentionnons ici que deux systèmes très répandus.

D'une part, comme leur nom l'indique, les machines de traitement de texte ont pour tâche principale l'édition de texte. La majorité d'entre elles visent des utilisateurs non nécessairement informaticiens (ex. : secrétaires).

D'autre part, la plupart des ordinateurs interactifs permettent aux informaticiens d'écrire leurs programmes grâce à des éditeurs de texte appropriés.

1.9.2. Le temps de réponse.

Le temps de réponse est classiquement considéré comme la durée séparant l'instant où l'utilisateur envoie, de son terminal, une demande à un ordinateur central, et celui où il reçoit la réponse. Cette définition tient donc compte de l'échange des informations dans les deux sens et du temps nécessaire à la recherche et au traitement des données.

Ce temps de réponse est plus ou moins court suivant les configurations, les éditeurs de texte et, pour chacun d'eux, suivant les types de commandes. Ainsi, avec un éditeur pour lequel le texte est une suite de lignes identifiées par un numéro, il est fort probable qu'en moyenne, la recherche de toutes les lignes contenant un certain string est plus longue que la recherche d'une ligne d'un numéro donné.

Néanmoins, la qualité du dialogue exige un temps de réponse moyen inférieur à un certain seuil de tolérance, à fixer suivant certains critères (ergonomie).

1.9.3. La visualisation du processus d'édition.

Le processus d'édition est une suite alternée de transferts d'informations.

Dans un sens, l'utilisateur commande l'éditeur de texte qui, en retour, informe de l'état de l'édition. Les buts des données transmises sont donc différents suivant l'orientation.

La nature de ce processus permet de distinguer deux types d'éditeurs.

Les uns reconnaissent la ligne (suite de caractères) comme information minimale : les commandes et le texte sont introduits par lignes entières. C'est pourquoi ces éditeurs sont dits "orientés lignes". Citons EDIT (dec 2060), ED (PDP-11/45 UNIX).

Les autres mettent constamment à jour la vision à l'écran du texte édité et permettent à l'utilisateur d'indiquer, par positionnement du curseur, l'endroit où doivent agir les commandes. EMACS ([1] et [6]) est un exemple type de ces éditeurs dits "orientés écran".

1.10. ETUDE DE L'UTILISATEUR.

Une classification des utilisateurs en fonction de leurs objectifs sortirait du contexte de notre travail, mais pourrait être très intéressante dans l'étude des éditeurs de texte.

En effet, il est évident qu'à une typologie des utilisateurs correspond une classification des éditeurs puisque ces derniers s'adaptent aux besoins des premiers.

Par exemple, les "machines de traitement de texte" (cfr. 1.9.1.) offrent souvent différents types de logiciels adaptés aux besoins des utilisateurs (avocats, médecins, ...). De plus, ces machines sont fort différentes des éditeurs de programmes pour informaticiens car les utilisateurs ne sont pas les mêmes.

Rappelons ici que c'est en fonction de l'utilisateur que nous nous sommes préoccupés tout au long de cette deuxième partie des concepts clés et apparentés du problème de l'édition de texte (cfr. 1.4.).

Dans le même ordre d'idées, nous partons des besoins de l'utilisateur pour le choix et la réalisation de notre éditeur.

2. ANALYSE FONCTIONNELLE.

(Serge Debacker)

2. ANALYSE FONCTIONNELLE.

2.1. Introduction.

Après avoir étudié le problème de l'édition de texte d'un point de vue général, nous consacrons cette deuxième partie à la description précise de l'éditeur que nous choisissons de construire. Il s'agit donc d'une analyse fonctionnelle qui choisit le "QUOI" sans se soucier du "COMMENT" : nous définissons le rôle précis que l'éditeur doit jouer, sans décrire les techniques de réalisation utilisées.

Dans ce contexte, il paraît normal de décider tout d'abord à quel type l'éditeur ressort. C'est le but du point 2.2. .

Ensuite, nous justifions la distinction que nous adoptons entre création et édition (2.3.).

En 2.4., nous donnons la définition du texte pour exposer les procédés d'identification de parties de textes offerts à l'utilisateur dans le cadre de ses commandes (2.5.).

Enfin, les étapes 2.6. à 2.10 détaillent l'ensemble des commandes dont l'utilisateur dispose. Elles sont regroupées conformément à la classification de la première partie.

Nous présentons l'éditeur de manière dynamique, en ce sens que nous justifions progressivement nos choix pour arriver finalement au résumé de toutes les commandes (2.11.).

Souvent, nous utiliserons la notation de Backus ([11]) pour

définir certains concepts.

2.2. Choix du type d'éditeur.

L'existence d'une contrainte de temps pour notre travail nous limite forcément dans le choix de l'outil que nous implémentons. C'est pourquoi nous voulons construire un éditeur de textes qui, tout en disposant de commandes minimales, soit extensible.

Dans cette partie, nous présentons notamment l'ensemble de ces commandes de base. Elles sont implémentées et permettent l'édition de n'importe quel texte.

Nous voyons l'extensibilité à deux points de vue. En effet, nous nous donnons comme contrainte que dans la suite, nous puissions facilement, c'est-à-dire sans modification fondamentale de l'éditeur, à la fois redéfinir le format des commandes qui peuvent être étoffées, et créer d'autres commandes s'avérant utiles.

- - - -

A cette double contrainte s'ajoute celle de la qualité : pour satisfaire l'utilisateur, il est fondamental d'assurer un bon temps de réponse, la sécurité du texte, l'aspect mnémonique des commandes, etc. .

Remarquons que le contrôle sur le temps de réponse n'intervient pour la majeure partie que dans les phases ultérieures du processus informatique que nous suivons. Néanmoins, nous mentionnons déjà cette contrainte sur sa qualité car cette

exigence est une spécification fonctionnelle.

D'ailleurs, quoique ce soit surtout la qualité de l'analyse organique, de l'implémentation et du matériel utilisé qui l'explique, le temps de réponse de l'éditeur peut d'ores et déjà être estimé non satisfaisant si l'analyse fonctionnelle choisit mal la définition du texte et l'ensemble des commandes.

- - - -

Enfin, nous savons qu'un éditeur de texte peut être "orienté ligne" ou "orienté écran" (cfr. 1.9.3.). Pour deux motifs, nous plaçons notre éditeur dans la première catégorie.

D'une part, nous basant sur l'hypothèse qu'un éditeur orienté ligne est plus simple à réaliser qu'un éditeur orienté écran, nous préférons choisir le premier type , vu la contrainte de temps déjà évoquée.

D'autre part, un éditeur de ce type peut, s'il est approprié, être employé utilement sur un type quelconque de terminal, ce qui n'est pas vrai pour un éditeur orienté écran. Par exemple, nous voyons mal, bien que ce soit possible, un utilisateur dialoguant avec EMACS ([1] & [6]) au moyen d'un télétype comprenant une imprimante et un clavier.

- - - -

En résumé, nous nous proposons de réaliser un éditeur de texte orienté ligne, minimal mais extensible, et qui respecte certains critères (principalement : temps de réponse satisfaisant, commandes mnémoniques, sécurité du texte).

2.3. Distinction entre création et édition.

Expliquons les raisons qui nous poussent à distinguer la création de l'édition.

Au moment où l'utilisateur s'apprête à éditer, deux situations peuvent se présenter : ou bien le texte a déjà fait l'objet d'éditions précédentes, ou bien il est "nouveau".

La première possibilité signifie que l'utilisateur est sur le point d'éditer le texte contenu dans un fichier déjà existant. En effet, un texte peut être mémorisé d'une édition à une autre dans un fichier, c'est-à-dire une collection d'informations identifiable par un nom.

Dans ce cas, il est nécessaire que l'utilisateur donne comme paramètre le nom du fichier au moment de l'appel à l'éditeur.

Dans la deuxième hypothèse, le texte est édité pour la première fois. Comme l'utilisateur demande habituellement de sauver une (ou plusieurs) version(s) du texte en cours d'édition, nous convenons que, dès l'appel à l'éditeur, il donne le nom d'un fichier non existant qui est aussitôt créé et qui sert de fichier de sauvetage privilégié (nous verrons, en 2.9.1., ce que ce terme signifie) pendant l'édition.

Ces deux règles peuvent être résumées comme suit (fig. 2.1.).

fig. 2.1 : ordres d'appel à l'éditeur :

ordres d'appel :	fichier existant	fichier non existant:
ordre d'édition + nom de fichier	accepté	refusé
ordre de création et d'édition + nom de fichier	refusé	accepté

Pour chacun de ces deux ordres, l'utilisateur indique donc le nom d'un fichier. Celui-ci doit exister en cas de simple édition, alors qu'il ne le peut dans l'hypothèse d'une édition débutant par une création. Le contrôle de ces deux principes est assuré par l'éditeur.

Dans la suite, nous verrons plus en détail la répercussion de cette distinction sur la sécurité du texte (cfr. 2.9.). Néanmoins, nous pouvons déjà en dire quelques mots.

Supposons qu'en requérant une simple édition (i.e. sans création), l'utilisateur se trompe dans l'orthographe du nom du fichier. Dans cette éventualité, il est probable que le fichier référencé n'existe pas. L'utilisateur en est immédiatement averti et peut redemander une édition en rectifiant le nom.

De même, il paraît assez logique de refuser la création d'un fichier existant. Si l'on donnait un sens à une telle opération, une erreur dans la graphie du nom renseigné par l'utilisateur pourrait être fatale dans le cas _ rare mais possible _ où le nom mal orthographié serait celui d'un fichier existant. En effet, lorsqu'elle est permise, la création d'un tel fichier signifie la perte totale de son contenu premier.

2.4. Définition du texte.

2.4.1. Introduction aux concepts.

Nous savons qu'un texte est une suite de caractères. En

pratique, il serait très lourd de se baser sur cette seule définition pour offrir à l'utilisateur des méthodes lui permettant d'accéder à des parties quelconques de textes. Plus précisément, il est des cas où il lui serait fastidieux de définir en termes de caractères les portions de textes sur lesquelles il veut agir.

Cela ne signifie naturellement pas que l'accès au caractère lui est inutile, mais simplement qu'il doit aussi pouvoir spécifier des parties de texte moins atomiques, c'est-à-dire des agrégations de caractères.

Justement, l'idée de "ligne" peut, si elle s'accompagne à la fois de ses concepts associés _ "suite de lignes", "ensemble de lignes" _ et de la notion de "string", convenir à l'accès efficace de tout fragment de texte.

Bien sûr, d'autres groupements de caractères pourraient convenir, tels le mot, la phrase ou le paragraphe. Mais, comme le soulignent B.W. KERNIGHAN et P.J. PLAUGER ([3], p. 164), les lignes conviennent le mieux pour une grande variété d'applications, et représentent une organisation naturelle car intrinsèquement, le texte vient par lignes.

Grâce au choix des parties de textes adressables par l'utilisateur, n'importe quel texte peut être édité puisque le "string" et la "suite de lignes" lui permettent de référencer des ensembles respectivement très petits _ jusqu'au niveau du caractère et même de l'ensemble vide _ et très grands _ jusqu'au texte tout entier.

Il nous reste à présent à définir exactement ce que nous entendons par "ligne", "suite de lignes", "ensemble de lignes" et "string" (2.4.2).

2.4.2. Définition des concepts.

Désignons par ‹retour› le caractère de fin de ligne et par ‹blanc› le caractère blanc (espace).

Nous proposons alors la définition suivante d'une ligne :

‹ligne› ::= ‹string› ‹retour›

‹string› ::= ‹vide› |
‹caractère› |
‹string› ‹caractère›

‹vide› ::=
(‹vide› représente l'ensemble vide)

‹caractère› désigne tout caractère admis dans le texte, à l'exception du caractère de fin de ligne.
Nous préférons définir ‹caractère› en compréhension qu'en extension car il existe plusieurs alphabets possibles.

La suite de lignes se définit ainsi :

‹suite de lignes› ::= ‹vide› |
‹ligne› |
‹suite de lignes› ‹ligne›

Il est impossible de définir le concept "ensemble de lignes" au moyen du métalangage BNF ([11]) qui ne permet la définition que de séquences.

Un "ensemble de lignes" est l'ensemble, éventuellement vide, des lignes vérifiant une certaine condition. Ainsi, l'ensemble des lignes contenant un string donné.

2.5. Identification de parties de texte dans les commandes.

2.5.1. Choix du numéro comme identifiant de ligne.

Ayant défini les parties de texte accessibles par l'utilisateur, déterminons comment celui-ci peut les identifier.

Plusieurs raisons nous poussent à choisir le nombre entier comme identifiant de ligne.

En premier lieu, l'ordre croissant habituellement défini sur les nombres semble adéquat à l'utilisateur pour traduire la succession des lignes dans le texte.

Bien sûr, l'ordre basé sur d'autres caractères que les chiffres (par ex. les lettres) pourrait être facilement défini. Mais son emploi serait beaucoup moins naturel et moins familier pour l'utilisateur, tant en lecture qu'en écriture.

Nous verrons que par le choix du numéro comme identifiant de ligne, l'implémentation impose un nombre maximum de lignes identifiables. Néanmoins, le grand nombre de valeurs possibles

pour un entier (ex : 2 exposant 15) permet l'adressage d'une quantité suffisamment élevée de lignes.

2.5.2. Rapport entre numéro identifiant et ligne.

Il nous faut à présent choisir un lien entre le numéro et la ligne. Nous pouvons envisager deux possibilités : l'entier est soit un numéro d'ordre, soit une étiquette.

Dans le premier cas, la $i^{ème}$ ligne du texte est identifiée par la valeur i. Cela signifie que dans l'hypothèse de la destruction de cette ligne ou de l'insertion d'une ligne juste après elle, l'identifiant de chaque ligne du texte de numéro supérieur à i est respectivement diminué ou augmenté d'une unité.

La fréquence de telles variations en cours d'édition peut être une gêne pour l'utilisateur, et nous incite à rejeter cette première solution.

La deuxième éventualité veut dire qu'une ligne reçoit à sa création un numéro qui, en principe, ne change pas. En fait, nous verrons (2.6.) que l'utilisateur a intérêt à modifier l'"étiquetage" à certains moments. Mais cette variation ne doit s'avérer nécessaire que le plus rarement possible, sinon nous retrouvons le désavantage majeur de la première solution.

2.5.3. Utilité d'un pas.

Puisque l'insertion de lignes est très courante, il est préférable de prévoir des intervalles entre les numéros des lignes

consécutives. D'où la notion de pas : à toute ligne ajoutée en fin de texte est associé comme identifiant la somme du pas (nombre entier positif) et du numéro de la ligne qui devient l'avant dernière.

Le choix du pas par défaut résulte d'un compromis : il doit être assez grand pour permettre assez bien d'insertions, mais assez petit pour que le numéro maximum de ligne ne soit pas trop vite atteint. En ce sens, la valeur 10 peut convenir.

Initialement, les lignes sont donc numérotées 10, 20, 30, 40, etc. . Si l'utilisateur demande d'insérer par exemple trois lignes entre la deuxième et la troisième, l'éditeur calcule un pas "local" qui permet, là encore, une répartition optimale des numéros en vue de faciliter les éventuelles insertions futures. Dans notre exemple, les trois nouvelles lignes ont pour numéros 23, 26 et 29 (cfr. 2.7.1.1.2. pour le calcul du pas local).

2.5.4. Modification du pas.

Comme l'identifiant de ligne est considéré comme étiquette "ordonnée", il est normal que l'éditeur rejette toute demande d'insertion, entre deux lignes, d'un nombre de lignes plus grand que l'intervalle qui les sépare. Ainsi, l'utilisateur ne peut insérer trois lignes entre les lignes supposées consécutives et de numéros 2 et 5, puisque seules les deux étiquettes 3 et 4 sont libres.

Mais il est inconcevable de s'appuyer sur ce principe pour interdire pendant tout le processus des insertions pourtant

nécessaires à l'utilisateur. C'est pourquoi ce dernier dispose de la commande "n". Nous l'étudierons en détail en 2.6. .

2.5.5. Identificateurs de lignes spéciales.

Bien que ce soit suffisant, il semble lourd de n'offrir à l'utilisateur que le numéro comme moyen d'identification des lignes.

D'une part, avec ce seul mécanisme, l'utilisateur devrait toujours se souvenir des numéros des première et dernière lignes du texte. Une telle contrainte serait évidemment trop sévère. C'est la raison pour laquelle il peut désigner la première ligne par la lettre "f" ou "F" (pour "first") et la dernière par "l" ou "L" (pour "last"). Ces identificateurs spéciaux peuvent être utilisés chaque fois qu'un numéro de ligne peut l'être.

D'autre part, l'utilisateur agit souvent sur la ligne la plus récemment affectée par la commande précédente. Ici encore, pour lui éviter de la désigner par un numéro, l'éditeur tient trace de cette ligne privilégiée appelée "ligne courante". Pour la désigner, l'utilisateur peut employer la lettre "c" ou "C" (pour "current") en lieu et place de son numéro.

Enfin, les commandes qui ajoutent des lignes ont la particularité de les ajouter APRES une ligne spécifiée (cfr. 2.7.1.). Il faut donc donner à l'utilisateur le moyen de désigner une ligne FICTIVE qui précède le texte afin qu'il puisse adjoindre du texte avant la première ligne. Conventionnellement, nous donnons à cette ligne le numéro 0. L'utilisateur peut aussi

l'indiquer par la lettre "b" ou "B" (pour "begin").

2.5.6. Synthèse.

2.5.6.1. Identification d'une ligne.

Voici la définition finale d'un identificateur de ligne :

```
<identificateur ligne> ::= <entier> |
                           <identificateur spécial>

<entier> ::= <chiffre> |
             <entier> <chiffre>

<chiffre> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

<identificateur spécial> ::= b | B |
                             c | C |
                             f | F |
                             l | L
```

2.5.6.2. Identification d'une suite de lignes.

Rappelons tout d'abord la définition d'une suite de lignes (2.4.2.) :

```
<suite de lignes> ::= <vide> |
                      <ligne> |
                      <suite de lignes> <ligne>
```

A notre sens, l'utilisateur n'a aucun intérêt, dans le cadre de ses commandes, à identifier une suite vide de lignes appartenant au texte.

La référence d'une suite non vide de lignes est complète en spécifiant ses première et dernière lignes.

Par conséquent, en choisissant le caractère ":" comme séparateur des identifiants de lignes, nous arrivons à cette définition :

‹identificateur suite de lignes› ::=

‹identificateur ligne› |

‹identificateur ligne› : ‹identificateur ligne›

Exemples :

F : L

c

1000 : L

1000 : 1000 (ou plus simplement : 1000)

2.5.6.3. Identification d'un ensemble de lignes.

Un ensemble de lignes peut, en accord avec sa définition (2.4.2.), être identifié si la condition que doivent remplir ses lignes est exprimée.

Les commandes "f" ("find") et "s" ("substitute") portent sur des ensembles de lignes. Nous les étudierons en 2.8. et 2.7.2. .

2.5.6.4. Identification d'un string.

Nous verrons (2.7.2.) que, pour identifier un string, l'utilisateur doit donner explicitement la suite des caractères qui le composent.

Dans le format des commandes, nous convenons d'un caractère spécial ("'") qui indique la fin du string. Comme marque de début, le string est précédé soit du nom de la commande qui le contient, soit du caractère spécial "'".

Nous verrons que l'utilisateur identifie des strings dans les commandes "s" (2.7.2.) et "f" (2.8.).

2.6. Commande de modification du pas: n.

Comme nous l'avons vu, chaque ligne du texte est identifiée par un entier considéré comme étiquette (cfr. 2.5.1. et 2.5.2.).

Ce principe impose l'emploi d'un pas dans l'attribution des numéros identifiants, afin de prévoir des étiquettes libres utilisables pour de futures insertions entre toute suite de deux lignes consécutives (cfr. 2.5.3.).

La valeur 10 a été fixée pour le pas par défaut (cfr. 2.5.3.). Ce choix peut s'avérer gênant pour l'utilisateur qui désire souvent insérer plus de 9 lignes entre deux lignes successives du texte initial.

Or, seul l'utilisateur peut savoir si ce pas lui convient ou non. C'est pourquoi il doit disposer d'une commande qui lui

permette de changer lui-même la valeur du pas.

Mais cette possibilité ne résout pas tous les problèmes. Quelle que soit la valeur choisie, le nombre de numéros disponibles entre chaque ligne du texte et sa suivante est FINI.

Si toutes ces étiquettes sont attribuées par après, il devient impossible pour l'utilisateur d'intercaler des lignes dans certaines parties de texte. D'où le besoin d'une commande de renumérotation des lignes suivant le pas.

Pour illustrer ce deuxième souhait, prenons pour hypothèse qu'en un instant d'un processus d'édition, le texte se compose de huit lignes de numéros 10, 20, 23, 26, 27, 28, 29 et 30.

Dans cette situation, l'utilisateur ne sait pas introduire du texte entre, par exemple, les lignes 28 et 29.

S'il le veut malgré tout _ ce qui est d'ailleurs tout à fait concevable _ il doit marquer les lignes de nouveaux numéros.

Admettons qu'il les choisisse de 15 en 15. Les identifiants deviennent donc 15, 30, 45, 60, 75, 90, 105 et 120. L'insertion devient possible entre les lignes 60 et 75 (ex- 26 et 27), 75 et 90 (ex- 27 et 28), 90 et 105 (ex- 28 et 29) et, finalement, 105 et 120 (ex- 29 et 30).

- - - -

En résumé, l'utilisateur doit pouvoir spécifier deux types d'actions : d'une part, changer le pas lorsque sa valeur courante est souvent disproportionnée par rapport aux nombres de lignes

insérées et, d'autre part, renuméroter toutes les lignes pour permettre des insertions.

Nous pouvons considérer que lorsqu'il demande l'une de ces deux actions, l'utilisateur a intérêt à vouloir l'effet de l'autre.

D'un côté, si la modification du pas ne s'accompagnait pas d'une renumérotation de toutes les lignes, les intervalles entre les étiquettes des lignes existant avant la modification seraient différents des intervalles entre les nouvelles lignes insérées.

De l'autre côté, si la renumérotation des lignes s'effectuait sans modification du pas, l'inégalité inverse se produirait : les intervalles entre les numéros des lignes présentes avant la renumérotation seraient différents des intervalles entre les lignes introduites par après.

C'est pourquoi nous ne prévoyons qu'une seule commande qui non seulement modifie le pas, mais aussi renumérote la suite des lignes du texte.

Le format de cette commande "n" (pour "number") est :

n ‹int›

‹int› est le nouveau pas d'insertion (cfr. 2.11.2.).

2.7. Commandes modifiantes : a, c, d, i, m et s.

commandes :	noms:
a	append
c	copy
d	delete
i	insert
m	move
s	substitute

Nous avons présenté en 1.8.4.2. les quatre modules nécessaires et suffisants pour définir toutes les commandes de modification du contenu du texte édité.

Nous classons en deux groupes les commandes de ce type que nous avons implémentées. L'un porte sur les lignes (cfr. 2.7.1.), l'autre sur les strings (cfr. 2.7.2.).

Cette distinction s'explique par la différence de nature des concepts visés par les commandes modifiantes. D'ailleurs, le premier ensemble peut se définir complètement par les modules (1) et (3), tandis que le second peut être entièrement précisé grâce aux primitives de définition (2) et (4) (cfr. 1.8.4.2.).

2.7.1. Commandes portant sur les lignes : a, c, d, i et m.

Ici encore, nous distinguons les commandes. En effet, nous allons voir que certaines d'entre elles portent sur des lignes existant déjà dans le texte lors de leur appel (c, d et m) alors que les autres concernent des lignes qui ne sont écrites que juste après l'appel à ces commandes (a et i).

Avant de voir plus en détail l'ensemble de ces commandes, il faut noter que celles qui ajoutent des lignes en un endroit quelconque du texte _ à savoir a, c, i et m _ le font APRES une ligne spécifiée par l'utilisateur.

En effet, puisque toute suite de lignes est écrite une ligne après l'autre, autant garder la même convention et demander à l'utilisateur de spécifier le lieu d'ajout de texte APRES une ligne.

Rappelons que cette règle n'est pas limitative quant aux endroits d'ajout de texte puisqu'en ayant défini la notion de ligne fictive précédant le texte, nous rendons possible l'insertion **AVANT** la première ligne du texte (cfr. 2.5.5.).

2.7.1.1. Commandes visant les lignes existantes : c, d et m.

La commande "d" ("delete") permet de supprimer une suite quelconque de lignes du texte.

Son format est :

d ‹identificateur suite de lignes›

En particulier, "d f : L" détruit la totalité du texte.

Conceptuellement, l'utilisateur ne peut détruire que des lignes existantes. Ce qui explique que des messages d'erreur sont prévus s'il demande la destruction de la ligne fictive de numéro 0 ou la suppression d'une suite de lignes "inconnues" dans le texte et de numéros strictement positifs.

Nous disons qu'une ligne est inconnue si l'étiquette que l'utilisateur donne pour l'identifier n'a pas été attribuée.

Par extension, une suite de lignes définie par deux bornes (<identificateur ligne> : <identificateur ligne>) est inconnue si au moins une des deux lignes est inconnue.

La ligne fictive de début de texte n'est jamais inconnue puisque le numéro 0 lui est toujours attribué.

Par exemple, pour un texte non vide, "d 0 : L" est refusé car, bien que les lignes de numéros 0 et L soient connues, la première borne n'est pas strictement positive.

Si un texte ne contient que les lignes de numéros 10, 20, 30, 33, 37 et 40, les commandes "d 20 : 30" et "d 33 : L" sont admises, mais "d 31 : 37" est refusé.

Ce dernier exemple montre un intérêt de la notion de pas. Si, dans une commande "delete", l'utilisateur se trompe dans le numéro d'une ligne, la probabilité est grande que celle-ci soit inconnue et le risque est faible que l'éditeur détruise des lignes suite à une erreur de l'utilisateur.

- - - -

Souvent, l'utilisateur éprouve le besoin d'ajouter en un lieu du texte une suite de lignes existant déjà ailleurs.

Mais tantôt, il désire simplement leur transfert, en ce sens que la suite de lignes est enlevée du lieu où elle se trouve et ajoutée en un autre endroit. Tantôt, il souhaite la duplication :

il s'agit d'un recopiage.

Dans le premier cas, la taille du texte est inchangée. Dans le second, elle augmente _ ou reste égale si la suite est vide.

A ces deux types d'ajouts correspondent respectivement les commandes "m" ("move") et "c" ("copy") dont voici les formats :

m ‹identificateur suite de lignes› ^ ‹identificateur ligne›

c ‹identificateur suite de lignes› ^ ‹identificateur ligne›

où { ‹identificateur suite de lignes› = source,

‹identificateur ligne› = destination.

Si les contrôles (cfr. 2.11.2.) par l'éditeur montrent la possibilité d'exécution de ce type de commande _ il faut entre autres que le nombre d'étiquettes disponibles soit suffisant _ le choix du pas local d'insertion doit assurer une répartition adéquate pour les éventuelles insertions futures.

Par conséquent, la formule de calcul du pas local est la suivante :

$$\text{pas} = \text{entier}\left(\frac{\text{nombre d'étiquettes disponibles}}{\text{nombre de lignes à insérer}}\right)$$

où entier(x) = le plus grand entier inférieur ou égal à x,

nombre d'étiquettes disponibles = nombre
d'étiquettes entre la ligne
‹identificateur ligne› et sa suivante,

nombre de lignes à insérer = nombre
de lignes de la suite
‹identificateur suite de lignes›.

Ce même calcul est utilisé pour la commande "i" (2.7.1.2.), ainsi que les commandes "c" et "m" (2.7.1.1.).

2.7.1.2. Commandes visant les lignes non existantes : a et i.

Il est clair qu'avant de pouvoir utiliser les commandes modifiantes portant sur un texte existant (c, d et m et s), ce dernier doit être initialement constitué. C'est le but des commandes "a" ("append") et "i" ("insert").

Nous pensons qu'il est intéressant de les distinguer en raison de la différence d'endroit du texte visé. La commande "i" est employée pour l'ajout de texte avant la dernière ligne, tandis que "a" commande la constitution du texte après elle.

En effet, pour un octroi optimal des numéros en cas d'ajout avant la dernière ligne, nous demandons à l'utilisateur de fournir le nombre de lignes qu'il désire ajouter. S'il ne le connaît pas a priori, il peut demander un nombre élevé et arrêter l'insertion par l'envoi du seul caractère "." en première position d'une ligne.

Il en découle que pour ce type d'ajout, deux arguments doivent être donnés dans la commande : le lieu d'ajout et le

nombre maximum de lignes à insérer.

Par contre, en cas d'addition de texte après la dernière ligne, aucun argument ne doit être donné.

C'est pourquoi nous distinguons deux commandes :

i ‹identificateur ligne› ^ ‹int›

a

(cfr. 2.11.2.)

Pour l'exécution de la commande "a", le pas d'insertion est le pas courant : soit celui par défaut, soit celui fixé par la commande "n". Dans le cas du "i", cfr. 2.7.1.1. .

2.7.2. Commande portant sur les strings : s.

La commande "s" de substitution d'un string permet la modification de toute suite de caractères du texte.

Le format doit être tel que l'utilisateur puisse indiquer le string changé (‹string a›) et le string remplaçant (‹string b›).

‹string a›, puisque sensé déjà appartenir au texte, pourrait être identifié par les numéros d'ordre de ses premier et dernier caractères dans la ligne.

Or, il arrive que l'utilisateur demande le remplacement de toutes les occurrences du même string dans une partie de texte plus grande que la ligne. Dans ce cas, l'identification par bornes serait très lourde puisque les strings ne sont pratiquement

jamais aux mêmes positions de chaque ligne les contenant.

Par conséquent, nous adoptons une autre règle qui veut que l'utilisateur écrive explicitement le string changé. Ce principe n'est pas gênant car la longueur des strings _ d'ailleurs limitée par celle des lignes _ est généralement petite.

Quant à ‹string b›, l'utilisateur doit l'écrire explicitement dans la commande puisqu'il est nouveau.

La commande "s" se présente donc comme suit :

s‹string a›'‹string b›' ‹identificateur suite de lignes›

où ‹identificateur suite de lignes› désigne la suite de lignes dans laquelle s'effectue le remplacement de toutes les occurrences de ‹string a› par ‹string b› (cfr. 2.11.2.).

2.8. Commandes de recherche de contenu : f et p.

Nous connaissons le double avantage de telles commandes, à savoir le caractère systématique et rapide de leur exécution (cfr. 1.8.4.1.).

Nous nous proposons de choisir leur format en examinant successivement les quatre types de parties de texte adressables dans le cadre d'une recherche (cfr. 2.5.6.).

Notons dès à présent que puisque celle-ci est susceptible d'être infructueuse, un message d'erreur est prévu lorsque l'éditeur ne trouve pas le (ou les) occurrence(s) voulue(s).

2.8.1. Recherche d'une ligne identifiée.

L'identifiant d'une ligne est soit un numéro, soit un identificateur spécial (cfr. 2.5.6.1.).

Si l'utilisateur s'interroge sur le contenu d'une ligne dont il connaît le numéro, il doit pouvoir, en spécifiant celui-ci, demander à l'éditeur l'impression de la ligne en question.

C'est pourquoi la commande "p" ("print") existe sous la forme "p ‹identificateur ligne›".

De plus, il est fréquent que l'utilisateur recherche une ligne dont il ne connaît pas l'étiquette, mais dont il sait la présence dans le voisinage de la ligne courante.

Pour faciliter une telle recherche, nous convenons que le seul envoi du caractère de fin de ligne comme commande est interprété par l'éditeur comme l'ordre d'impression de la ligne courante, suivi de l'affectation à la variable "c" du numéro de la ligne suivante.

Ainsi, l'utilisateur prend connaissance de la suite des lignes suivant celle de numéro "c" par simples pressions successives de la touche de fin de ligne.

2.8.2. Recherche d'une suite de lignes identifiées.

En accord avec le principe de son identification (cfr. 2.5.6.2.), le format "p ‹identificateur suite de lignes›" permet à l'utilisateur de demander l'impression d'une suite de lignes.

Le format vu en 2.8.1. doit donc être étendu :

p ‹identificateur suite de lignes›

2.8.3. Recherche d'un ensemble de lignes.

Vu qu'il existe de nombreux types de conditions définissant des ensembles de lignes, nous pourrions définir plusieurs formats pour cette commande.

Ainsi, l'utilisateur pourrait demander l'affichage de l'ensemble de lignes :

- de numéros spécifiés dans une liste,

- dont le numéro est multiple du pas (pour désigner la partie de texte ne renfermant aucune ligne insérée, au cas où la commande "n" n'aurait pas été employée),

- contenant un string donné,

- etc.

Cependant, pour le format de la commande "f", ("find"), nous ne retenons que la dernière condition car c'est elle qui rend généralement le plus de services à l'utilisateur.

Mais nous l'étudions au point suivant qui traite spécifiquement de l'identification du string.

2.8.4. Recherche d'un string.

Le format de la commande "f" doit contenir au minimum un champ pour la désignation du string.

En plus, nous prévoyons une zone où l'utilisateur exprime la suite de lignes dans laquelle il demande la recherche.

2.8.5. Résumé.

Les commandes que nous venons d'examiner permettent donc la recherche de toute partie adressable du texte, grâce à la définition suivante des formats :

1) p ‹identificateur suite de lignes›

2)

(commande vide = commande sans caractère
avant ‹retour›)

3) f‹string›' ‹identificateur suite de lignes›

2.9. Commandes portant sur la sécurité du texte : v et e.

commandes :	noms :
e	end
v	save

2.9.1. Commande v.

Nous avons mis en exergue l'importance de ce mécanisme assurant la sécurité du texte (cfr. 1.8.5).

Pour des sauvetages espacés dans le temps, l'utilisateur dispose de la commande "v".

Puisque généralement il souhaite le recopiage du texte toujours dans le même fichier _ que nous appelons "fichier de sauvetage privilégié" _ il trouverait certainement agaçant de devoir passer à chaque fois le même nom de fichier comme argument à la commande "v".

Or, si nous concevons d'un fichier de sauvetage privilégié, l'éditeur, en recevant de l'utilisateur la simple lettre "v" comme commande, sait dans quel fichier particulier il doit sauver le texte.

Il nous faut donc nous mettre d'accord sur ce fichier de sauvetage par défaut. Dans cet esprit, rappelons-nous les deux appels possibles à l'éditeur (avec et sans création : cfr. 2.3.).

D'une part, si l'utilisateur demande une simple édition _ sans création _ du texte dont il passe le nom comme paramètre, c'est probablement que la version actuelle du texte ne l'intéresse plus. Dans cette optique, ce fichier peut être privilégié pour les sauvetages.

D'autre part, dans le cas d'une création, le même type d'arguments ne doit pas être fourni puisque le texte nouvellement édité n'existe pas encore dans le fichier. Il est donc possible de

fixer plutôt comme paramètre le nom d'un fichier qui, en cours d'édition, servira fréquemment pour les sauvetages.

En résumé, tout au long de l'édition, l'utilisateur peut, au moyen du format minimal de la commande "v", mettre le texte en sécurité dans le fichier de sauvetage privilégié, c'est-à-dire celui dont le nom est communiqué lors de l'appel à l'éditeur.

- - - -

Néanmoins, il peut être avantageux pour l'utilisateur de garder plusieurs versions d'un texte qu'il édite. Nous devons donc lui offrir un moyen de sauver le texte dans des fichiers différents.

C'est la raison pour laquelle des noms de fichiers peuvent être spécifiés en utilisant le paramètre optionnel ‹nom de fichier›.

Bien sûr, pour ne pas faire courir à l'utilisateur le risque de détruire des informations auxquelles il tient, il est nécessaire d'apporter des restrictions à la variable ‹nom de fichier›.

En particulier, l'éditeur refuse de transférer le texte dans un fichier existant avant le processus d'édition, sauf évidemment s'il s'agit, dans le cas d'une simple édition, du fichier de sauvetage privilégié.

Sans cette règle, si l'utilisateur désignait un ancien fichier important _ par exemple dont il oublierait l'existence _ comme lieu de sauvetage, le contenu pourtant essentiel serait

écrasé par le nouveau texte, et donc irrémédiablement perdu.

Remarquons que ce principe n'interdit pas l'usage répété d'un même fichier de sauvetage (privilégié ou non), ceci pour ne pas obliger l'utilisateur à retenir un nombre élevé de noms de fichiers dont un seul serait utile. D'ailleurs, la place qu'ils occuperaient serait trop importante.

- - - -

2.9.2. Commande e.

Supposons maintenant que l'utilisateur veuille arrêter l'édition. Il ne peut faire part de cette intention que par l'envoi à l'éditeur d'une commande _ que nous appelons "e" _ de fin d'édition.

Une simple question montre qu'elle est liée à la sécurité du texte : l'arrêt du processus doit-il être conditionnel à un sauvetage antérieur de l'état final (le contenu juste après la dernière commande modifiante) du texte ?

En fait, nous ne pouvons répondre une fois pour toutes à la place de tous les utilisateurs potentiels.

C'est pourquoi l'utilisateur entame un dialogue avec l'utilisateur chaque fois que celui-ci requiert l'arrêt de l'édition alors que l'état terminal du texte n'a pas été stocké dans le fichier de sauvetage privilégié. Cet échange permet à l'utilisateur de dire s'il souhaite vraiment l'arrêt.

Dans l'affirmative, le processus d'édition se termine. Dans

la négative, il continue normalement, ce qui veut dire que l'utilisateur peut employer n'importe quelle commande.

En l'occurrence, s'il avait oublié de demander le stockage du texte dans le fichier de sauvetage privilégié, il peut envoyer successivement les commandes "v" _ sous son format minimal _ et "e" qui entraînent la fin immédiate.

2.9.3. Résumé.

En résumé, les formats des deux commandes visant la sûreté du texte se présentent ainsi :

v [<nom de fichier>]

e

(voir 2.11.2.)

2.10. Commande d'aide : h.

Il peut arriver qu'en cours d'édition, l'utilisateur oublie les possibilités offertes par l'éditeur ou le format de certaines commandes.

A ces deux types de situation où l'utilisateur aimerait être renseigné, correspondent les deux formats pour la commande "h" (pour "help").

Si le premier est utilisé, c'est-à-dire si l'utilisateur ne donne aucun argument à la commande, la demande d'informations

porte sur l'ensemble des commandes disponibles en cours d'édition.

Si le deuxième format ("h <nom de commande>) est choisi, l'utilisateur requiert des précisions sur la commande spécifiée. Par exemple, si l'utilisateur commande "h f", l'éditeur fournit des indications sur "f".

Les renseignements donnés pour chaque commande concernent principalement son format et sa définition. Ce choix correspond au minimum garantissant la compréhension de la commande par l'utilisateur.

2.11. Tableau synoptique.

2.11.1. Appels à l'éditeur.

2.11.1.1. Création avec édition.

Format :

CREATE ‹nom de fichier›

Définition :

- ‹nom de fichier› est obligatoire et désigne un fichier sensé ne pas exister.

- Si ‹nom de fichier› est le nom d'un fichier n'existant pas, l'utilisateur travaille en mode éditeur (cfr. commandes). Le fichier ‹nom de fichier› est créé et sert de fichier de sauvetage privilégié. Sinon, un message d'erreur est envoyé à l'utilisateur et l'exécution de la commande se termine immédiatement.

2.11.1.2. Edition.

Format :

EDIT ‹nom de fichier›

Définition :

- ‹nom de fichier› est obligatoire et désigne un fichier sensé exister.

- Si ‹nom de fichier› est le nom d'un fichier existant, l'utilisateur travaille en mode éditeur (cfr. commandes) sur le texte initialement contenu dans ‹nom de fichier›.

- Sinon, un message d'erreur est envoyé à l'utilisateur et le processus se termine aussitôt.

2.11.2 Commandes d'édition.

Nom :

a - append

Format :

a

Vérification par l'éditeur :

- Format.

Définition :

- L'éditeur ajoute en fin de texte les lignes écrites par l'utilisateur. L'identifiant de chacune d'entre elles est la somme du pas et du numéro de la ligne précédente.

- Le processus se termine lorsque l'utilisateur écrit "." comme seul caractère en première position d'une ligne.

- B reste inchangé.

- C devient le numéro de la dernière ligne ajoutée.

- L devient le numéro de la dernière ligne ajoutée.

- F reste inchangé, sauf si le texte est vide avant l'appel à la commande, auquel cas F devient égal au pas.

Voir aussi :

- Autres commandes modifiantes : c, d, i, m et s.

Nom :

c - copy

Format :

c ‹identificateur suite de lignes› ^ ‹identificateur ligne›

Vérification par l'éditeur :

- Format.

- ‹identificateur suite de lignes› désigne une suite de lignes appartenant au texte.

- ‹identificateur ligne› spécifie une ligne existante.

- Le nombre de numéros disponibles entre, d'une part, la ligne ‹identificateur ligne› et, d'autre part, la ligne qui la suit, est supérieur ou égal à la quantité de lignes de la suite ‹identificateur suite de lignes›.

Définition :

- La commande c copie les lignes ‹identification suite de lignes› après la ligne ‹identificateur ligne›.

- B reste inchangé.

- C devient le nouveau numéro de la dernière ligne transférée.

- F est mis à jour si ‹identificateur ligne› désigne la ligne fictive de début de texte.

- L devient le nouveau numéro de la dernière ligne de la suite ‹identificateur suite de lignes› si ‹identificateur ligne› vise la dernière ligne du texte.

Voir aussi :

- Autres commandes modifiantes : a, d, i, m et s.

Nom :

d - delete

Format :

d ‹identificateur suite de lignes›

Vérification par l'éditeur :

- Format.

- ‹identificateur suite de lignes› identifie une suite existante de lignes dans le texte.

Définition :

- Les lignes identifiées par ‹identificateur suite de lignes› sont détruites.

- B reste inchangé.

- F est mis à jour si la première ligne du texte est une des lignes identifiées par ‹identificateur suite de lignes›.

- C devient le numéro de la ligne suivant la suite identifée par ‹identificateur suite de lignes›.

- L est mis à jour si la dernière ligne du texte est une des lignes identifiées par ‹identificateur suite de lignes›.

Voir aussi :

- Autres commandes modifiantes : a, c, i, m, s.

Nom :

e - end

Format :

e

Vérification par l'éditeur :

- Format.

Définition :

- La commande e est utilisée pour arrêter le processus d'édition. La fin est inconditionnelle si le texte a été sauvé après l'exécution de la dernière commande modifiante, dans le fichier dont le nom a été passé comme argument lors de l'appel à l'éditeur. Sinon, un dialogue avec l'utilisateur permet de savoir si celui-ci veut malgré tout la fin du processus d'édition.

- Au cas où l'utilisateur demande la continuation du processus d'édition, B, C, F et L sont inchangés.

Voir aussi :

- commande v.

Nom :

f - find

Format :

f‹string›^‹identificateur suite de lignes›

Vérification par l'éditeur :

- Format.

- ‹identificateur suite de lignes› identifie une suite de lignes existant dans le texte.

Définition :

- L'éditeur imprime , s'il existe, l'ensemble des lignes appartenant à la suite identifiée par ‹identificateur suite de lignes› qui contiennent le string ‹string›.

- Si le string n'existe pas dans la suite de lignes, un message d'erreur est envoyé à l'utilisateur.

- B, F et L restent inchangés.

- C devient le numéro de la dernière ligne imprimée, sauf si le string n'apparaît pas dans la suite de lignes identifiée, auquel cas C reste inchangé.

Voir aussi :

- Autre commande de recherche : p.

Nom :

h - help.

Format :

h [‹nom de commande›]

Vérification par l'éditeur :

- Format.

- ‹nom de commande›, s'il est spécifié, doit être le nom d'une commande de l'éditeur.

Définition :

_ Si ‹nom de commande› existe, l'éditeur donne des renseignements sur la commande ‹nom de commande›. Sinon, l'éditeur renseigne l'utilisateur sur toutes les commandes d'édition.

- B, C, F et L restent inchangés.

Nom :

i - insert

Format :

i ‹identificateur ligne› ^ ‹int›

Vérification par l'éditeur :

- Format.

- ‹int› est un entier strictement positif.

- Il existe dans le texte une ligne identifiée par ‹identificateur ligne›.

- ‹identificateur ligne› ne désigne pas la dernière ligne du texte.

- Le nombre de valeurs disponibles entre, d'une part, le numéro de la ligne identifiée par ‹identificateur ligne› et, d'autre part, le numéro de la ligne qui la suit, est plus grand ou égal à ‹int›.

Définition :

- Si le nombre de numéros disponibles pour l'insertion est inférieur à ‹int›, un message d'erreur est envoyé à l'utilisateur.

- Si l'insertion est possible, l'éditeur insère les lignes écrites par l'utilisateur après la ligne identifiée par ‹identificateur ligne›. L'insertion s'arrête lorsque l'utilisateur termine la ‹int›^{ème} ligne ou lorsqu'il écrit comme seul caractère d'une ligne le caractère "." en première position.

- B et L restent inchangés.

- Si l'insertion est possible, C devient le numéro de la dernière ligne insérée. Sinon, C reste inchangé.

- Si l'insertion est impossible, F ne change pas. Sinon, F ne devient le numéro de la première ligne insérée que si ‹identificateur ligne› désigne la ligne fictive de début de texte.

Voir aussi :

- Autres commandes modifiantes : a, c, d, m et s.

Nom :

m - move

Format

m ‹identificateur suite de lignes› ^ ‹identificateur ligne›

Vérification par l'éditeur :

- Format.

- La suite de lignes identifiée par ‹identificateur suite de lignes› existe, ainsi que la ligne identifiée par ‹identificateur ligne›.

- Le nombre de numéros disponibles entre, d'une part, la ligne identifiée par ‹identificateur ligne› et, d'autre part, la ligne qui la suit, est plus grand ou égal au nombre de lignes de la suite identifiée par ‹identificateur suite de lignes›.

Définition :

- La commande "m" place la suite des lignes identifiées par ‹identificateur suite de lignes› entre la ligne ‹identificateur ligne› et celle qui la suit.

- B est inchangé.

- Si ‹identificateur ligne› désigne la ligne de numéro 0, F devient le nouveau numéro attribué à la première ligne de la suite ‹identificateur suite de lignes›. Sinon, F est inchangé.

- Si ‹identificateur ligne› désigne la dernière ligne du texte, L devient le nouveau numéro attribué à la dernière ligne de la suite ‹identificateur suite de lignes›. Sinon, L est inchangé.

- C devient le nouveau numéro attribué à la dernière ligne de la suite ‹identificateur suite de lignes›.

Voir aussi :

- Autres commandes modifiantes : a, c, d, i et s.

Nom :

n - number

Format :

n <int>

Vérification par l'éditeur :

- Format.

- <int> > 0.

Définition :

- La commande n donne <int> comme nouvelle valeur au pas et renumérote toutes les lignes du texte en fonction de ce nouveau pas.

- B est inchangé.

- F devient égal à <int>.

- C devient : <int> * (1 + nombre de lignes avant C).

- L devient : <int> * (1 + nombre de lignes avant L).

Nom :

p - print

Format :

p ‹identificateur suite de lignes›

Vérification par l'éditeur :

- Format.

- La suite de lignes identifiées par ‹identificateur suite de lignes› existe dans le texte.

Définition :

- La commande p imprime la suite des lignes ‹identificateur suite de lignes›, si cette suite existe.

- B, F et L ne changent pas.

- C devient le numéro de la dernière ligne imprimée.

Voir aussi :

- Autre commande de recherche : f.

Nom :

s - substitute

Format :

s ‹string a›^‹string b›^ ‹identificateur suite de lignes›

Vérification par l'éditeur :

- Format.

- La suite de lignes ‹identificateur suite de lignes› existe dans le texte.

Définition :

- La commande s remplace toute occurrence du string ‹string a› dans la suite de lignes ‹identificateur suite de lignes› par le string ‹string b›.

- B, F et L sont inchangés.

- C devient le numéro de la dernière ligne dans laquelle a eu lieu une substitution, mais reste inchangé en cas de non substitution.

Voir aussi :

- Autres commandes modifiantes : a, c, d, i et m.

Nom :

v - save

Format :

v [<nom de fichier>]

Vérification par l'éditeur :

- Format.

Définition :

- Si aucun nom de fichier n'est spécifié, le texte est sauvé dans le fichier de sauvetage privilégié, c'est-à-dire celui dont le nom est donné comme argument lors de l'appel à l'éditeur.

- Sinon , le sauvetage a lieu dans le fichier <nom de fichier> si celui-ci est soit le fichier de sauvetage privilégié, soit le nom d'un fichier qui n'existe pas avant l'appel à l'éditeur. En particulier, <nom de fichier> peut désigner un fichier qui a déjà servi de fichier de sauvetage pendant l'édition.

- B, C, F et L ne sont pas modifiés.

Voir aussi :

Commande e.

3. PRINCIPES DE REALISATION.

(Michel Lestrade)

3. PRINCIPES DE REALISATION.

3.1. INTRODUCTION.

Après avoir défini le concept de texte ainsi que les différentes commandes de notre éditeur, nous allons maintenant analyser comment stocker et accéder au texte. Ces deux opérations étant maintenant vues non plus du côté utilisateur mais éditeur.

Comme vu précédemment, nous savons que : (point 2.4.1.)

- le texte est une suite de lignes
  une ligne est une suite de caractères. Le nombre de lignes par texte est variable de même que le nombre de caractères par lignes.

- chaque ligne est accessible par un numéro ou par un identificateur spécial (b, c, f ou l) pour certaines lignes du texte vu leur caractère particulier au sein du texte. (point 2.5.5.)

- l'accès à la ligne suivant la ligne courante doit être aisé : pour permettre à l'utilisateur de parcourir facilement son texte.

La façon de stocker la copie du texte en mémoire doit tenir compte des contraintes précitées.

Pour des raisons de sécurité, l'éditeur ne travaille pas directement sur le fichier source (édité) . Il est recopié dans le fichier de travail de l'éditeur. L'état initial du fichier source reste inchangé et donc récupérable par l'utilisateur tant qu'il n'y a pas eu sauvetage explicite (par une commande "save") du texte dans ce fichier.

Nous allons analyser successivement :

- les différents supports possibles
- les façons de stocker le texte sur le support choisi
- les mécanismes d'accès au texte.
- les mécanismes de gestion des pages et des fichiers de sauvetage.

3.2. REPRESENTATION DU TEXTE.

Plusieurs types de supports permettent de stocker des données informatiques.

Notre choix s'est effectué en deux étapes :

- tout d'abord, choix du type de support en fonction des accès

- ensuite, le choix du support en fonction de la taille variable des textes à éditer.

La représentation du texte en mémoire est liée au choix du support.

3.2.1. Choix du type de support en fonction des accès.

Du point de vue de leur structure logique, nous pouvons distinguer deux types de support :

- les supports permettant un accès séquentiel
- les supports permettant un accès direct.

Etant donné les accès effectués par l'utilisateur, nous avons retenu les supports permettant l'accès direct. Ceux-ci sont les mieux adaptés pour l'implémentation d'un éditeur de texte.

3.2.2. Choix du support en fonction de la taille variable des textes à éditer.

Plusieurs étapes pour ce choix :

- brève présentation des différents supports à accès direct
- différentes solutions pour stocker un texte sur de tels supports

- solution retenue

3.2.2.1. Présentation des différents supports à accès direct.

Les supports à accès direct sont des supports tels la mémoire centrale et les mémoires de masse : disques rigides, disques souples, floppy disks, ...

Bien que ces supports permettent tous l'accès direct, leurs caractéristiques diffèrent.

Pour une configuration donnée, nous avons retenu les caractéristiques suivantes :

Pour la mémoire centrale :

- taille limitée
- mémoire volatile : l'information existe tant qu'elle n'est pas détruite mais une panne du système d'alimentation entraîne sa destruction.
- l'accès aux informations en mémoire centrale est le plus rapide.

Pour les mémoires de masse :

- taille beaucoup plus importante que pour la mémoire centrale

- mémoire stable : l'information existe tant qu'elle n'est pas détruite

- l'information doit être amenée en mémoire centrale pour être traitée par l'ordinateur : existence d'entrées - sorties. Elles sont relativement lentes.

3.2.2.2. Différentes solutions pour stocker un texte sur de tels supports.

Les solutions analysées sont :

- tout en mémoire centrale

- tout en mémoire de masse

- partage du texte entre la mémoire centrale et la mémoire de masse

A. Tout en mémoire centrale.

Toute la copie du texte est en mémoire centrale. L'accès au texte est très rapide. Cependant, la mémoire centrale, de taille limitée, est partagée entre les différents utilisateurs de l'ordinateur. Elle ne peut pas contenir un texte de taille supérieure à la place allouée à chacun d'eux.

B. Tout en mémoire de masse.

Cette solution est irréaliste car si aucune partie du texte ne se trouve en mémoire centrale, l'éditeur ne peut pas travailler.

C. Partage du texte entre la mémoire centrale et la mémoire de masse.

Puisque :

- le texte est de taille variable et peut être très grand
- les informations doivent être en mémoire centrale pour que l'éditeur puisse y accéder
- la taille de la mémoire de masse est très grande par rapport à la taille de la mémoire centrale,

nous avons choisi de partager le texte entre la mémoire centrale et la mémoire de masse.

Seule une partie du texte est stockée en mémoire centrale, le reste se trouve en mémoire de masse. Cette solution entraîne des échanges d'informations entre la mémoire centrale et la mémoire de masse.

Le texte ainsi partagé, est découpé en une suite de morceaux.

Chacun d'eux, est appelé "page". Ces dernières peuvent donc se trouver soit en mémoire centrale ou soit en mémoire de masse. L'éditeur ne dispose, en mémoire centrale, que d' une seule page à la fois.

Une page a les caractéristiques suivantes :

- elle est une suite de caractères
- chaque page possède un identificateur. Il servira de clé d'accès en mémoire de masse.

Cette définition sera complétée ultérieurement. (cfr point 3.3.2.)

Pour cette solution, nous avons relevé les avantages suivants :

- la taille du texte est variable et peut être très grande
- la place occupée en mémoire centrale est faible par rapport à la taille totale du texte.

Malheureusement, cette solution entraîne des entrées - sorties.
Le texte, partagé entre la mémoire centrale et la mémoire de masse, doit être en mémoire centrale pour être accessible par l'éditeur. Les entrées - sorties nécessaires, plus lentes, peuvent diminuer le temps de réponse à l'utilisateur.

3.2.3. Remarque : pourquoi créer nous-mêmes la gestion des pages ?

Nous tenons nous-mêmes la gestion des pages bien qu'elle existe sur certaines machines (mémoire virtuelle) parce que :

- nous voulons une généralité suffisamment grande pour que cette analyse soit valable pour des machines ne disposant pas de ce mécanisme, les micros par exemple.
- il nous est apparu intéressant d'aborder ce problème.

3.2.4. Synthèse : représentation du texte pour l'éditeur.

Les représentations du texte pour l'utilisateur et pour l'éditeur peuvent être considérées comme différentes.

Pour l'éditeur, un texte a la représentation suivante :

- un texte est une suite de pages
- une page est une suite de lignes
- une ligne est une suite de caractères.

Nous pouvons remarquer que suite au partage du texte entre les deux supports, il y a eu introduction du concept de page.

Cette découpe supplémentaire doit être transparente pour l'utilisateur.

3.2.5 Description plus précise de la solution retenue.

Dans ce paragraphe nous allons définir la solution choisie d'une manière plus détaillée. La description comporte deux phases. Tout d'abord, le choix de la taille d'une page et, ensuite, la technique de découpe du texte.

3.2.5.1. Taille d'une page.

Comme nous l'avons vu, le texte est découpé en une suite de pages, nous allons maintenant fixer la taille d'une telle page.

Supposons qu'une page soit constituée d'un nombre variable de caractères.

Cette solution entraîne divers problèmes :

- Chaque page est susceptible d'être amenée en mémoire centrale. Il faut donc que la place réservée en mémoire centrale soit

  * de taille supérieure ou égale à la taille de la plus grande page

  * ou bien que la place soit réservée dynamiquement c'est-à-dire lorsque la place allouée en mémoire centrale n'est plus suffisante.

- les traitements de l'éditeur sur chaque page seront fonction de cette taille qui doit alors être passée comme paramètre

Si la taille d'une page est fixe, la place réservée en mémoire centrale peut être choisie à l'avance et est constante.

Nous avons choisi une page de taille fixe parce que :

- la zone en mémoire centrale est de taille fixe et cette taille est connue à l'avance de l'éditeur
- si la page est de taille fixe et si le système dispose du mécanisme de la mémoire virtuelle, le travail de transfert des pages peut être laissé à l' "operating system".

Pour une mémoire de masse, les entrées - sorties s'effectuent par blocs.

La taille d'une page est égale à la taille d'un bloc. Lorsqu'une page est amené en mémoire centrale il n'y a qu'un seul accès physique (si nécessaire) en mémoire de masse.
Cette taille est optimale pour une entrée-sortie.

Remarque :

Le nombre de caractères par page est constant, le nombre de caractères par ligne est variable.

En conséquence, nous pouvons énoncer le corollaire

suivant :

le nombre de lignes par pages est variable.

3.2.5.2. Découpe du texte en pages.

Cette découpe en pages est associée au fichier de travail de l'éditeur.

Le texte provenant du fichier à éditer ne possède pas cette structure de pages.

Nous allons maintenant analyser quand et comment elle s'effectue.

3.2.5.2.1. Localisation temporelle de la découpe.

Nous avons distingué précédemment (au point 2.3.) deux phases différentes :

- la création d'un texte
- l'édition.

Lors d'une création, les pages sont constituées pendant l'introduction initiale des lignes par l'utilisateur.

Lors d'une édition, nous pouvons distinguer deux phases :

- une phase d'initialisation : le fichier source est lu et recopié dans le fichier de travail de

l'éditeur. La découpe du texte en pages se fait pendant cette phase d'initialisation.

- l'édition proprement dite. Pendant cette dernière, le contenu des pages peut évoluer.

Ces mécanismes seront expliqués par la suite. (3.4.4.)

3.2.5.2.2. Technique de découpage.

Lors de la création d'un texte et lors de la phase d'initialisation de l'édition, les lignes sont lues et introduites consécutivement dans les pages.
Chaque page est donc constituée d'une suite séquentielle de lignes.
Si l'utilisateur parcourt son texte de manière séquentielle, le nombre d'entrées - sorties est diminué.

Cette suite de lignes peut cependant être modifiée par le processus d'édition.

Tant que la place disponible dans la page est suffisante pour insérer une nouvelle ligne, l'éditeur l'y introduit sans problème.

Lors de la constitution des pages, si une page ne contient plus assez de place pour contenir entièrement une nouvelle ligne :

- ou bien on accepte le débordement d'une ligne sur une deuxième page

- ou bien on le refuse.

Si on accepte le débordement, toute page est remplie au maximum. Cependant, cette solution présente certains désavantages :

- l'accès à une ligne du texte peut entraîner le transfert de deux pages en mémoire centrale.

- en cours d'édition, de la place libre peut être créée dans une page. Supposons qu' une ligne ait été éclatée sur une deuxième page. La place libérée est-elle suffisante pour ramener la partie de ligne éclatée sur cette page ?
  SI oui, faut-il ou non ramener la partie de ligne éclatée ?

  Si oui : la gestion risque d'être très lourde

  Si non : on ne profite pas de la possibilité de regrouper la ligne dans une seule page. Ceci aurait pour effet de diminuer le nombre d'entrées- sorties lors d'un accès ultérieur à cette ligne.

Si on refuse le débordement : toute ligne doit être entièrement contenue dans une page.
Avec cette façon de procéder, il peut exister de la place inoccupée en bas de page. Néanmoins, elle peut être récupérée

par l'éditeur lors d'insertions futures.

Nous avons retenu la deuxième solution parce que :

- lors de l'accès à une ligne, le transfert (si nécessaire) d'une seule page en mémoire centrale suffit
- la gestion est plus simple lors de la libération de place
- le pourcentage de place inoccupée dans une page est faible. En moyenne, il y a 1/2 ligne inoccupée par page.

Toute page ainsi constituée est copiée dans le fichier temporaire.

3.3. MECANISME D'ACCES AUX LIGNES.

Après avoir choisi le support ainsi que la découpe du texte en pages, nous allons maintenant analyser comment s'effectuent les accès aux lignes.

L'utilisateur peut référencer les différentes lignes de son texte grâce à leur(s) identificateur(s).

Pour l'éditeur, l'accès à une ligne se décompose en deux étapes : tout d'abord l'accès à la page la contenant puis à la

ligne proprement dite.

3.3.1. Accès aux pages.

Pour que l'éditeur puisse travailler sur une ligne du texte, la page la contenant doit être en mémoire centrale.

En mémoire centrale, l'éditeur ne dispose que d'une seule page.

Deux possibilités :

- ou bien la page contenant la ligne recherchée est déjà en mémoire, la ligne est donc immédiatement accessible à l'éditeur
- ou bien la page n'est pas en mémoire centrale. L'éditeur doit alors recopier cette page en mémoire de masse et amener la page désirée en mémoire centrale.

Chaque page doit donc pouvoir être identifiée.

L'identificateur choisi est un numéro. Il est attribué séquentiellement par pas de un. Il servira de clé d'accès en mémoire de masse.

Le choix d'un numéro pour référencer les différentes pages a été influencé par la technique d'accès (en mémoire de masse)

dont nous diposions. Les pages sont rangées consécutivement en mémoire de masse. Le numéro permet d'aller se positionner directement sur le début de la page.

3.3.2. Complément à la définition d'une page.

Deux nouveaux éléments viennent s'ajouter à la définition d'une page :

- chaque page est identifiée par un numéro. Ce numéro sert de clé d'accès.
- toute ligne est entièrement contenue dans une page.

3.3.3. Accès aux lignes.

Pour accéder aux lignes de son texte, l'utilisateur fournit leur(s) identificateur(s) à l'éditeur.

Rappelons qu'un identificateur d'une ligne peut être :

- soit un numéro
- soit un identificateur spécial pour certaines lignes du texte.

  (cfr. 2.5)

Un identificateur spécial (b, c, f ou l) sera transformé en un numéro par l'éditeur.

L'accès se fait donc toujours sur base d'un numéro.

Nous allons maintenant étudier le mécanisme d'accès et les différentes possibilités pour le réaliser sur base d'un numéro.

3.3.3.1. Les numéros de lignes appartiennent au texte.

Le numéro est la seule information dont dispose l'éditeur. Il se trouve en tête de chaque ligne.

Les lignes peuvent être triées ou non sur base de leurs numéros dans chaque page.

A) Les lignes ne sont pas triées.

Avec cette solution, l'ordre des lignes dans chaque page peut être quelconque.

Cependant , cette solution présente les inconvénients suivants :

- quand l'éditeur recherche une ligne sur base de son numéro, la recherche ne peut s'arrêter que lorsque l'on a trouvé la ligne recherchée ou que l'on est arrivé en fin de texte.
  Cette solution entraîne éventuellement beaucoup d'entrées - sorties.

- à partir d'une ligne, la recherche de la ligne suivante pose les mêmes problèmes.

B) Les lignes sont triées par ordre croissant des numéros.

Grâce à ce tri, la recherche d'une ligne peut s'arrêter soit lorsque l'éditeur a trouvé la ligne demandée ou une ligne de numéro supérieur.

Pour cette solution, nous avons relevé les inconvénients suivants :

- la recherche doit se faire au niveau de la page et risque donc d'entraîner un nombre relativement important d'entrées - sorties
- les lignes doivent garder, toujours, le tri par ordre croissant de leurs numéros.

3.3.3.2. Les numéros n'appartiennent pas au texte.

Sur base du numéro de ligne, l'éditeur doit retrouver le texte de la ligne. Il faut donc établir une fonction de correspondance entre les numéros et les lignes.

A. Elle est interne aux numéros.

Le numéro à lui seul permet de retrouver la ligne c'est-à-dire de retrouver en premier lieu le numéro de la page puis de la localiser à l'intérieur de celle-ci.

Ce numéro a la forme XX YYY par exemple où XX est le numéro de la page et YYY le déplacement de la ligne dans la page ou

encore le numéro d'ordre de la ligne dans la page.

Cette solution a été rejettée immédiatement parce que :

- le numéro n'est pas habituel pour l'utilisateur
- la découpe en pages n'est plus transparente pour l'utilisateur.

B. La fonction de correspondance est externe aux numéros.

D'autres informations que le numéro de ligne permettent à l'éditeur de localiser une ligne. Le numéro permet en réalité de retrouver une suite d'informations concernant la ligne.

La première d'entre elles est le numéro de page. Ensuite, l'éditeur doit pouvoir situer la ligne à l'intérieur de celle-ci.

L'information à ce sujet peut être :

- soit le numéro d'ordre de la ligne dans la page.
  Avec cette solution, l'éditeur doit parcourir le texte dans la page juqu' à la ligne recherchée.
  Pour que l'éditeur puisse accéder immédiatement au premier caractère de la ligne, il doit alors connaître la longueur des différentes lignes. La localisation du premier caractère s'obtenant, facilement, par calcul.
- soit le déplacement de la ligne dans la page.
  Avec cette solution, l'éditeur peut accéder

directement au premier caractère de la ligne.

Avec ces deux solutions,les lignes ne doivent pas être, nécessairement, triées.

3.3.3.3. Solution retenue.

Les numéros n'appartiennent pas au texte, la fonction de correspondance leur est externe.
Sur base du numéro de ligne, l'éditeur peut retrouver le numéro de la page et le déplacement de la ligne dans celle-ci.

Avec cette solution :

- la numérotation des lignes est indépendante de la découpe en pages. Elle est transparente pour l'utilisateur.
- l'éditeur peut accéder rapidement au premier caractère de la ligne grâce au déplacement.
- l'ordre des lignes peut être quelconque à l'intérieur de chaque page.

Si le contenu de la page évolue, les déplacements des lignes, à l'intérieur de celle-ci, doivent être mis à jour.

3.3.3.4. Délimitation des lignes.

Sur base du numéro, l'éditeur peut accéder au premier caractère d'une ligne. Il ne connaît cependant pas sa fin.

Plusieurs possibilités s'offrent à nous :

- chaque ligne est terminée par un caractère spécial
- pour toute ligne, l'éditeur connaît sa longueur.

Si pour toute ligne, l'éditeur connaît sa longueur, cette information doit être stockée et mise à jour pour toute modification de cette dernière (lors d' une substitution par exemple).

Si par contre, elle est terminée par un délimiteur, aucune information supplémentaire ne doit être mémorisée.
Cependant, lors de la lecture d'une ligne, l'éditeur doit tester chaque caractère pour déterminer la fin de ligne.

Nous avons choisi la solution avec délimiteur parce que :

- aucune information supplémentaire ne doit être mémorisée si ce n'est ce délimiteur
- le délimiteur existe. L'utilisateur détermine lui-même la fin de ses lignes par un caractère spécial (le carriage return).
  Ce caractère peut être repris par l'éditeur comme délimiteur.

3.3.2.5. Description de la fonction externe.

Pour tout accès à une ligne de son texte, l'utilisateur fournit son identificateur à l'éditeur.

Sur base de celui-ci, l'éditeur doit être capable de retrouver le numéro de la page contenant cette ligne et son déplacement.

Ces informations doivent être facilement accessibles à l'éditeur.

Nous allons maintenant étudier comment organiser ces informations.

3.3.3.5.1. Tri des informations nécessaires à l'éditeur.

Les informations peuvent être triées ou non sur base du numéro de ligne.

Si les informations ne sont pas triées par ordre croissant des numéros :

- lors d'une recherche, l'éditeur ne peut s'arrêter que s'il a trouvé les informations cherchées ou bien lorsqu'il les a parcourues toutes.
- l'accès aux informations de la ligne suivant la ligne courante n'est pas aisé. L'éditeur ne sait

pas à priori où elles se trouvent .

- leur ordre peut peut être quelconque. L'insertion ou la suppresion d'informations ne nécessite pas de réorganisation.

Si les informations sont triées :

- lors de l'insertion ou la suppresssion d'informations, le tri doit être respecté
- l'éditeur peut arrêter sa recherche lorsqu'il a trouvé les informations demandées ou lors de la découverte d'un numéro supérieur.
- les informations concernant la ligne suivant la ligne courante sont situées juste après celles de cette dernière.

Avec cette solution cependant, le tri doit être respecté.

Nous avons choisi de trier ces informations pour faciliter la recherche de l'éditeur et donc essayer de diminuer le temps de réponse pour l'utilisateur.

3.3.3.5.2. Organisation de ces informations.

Pour chaque ligne les informations sont :

- numéro de ligne

- numéro de page
- déplacement de la ligne dans la page.

Nous allons essayer de structurer ces informations triées de telle manière que l'éditeur puisse y accéder le plus facilement possible.

Plusieurs représentations sont possibles pour les listes linéaires :

([3] KNUTT "Fundamental Algorithms" pages 234 et sq.)

Une liste linéaire est une suite de n (supérieur ou égal à 0) éléments X(0),X(1),X(2),...X(n) dont la propriété structurale est la position linéaire (une seule dimension) des éléments.

Si n > 0, X(1) est le premier élément, quand 1 < k < n, le k-ème élément X(k) est précédé par l'élément X(k-1) et suivi par X(k+1) et X(n) est le dernier élément.

A) Allocation séquentielle.

Pour stocker une liste linéaire, la solution la plus simple est l' allocation séquentielle en mémoire.

On a alors :

adresse[X(j+1)] = adresse [X(j) + c]

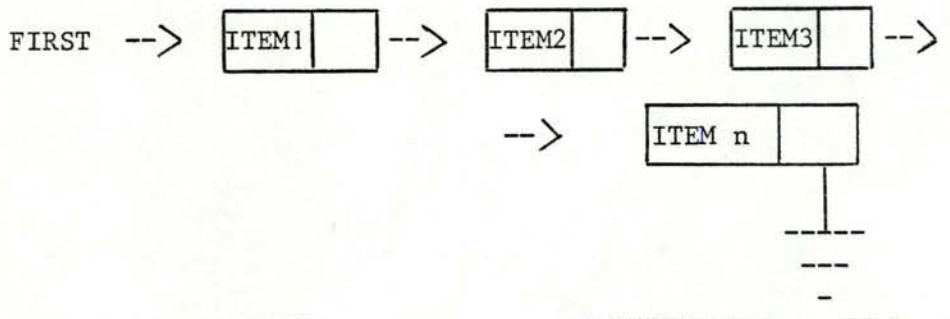
où c est le nombre de mots mémoire

par élément.

B) Allocation chaînée.

Plutôt que d'avoir une représentation séquentielle, nous pouvons utiliser une solution plus souple : l'allocation chaînée.

La liste peut être représentée graphiquement comme suit :



en supposant que la liste
comporte n éléments

fig. 3.1.

Chaque élément de la liste possède deux champs :

- un champ d'informations
- un champ de lien (vers l'élément suivant par exemple).

Ici, FIRST est un pointeur vers le premier élément de la liste. Le champ de lien du dernier élément de la liste contient le lien null.

Par rapport à la solution précédente, les différences suivantes peuvent être relevées :

- les liens occupent de la place supplémentaire en mémoire
- la destruction d'un élément consiste en la modification d'un lien. Pour une représentation séquentielle, cette destruction entraîne le déplacement d' une partie plus ou moins grande de la liste.
- l'introduction d'un élément se fait facilement par modification de deux liens.

L'emploi des listes chaînées implique l'existence de mécanismes pour trouver de la place libre lors de l'insertion de nouveaux éléments.

C) Liste circulaire.

Une liste circulaire a la propriété suivante : son dernier élément pointe vers le premier.

On peut accéder à n'importe quel élément en partant d'un noeud quelconque.

On détecte la fin de la liste quand on est revenu au point de départ.

Pour éviter le problème de la liste vide, on peut introduire un élément de tête qu'on s'interdit de détruire.

D) Liste à liens doubles.

Les listes chaînées ou circulaires peuvent être à lien double.

Chaque élément possède un lien vers l'élément suivant et un vers le précédent.

La place supplémentaire occupée est compensée par :

- le parcours de la liste peut s'effectuer dans les deux sens
- lors de la destruction d'un élément, on doit connaître le précédent pour rétablir les liens. Si la liste n'est chaînée que dans un seul sens et si on ne possède pas de pointeur explicite vers cet élément, on l'obtient en parcourant la liste. Ce mécanisme est très lourd.

L'introduction d'un élément de tête évite ici aussi, le problème de la liste vide.

E) Solution retenue.

Vu les opérations qui seront effectuées par l'éditeur, nous avons choisi une liste chaînée circulaire à doubles liens.

La liste est chaînée parce que :

- les opérations de suppression et d'insertion se font plus facilement que si la représentation est séquentielle
- la taille de la liste est indéterminée à priori. La liste peut croître tant qu'il subsiste de la place libre en mémoire.

La liste est circulaire parce que :

- à partir d'un élément, l'éditeur peut accéder à n'importe quel élément de la liste
- les insertions et suppression d'éléments en début ou en fin de liste sont symétriques. On ne doit pas penser en terme de premier ou de dernier élément.

La liste est à doubles liens parce que :

- l'éditeur peut parcourir la liste dans les deux sens
- à partir d'un élément, l'éditeur peut retrouver facilement son prédécesseur ou son successeur pour modifier les liens lors d'opérations d'insertion ou de suppression.

Bien que :

- cette solution occupe plus de place que lors d'une allocation séquentielle. La place occupée par le texte est faible. On peut supposer qu'il restera suffisamment de place disponible pour implanter cette liste en mémoire centrale.

  Si la configuration était telle qu'il serait manifestement impossible de la stocker en mémoire centrale, plusieurs solutions pourraient alors être envisagées.

  Notamment, le partage de cette liste entre la mémoire centrale et la mémoire de masse. Mais, étant donné les accès très fréquents de l'éditeur, ce partage peut diminuer le temps de réponse à l'utilisateur.

  Ou encore, diminuer le nombre d'informations dans la liste et adopter une autre technique d'accès aux lignes.

L'accès à un élément de la liste se fait par un parcours de la liste.

Pour éviter le problème de la liste vide, nous avons introduit un élément de tête qu'on interdit de détruire.

De plus, pour faciliter l'entrée de l'éditeur dans la liste,

nous avons également introduit deux éléments fictifs :

- un élément correspondant à la ligne fictive de début de texte (élément "begin")
- et un autre correspondant à la ligne fictive de fin de texte (élément "end").

L'élément fictif de fin ne sait être référencé par l'utilisateur. Il est transparent pour lui. La ligne fictive de début peut intervenir dans les commandes "insert","move" et "copy". L'effet est alors d'insérer, de transférer et de copier des lignes avant la première ligne réelle du texte.

Ces deux éléments possèdent chacun leur pointeur propre :

- " pt_b " pour l'élément fictif de début
- " pt_e " pour l'élément fictif de fin.

Chaque élément de la liste a la forme :

- numéro de la ligne
- pointeur vers l'élément suivant
- pointeur vers l'élément précédent
- numéro de la page contenant la ligne
- déplacement de la ligne dans la page.

Cette liste est créée lors de la phase d'intialisation de l'édition ou de création du texte.

Désormais, nous appelerons cette liste : la " liste des lignes ".

REMARQUE :

L'évaluation de cette technique choisie ainsi que la proposition de solutions alternatives sera faite au chapitre 6.

3.4. MECANISME DE GESTION DES PAGES.

3.4.1. Introduction.

Nous allons maintenant étudier la gestion des pages.

En cours d'édition, le contenu des pages peut évoluer. L'utilisateur peut ajouter et/ou supprimer des caractères, des lignes dans son texte. Ces modifications vont avoir des répercussions sur le contenu des pages.

L'éditeur doit gérer la place demandée ou libérée dans les différentes pages.
Rappelons que la taille d'une page est constante.

3.4.2. Localisation de la place libre.

L'utilisateur peut supprimer des caractères. Des "trous" vont se créer à l'intérieur des pages. Nous appelons "trou" une suite

de caractères libres au sein d'une page.

Deux possibiltés s'offrent à nous :

- ou bien, les trous sont regroupés en bas de page
- ou bien, ils subsistent morcellés au sein de chaque page.

Nous allons raisonner sur une et une seule page, le mécanisme étant identique pour toutes.

3.4.2.1. Les caractères sont regroupés en bas de page.

Lors de la suppression de x caractères, les caractères suivant le dernier caractère libéré sont shiftés de x positions vers la gauche. Ce mécanisme a pour effet de regrouper les caractères libres en bas de page.

Nous avons relevé les avantages suivants :

- la place ainsi regroupée, pourra être plus facilement récupérée par l'éditeur
- il est facile de localiser et de dénombrer les caractères libres.

Cette solution présente néanmoins des désavantages :

- l'existence de shifts dans la page

- nécessité de la mise à jour de la liste des lignes. Suite au shift, les déplacements de certaines lignes ont changé. Pour toute ligne dont le déplacement était supérieur à celui du premier caractère libéré, le nouveau déplacement devient égal à : vieux déplacement - le nombre de caractères libérés.

3.4.2.2. Les caractères libres sont disséminés dans la page.

La place libérée "reste" en corps de page.

Pour cette solution, les avantages sont :

- la suppression de caractères n'entraîne pas de shifts dans la page
- la liste des lignes ne doit pas être mise à jour.

Par contre, les désavantages sont les suivants :

- les trous ainsi disséminés sont de petite taille. L'éditeur ne pourra pas les récupérer facilement
- la localisation des trous n'est pas facile. On pourrait imaginer de chaîner les trous dans chaque page. Ce mécanisme risque d'être très lourd.

3.4.2.3. Solution retenue.

Nous avons choisi de regrouper les caractères libres en bas de page parce que :

- les shifts à effectuer sont localisés à une seule page d'où leur nombre relativement faible
- ces shifts sont effectués sur une page en mémoire centrale et sont donc très rapides
- les modifications des déplacements dans la liste des lignes, stockée en mémoire centrale sont très rapides.

3.4.3. Principes de gestion.

Nous savons maintenant que les caractères libres sont regroupés en bas de page.

Lors de l'insertion de caractères par l'utilisateur, l'éditeur va rechercher de la place libre.
La contrainte de non débordement d'une ligne hors d'une page doit être respectée.

3.4.3.1. Technique de localisation de la place libre.

L'éditeur doit disposer d'informations pour localiser la place libre dans chaque page.

Pour chaque page, cette information peut être :

- l'adresse du premier caractère libre
- le nombre de caractères occupés
- le nombre de caractères libres.

A) Adresse du premier caractère occupé.

L'éditeur recherche la place libre c'est-à-dire le nombre de caractères encore inoccupés dans la page.

Cette information n'est pas disponible immédiatement, elle s'obtient par calcul.

Soit Xn = l'adresse du dernier caractère de la page, et Xj, l'adresse du premier caractère libre. L'unité adressable étant le caractère.

La formule pour trouver le nombre de caractères inoccupés est :

NOMBRE = Xn - Xj (1)

Cette solution nous paraît cependant peu intéressante pour deux raisons :

- l'information fournie à l'éditeur est peu significative
- l'adresse fournie est une adresse physique. Cette adresse peut évoluer si

l'ordinateur dispose d'un mécanisme de mémoire virtuelle.

B) Nombre de caractères occupés.

L'éditeur connaît le nombre de caractères occupés par page.

Cette information permet de déterminer facilement l'adresse du premier caractère libre :

elle est égale à : Xo + nb_occupés (2)

où Xo est l'adresse du début de page

et nb_occupés le nombre de caractères occupés dans la page.

L'information intéressant l'éditeur en premier lieu, le nombre de caractères inoccupés, est égal à la taille de la page - nb_occupés.

C) Nombre de caractères inoccupés.

L'adresse du premier caractère libre est égale à :

Xn - nb_libres (3)

où Xn est l'adresse de fin de page

et nb_libres le nombre de caractères

libres dans la page.

L'information concernant le nombre de caractères libres est évidemment accessible immédiatement.

D) Choix.

Nous avons choisi de retenir le nombre de caractères occupés parce que :

- le calcul de l'adresse du premier caractère libre s'obtient très facilement par la formule (2).
- ce nombre représente le déplacement par rapport au début de la page pour le premier caractère libre. Même philosophie que pour repérer la ligne dans une page.

Cette solution est cependant plus coûteuse que celle exposée au point C). L'éditeur doit évaluer le nombre de caractères libres dans la page. Ce calcul, élémentaire, est très rapide.

3.4.3.2. Outils de gestion.

Pour la solution retenue, nous voyons donc que

l'éditeur doit pouvoir déterminer le nombre de caractères libres dans chaque page.

La même analyse peut être faite que pour la liste des lignes.

Ici aussi, nous avons choisi une liste circulaire. Elle est chaînée dans un seul sens.

Nous avons choisi une telle liste parce que :

- l'insertion se fait aisément dans cette liste de taille plus faible que celle des lignes. L'ajout d'un élément est moins fréquent. Le parcours, nécessaire pour l' insérer dans cette liste, est peu coûteux.

- la recherche dans cette liste se fait séquentiellement et est très rapide par pointeurs

- par soucis de cohésion en regard du choix effectué pour les lignes.

Chaque élément a la forme :

- numéro de page

- nombre de caractères occupés

- pointeur vers l'élément suivant de la liste.

Pour éviter le cas de la liste vide,nous avons introduit un élément de tête qu'on interdit de détruire.

Cette liste est créée lors de la création d'un texte ou lors de la phase d'initialisation de l'édition. Elle évolue en cours d'édition.

En cours d'édition, lors d'insertions et de suppressions, elle sera mise à jour.

Cette liste sera, désormais, appelée la " liste des pages ".

3.4.4. Politique de récupération de la place libre.

En cours d'édition, l'utilisateur va insérer de nouveaux caractères, de nouvelles lignes dans son texte.

Rappelons que l'éditeur doit respecter le non débordement d'une ligne hors d'une page.

Dans ce chapitre, nous nous contentons d'exposer uniquement la solution adoptée. Son évaluation ainsi que la proposition d'autres solutions étant faite au chapitre 6.

3.4.4.1. Insertion de nouvelles lignes.

L'utilisateur désire insérer de nouvelles lignes dans son texte.

L'analyse développée traite l'insertion d'une seule ligne.
Pour introduire plusieurs lignes, le mécanisme doit être répété autant de fois qu'il n'y a de lignes à insérer.

Supposons que l'utilisateur désire insérer une ligne à la suite de la ligne référencée par l'identificateur "k".

L'éditeur va chercher de la place libre pour introduire cette ligne dans une page.

Cette recherche s'effectue comme suit :

l'éditeur recherche la première page qui contient assez de caractères libres pour accueillir la nouvelle ligne.

Si une telle page existe, l'éditeur insère la ligne à partir du premier caractère libre en bas de page.

Avantages de cette solution :

- simplicité des algorithmes de recherche
- l'éditeur essaye de récupérer la place libre dans les pages.

Par contre les désavantages sont :

- les lignes vont être dispersées à l'intérieur des pages. Le rangement séquentiel des lignes ne sera plus respecté.
  Ceci risque d'entraîner des entrées-sorties supplémentaires lors d'un parcours séquentiel du texte par l'utilisateur.

Si aucune page ne satisfait à la recherche, l'éditeur crée une nouvelle page. Il recopie la page, actuellement en mémoire centrale, en mémoire de masse (pour la sauver) et libère ainsi la zone. Il introduit également un nouvel élément dans la liste des pages.

3.4.4.2. Insertion de nouveaux caractères dans une ligne existante.

La ligne appartient déjà à une page.

L'éditeur effectue la substitution en mémoire centrale. Il considère alors cette ligne comme une nouvelle ligne et l'introduit comme expliqué en 3.4.4.1. L'ancienne ligne est détruite.

Cette solution a été choisie en raison de sa simplicité. Celle-ci résulte du fait que le traitement du remplacement

est identique pour toute substitution quelle que soit la nouvelle longueur de la ligne modifiée.

3.5. Mécanisme de gestion des fichiers de sauvetage.

L'utilisateur peut sauver son texte suivant les règles définies au point 2.9.1. (commande "save").

L'éditeur doit pour toute commande "save", avoir la possibilité de connaître les fichiers ayant déjà servi au sauvetage. Pour chacun d'eux, il doit au moins connaître son nom.

Pour rassembler ces informations, nous avons créé une troisième liste circulaire : la liste des fichiers de sauvetage.

Les raisons de ce choix sont identiques à celles qui nous ont déjà amené à définir les listes des lignes et des pages.

4. STRUCTURE DES TRAITEMENTS.

(Michel Lestrade)

4. STRUCTURE DES TRAITEMENTS.

4.1. INTRODUCTION.

Dans ce chapitre, nous allons exposer la découpe modulaire du programme. La méthode d'analyse est le " TOP-DOWN ".

Pour ce faire, nous allons d'abord, définir une représentation simplifiée (en 4 niveaux) de l'architecture globale (point 4.2.1.). Les différents modules ainsi déterminés sont détaillés au point 4.2.2. et sq.

Le paragraphe 4.3. décrit les modules de plus bas niveaux (supérieur à 3). A une représentation graphique trop compliquée, nous avons préféré une description linéaire des différents modules. L'enchaînemnt entre ces derniers étant valorisé par la notion de " module appelant ".

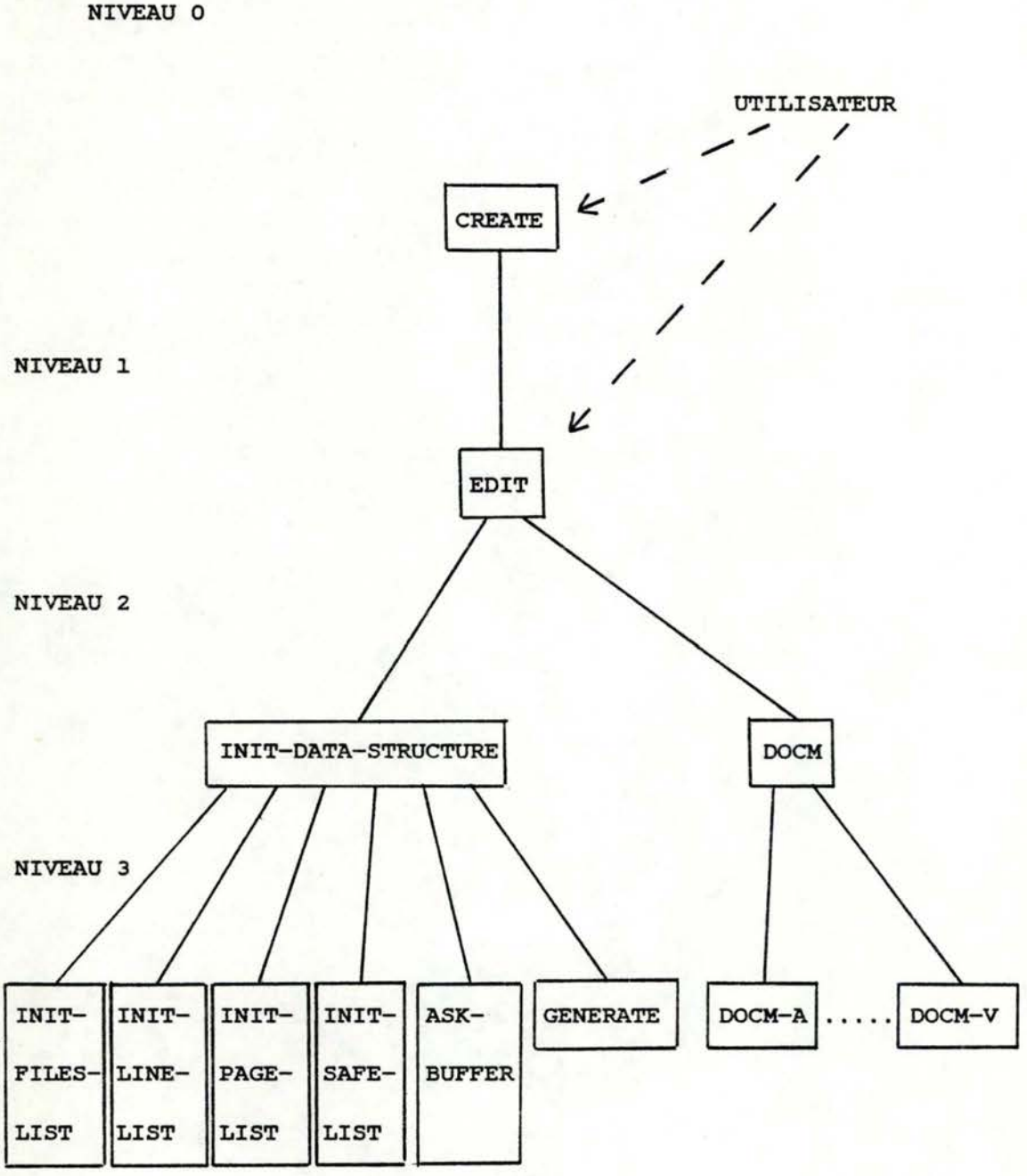
Le dernier chapitre (cfr 4.4.) comprend quelques remarques sur cette découpe.

4.2. ARCHITECTURE GLOBALE SIMPLIFIEE.

Nous allons tout d'abord définir une architecture très simple du programme.

Cette description comprend tout d'abord, une représentation graphique. Vient ensuite, la définition des modules.

4.2.1. Représentation graphique.



(figure 4.2.)

remarque :

Les commandes utilisateurs sont distinctes. Nous retrouvons

donc un module de niveau 3 pour chaque commande.

4.2.2. Description des différents modules.

Nous allons procéder par niveau.

Pour chaque module, nous allons décrire :

- les arguments en entrée
- sa fonction
- ses arguments en sortie si ils existent.

Sauf mention explcite, les paramètres d'entrée d'un module restent inchangés.

4.2.2.1. Module de niveau 0.

Module "CREATE".

entrée :

ce module reçoit la commande de création d'un nouveau fichier.

objectif :

créer le fichier de nom spécifié par l'utilisateur.

a) vérifier qu'un fichier de ce nom n'existe pas encore. Si la réponse est négative, le module envoie un message à l'utilisateur lui spécifiant le conflit et le progamme se termine immédiatement. Aller en b) sinon.

b) le fichier est créé. Le module EDIT est alors appelé. Le processus d'édition normale se déroule, le fichier d'entrée étant vide. La seule opération exécutable par l'utilisateur, à cet instant, est l'introduction de texte.

sortie :

un message d'erreur est destiné à l'utilisateur si un fichier de ce nom existait déjà, néant sinon.

4.2.2.2. Module de niveau 1.

Module "EDIT".

entrée :
ce module reçoit le nom du fichier à éditer.

fonction :
la première étape consiste à vérifier l'existence du fichier à éditer.
Si le fichier existe, il appele le module d'initialisation de l'édition (module INIT-DATA-STRUCTURE) et crée le fichier de travail de l'éditeur.

Puis, boucle sur le module DOCM (développé plus loin) jusque la commande de fin d'édition.

Si le fichier n'existe pas, il provoque la sortie du programme d'édition.

sortie :
envoi d'un message à l'utilisateur si le fichier à éditer n'existe pas. Sinon, le module retourne le nom et le descripteur du fichier temporaire associé au fichier édité.

Remarque : l'entrée dans ce module a deux origines différentes. Soit l'utilisateur crée un nouveau fichier (commande "create" [< nom de fichier >]), l'appel à l'éditeur lui est alors transparent. Soit lors de l'appel explicite de l'éditeur (commande "edit" [< nom de fichier>]).

4.2.2.3. Niveau 2.

Module "INIT-DATA-STRUCTURE".

entrée :
ce module reçoit le nom du fichier à éditer.

fonction :
ce module initialise l'édition c'est-à-dire pour le fichier temporaire :

- crée la liste des lignes (appel au module INIT-LINE-LIST)
- crée la liste des pages (appel au module INIT-PAGE-LIST)
- crée la liste des fichiers de sauvetage (module INIT-SAFE-LIST)
- réserve une zone en mémoire centrale destinée à accueillir une page en mémoire (appel au module ASK-BUFFER)
- recopie le fichier source dans le fichier de travail de l'éditeur, le découpe en pages et remplit les listes des lignes et des pages (appel au module GENERATE)

sorties :
le module retourne soit 1 et les pointeurs correspondant respectivement aux listes créées et à la zone réservée. Soit la valeur 0 et envoie un message à l'utilisateur si l'opération n'a pu se dérouler normalement.

Module DOCM.

entrées :

- nom et descripteur du fichier temporaire, pointeur vers les listes associées à ce fichier. Ces informations seront désormais appelés "indentification du fichier temporaire"
- nom et descripteur du fichier édité.

fonction :

ce module imprime d'abord le caractère "*" (l'éditeur attend une commande). Puis, il va chercher la commande introduite au terminal par l'utilisateur et la place dans une zone en mémoire centrale (module GETIN).

Il cherche le premier caractère non blanc c'est-à-dire, en fait, le caractère identifiant la commande (module SKWH).
Il vérifie s'il correspond à une commande existante.
si oui, il passe la main à un module spécialisé correspondant à la commande proprement dite.
sinon il renvoie un message d'erreur à l'utilisateur et attend la commande suivante.

sortie :

néant sinon l'envoi d'un message en cas d'erreur.

4.2.2.4. Niveau 3.

Nous allons maintenant analyser l'architecture d'une façon plus détaillée.

Nous allons, tout d'abord, décrire les modules concernant la phase d'initialisation de l'édition. Viennent ensuite, les modules spécifiques à la réalisation des différentes commandes.

Module INIT-LINE-LIST.

entrée :
identification du fichier temporaire.

objectif :
créer et initialiser, en mémoire centrale, la liste des lignes associé à ce fichier.

Les éléments suivants sont générés :
- l'élément de tête
- deux éléments correspondant respectivement aux lignes fictives de début et de fin de texte.
  Le numéro de la ligne fictive de début de texte est, conventionnellement égal à 0. Celui de la ligne imaginaire de fin de texte prend la valeur maximale d'un entier. Cette contrainte nous a été dictée par la représentation en machine du numéro de ligne. Nous avons choisi un entier. De ce fait, le nombre maximum de lignes dans le fichier se trouve aussi limité. Sa valeur maximale est , pour la machine dont nous disposions, (2 exposant 15) - 2. Les numéros zéro et 2 exposant 15 étant réservés.
  Le pointeur vers la ligne courante est initialisé à la ligne fictive de début.
  De plus, la taille d'une ligne a été limitée. Cette contrainte vient de la configuration du terminal. Une taille maximale a donc été introduite.

sortie :
la valeur 0 s'il n'existe plus de place mémoire pour créer ces éléments , 1 et un pointeur vers chacun d'eux sinon.

Module INIT-PAGE-LIST.

entrée :
identification du fichier temporaire

objectif :
créer et initialiser la liste des pages. L'élément de tête est créé. Le numéro de page -1 lui est attribué par convention.

sortie :
0 et un message est envoyé à l'utilisateur si la place disponible en mémoire est insuffisante, 1 et un pointeur vers cet élément sinon.

Module INIT-SAFE-LIST.

entrée :
identification du fichier temporaire.

objectif :
créer la liste des fichiers de sauvetage pour le fichier édité.

L'élément de tête de liste est généré.

sortie :
retourne 1 et un pointeur vers cet élément si la création a été possible, 0 et envoie un message à l'utilisateur sinon.

Module ASK-BUFFER.

entrée :
identification du fichier temporaire.

objectif :
réserver une zone, de la taille d'une page, en mémoire centrale pour ce fichier.

sortie :
retourne la valeur 1 et un pointeur vers le premier caractère de cette zone si la place diponible était suffisante, 0 et envoie un message à l'utilisateur sinon.

Module GENERATE.

entrée :
noms et descripteurs de fichier des fichiers à editer et temporaire. Pointeurs vers les listes des lignes et des pages associées au fichier temporaire.

objectif :
recopier le fichier à éditer dans le fichier temporaire en le découpant en pages.

Pour toute ligne du fichier édité, terminée par un "carriage return", l'amener en mémoire centrale et l'injecter dans le fichier temporaire (appel au module ADLN, developpé plus loin). Leurs numéros sont calculés. Les listes des lignes et des pages sont étoffées pendant cette opération.

sortie :
le module retourne 0 s'il y a eu une erreur d'entrée-sortie et envoie un message à l'utilisateur, 1 sinon.

Module DOCM-A.

entrée :

- ce module reçoit la commande utilisateur
- identification fichier temporaire.

objectif :

1) Contrôle du format de la commande (appel au module CHECK-A-FORMAT).

2) Si le format est correct : exécution de la commande proprement dite (appel au module INSERT-LINES).
Sinon, retour au module DOCM et envoi d'un message à l'utilisateur.

sortie :

néant sinon message à l'utilisateur en cas d'erreur.

Module DOCM-C.

entrée :
- commande utilisateur
- identification du fichier temporaire.

objectif :
ce module copie la suite de lignes référencée par < identificateur suite de lignes >, après la ligne < identificateur de ligne >.

Ce module comporte trois phases principales :

1) Contrôle du format et de la validité des arguments de la commande utilisateur (appel au module CHECK-C-FORMAT). < identificateur suite de lignes > ne peut inclure la ligne fictive de début de texte.
Si la commande est correcte, aller en 2) ; retour au module DOCM et envoi d'un message à l'utilisateur sinon.

2) Contrôle de l'existence des lignes identifiées (appel au module CKEXSTLN).
Si les lignes n'existent pas, un message est envoyé à l'utilisateur et le module DOCM reprend la main, aller en 3) sinon.

3) Exécution de la commande :

a) vérification de la disponibilité de numéros en nombre suffisant entre la ligne après laquelle l'insertion a lieu et la suivante (appel au module AVAILABLE).
S'il n'existe pas assez de numéros disponibles : retour au module DOCM et envoi d'un message à l'utilisateur.
Sinon, aller en b).

b) calcul du numéro des nouvelles lignes en fonction de la "distance" entre la ligne d'insertion et la suivante.

c) vérification de l'existence des lignes référencées dans la commande utilisateur (appel à CKEXSTLN).
Si les lignes n'existent pas, retour au module appelant (DOCM) et un message d'erreur est envoyé à l'utilisateur. Aller en d) sinon.

d) pour chaque ligne à copier, la page la contenant est amenée en mémoire centrale (module MSPGMC).
Elle est alors introduite dans une page (module ADLN).

Boucle en d) jusqu'à ce que la suite entière

de lignes ait été copiée.

sortie :

néant sinon un message envoyé à l'utilisateur en cas d'erreur.

Module DOCM-D.

entrée :
- commande utilisateur
- identification du fichier temporaire.

objectif :
détruire la suite de lignes indiquée par < identificateur suite de lignes >.

Trois phases :

a) contrôle du format et de la validité des arguments de la commande utilisateur (appel au module CHECK-D-FORMAT). La ligne de début de texte ne peut être référencée.
Si la commande est valide, aller en b); retour en DOCM et envoi d'un message à l'utilisateur sinon.

b) vérification de l'existence des lignes à détruire. (module CKEXSTLN)
Si les lignes existent, aller en c); un message est imprimé et le module DOCM reprend la main sinon.

c) Si les lignes existent : destruction des lignes. Boucle sur le module DLLN ("delete line ") pour toute ligne spécifiée.

sortie :
néant ou envoi d'un message à l'utilisateur en cas d'erreur.

Module DOCM-E.

entrée :
- commande utilisateur
- identification fichier temporaire.

objectif :
l'utilisateur veut clôturer le processus d'édition.

Deux phases :

a) Vérification du format de la commande (module CHECK-E-FORMAT). Si le format est correct, aller en b); envoi d'un message à l'utilisateur et retour au module DOCM sinon.

b) Deux cas sont possibles :
- l'utilisateur n'a pas changé son texte. La sortie est alors inconditionnelle.
- ou bien, il l'a modifié.

1er cas : le texte a été sauvé, après la dernière commande modifiante, dans le fichier privilégié de sauvetage (ce concept est défini au point 2.9.). La sortie est alors immédiate.

2ème cas : l'utilisateur n'a pas sauvé le texte dans ce fichier. (remarque : il peut avoir été recopié dans un autre). Un dialogue s'établit alors avec l'utilisateur : désire-t-il encore avoir la possibilité de transférer son texte dans le fichier privilégié ? (module GET_YES_NO).

Si oui : retour au module DOCM et attente de la prochaine commande.

Sinon : le processus d'édition se termine. Les modifications apportées au fichier temporaire ne sont pas repercutées dans le fichier privilégié de sauvetage.

sortie :
néant ou dialogue avec l'utilisateur.

Module DOCM-F.

entrée :
- commande utilisateur
- identification fichier temporaire.

objectif :
rechercher l'ensemble des lignes contenant un string donné, dans une suite de lignes référencée par « identificateur suite de lignes ».
pour toute ligne de cet ensemble, l'imprimer avec son numéro à l'écran.

Trois phases :

a) Contrôle du format et des arguments de la commande "find" (appel du module CHECK-F-FORMAT). La référence à la ligne fictive de début de texte est illégale.

Si la commande est incorrecte : retour au module DOCM et un message est envoyé à l'utilisateur; aller en b) sinon.

b) Vérification de l'existence des lignes identifiées (module CKEXSTLN).

Si les lignes n'existent pas, envoi d'un message à l'utilisateur et le module DOCM reprend la main; aller en c) sinon.

c) Pour toute ligne identifiée, l'éditeur amène sa page en mémoire centrale (module MSPGMC) et recherche si le string demandé appartient ou non à la ligne.

Si oui : la ligne et son numéro sont imprimés à l'écran. L'éditeur continue en c).
Sinon : retour en c).
Si aucune ligne ne contient le string recherché, le message "search failed" est imprimé pour stipuler à l'utilisateur que la recherche a echoué.

sortie :
soit les lignes contenant la suite de caractères recherchée
soit le message d'échec de la recherche.

Module DOCM-H.

entrée :

- commande utilisateur
- noms et numéros des fichiers d'aide à l'utilisateur.

objectif :

fournir à l'utilisateur les informations concernant la (les) commande(s) spécifiée(s). A toute commande correspond un fichier d'aide.

Deux phases :

a) vérification du format de la commande utlisateur (CHECK-H-FORMAT). Si le format est correct, aller en b); retour en DOCM et envoi d'un message sinon.

b) imprimer les informations.

sortie :

les informations demandées ou un message d'erreur.

module appelant :

DOCM.

Module DOCM-I.

entrée :

- commande utilisateur
- identification du fichier temporaire.

objectif :

Insérer <int> nouvelles lignes après la ligne specifiée dans la commande de l'utilisateur (< identificateur ligne >).

Trois phases principales :

a) Contrôle du format de la commande utilisateur et de ses arguments (appel au module CHECK-I-FORMAT).

Si la commande est erronée, retour au module DOCM et un message est envoyé à l'utilisateur; aller en b) sinon.

b) Vérification de l'existence de la ligne après laquelle l'insertion doit avoir lieu (module CKEXSTLN). Si elle n'existe pas, un message est envoyé à l'utilisateur et retour en DOCM; aller en c) sinon.

c) insertion des lignes proprement dite (module INSERT-LINES).

sortie :

néant ou un message destiné à l'utilisateur en cas d'erreur.

Module DOCM-M.

entrée :

- commande uitilisateur
- identification du fichier temporaire.

objectif :

Transférer les lignes spécifiées dans la commande utilisateur (< identificateur suite de lignes >) après la ligne < identificateur ligne >.

Trois phases :

a) vérification du format de la commande et de la validité de ses arguments (module CHECK-M-FORMAT).
< identificateur suite de lignes > ne peut inclure la ligne fictive de début de texte.

Si le format est correct, aller en b); le module DOCM reprend la main et un message est envoyé à l'utilisateur sinon.

b) contrôle de l'existence des lignes spécifiées (module CKEXSTLN). Si les lignes existent : aller en c), envoi d'un message et retour au module DOCM sinon.

c) vérification de la disponibilité de numéros pour les lignes à transféerer (appel au module AVAILABLE).
Si la numérotation est possible, calculer les numéros et aller en b); envoyer un message à l'utilisateur et retourner en DOCM sinon.

d) Transfert des lignes. Les lignes ne changent pas physiquement de place mais logiquement. Le "transfert" s'effectue par modifications de pointeurs dans la liste des lignes. (appel au module JOIN-IN-LINE-LIST).

sortie :

néant ou message à l'utilisateur en cas d'erreur.

Module DOCM-N.

entrée :

- commande utilisateur
- identification du fichier temporaire.

objectif :

Renumérotation des lignes suivant le pas < int > fourni par l'utilisateur.

Deux phases :

a) Contrôle du format de la commande (module CHECK-N-FORMAT). Si le format et le pas sont corrects, aller en b); retour en DOCM et envoi d'un message à l'utilisateur sinon.

b) Vérifier si la renumérotation est possible :

- vérifier que le pas est différent de 0
- vérifier que le nouveau pas n'est pas trop grand.
  Ce contrôle nous a été imposé par la représentation en machine du numéro de ligne (un entier). La renumérotation ne peut entraîner le dépassement de la valeur maximale d'un entier. Si la renumérotation est possible, aller en c); le module DOCM reprend la main et un message est envoyé à l'utilisateur sinon.

c) Effectuer la remumérotation proprement dite. Cette dernière s'effectue par parcours de la liste des lignes.

Module DOCM-NX.

entrée :

- commande utilisateur
- identification du fichier temporaire.

objectif :

imprimer la ligne suivant la ligne courante ainsi que son numéro à l'écran. (appel à PRLNMS).

sortie :

impression d'un message en cas de fin de fichier, la ligne sinon.

Module DOCM-P.

entrée :
commande utilisateur.

objectif :
imprimer les lignes spécifiées par < identificateur suite de lignes > ainsi que leurs numéros à l'écran.

Trois phases :

a) vérification du format et de la validité des arguments de la commande introduite par l'utilisateur (module CHECK-P-FORMAT). La ligne de début de texte ne peut être référencée.
Si le contrôle est positif, aller en b) ; retour en DOCM et envoi d'un message à l'utilisateur sinon.

b) contrôler l'existence des lignes à imprimer (appel au module CKEXSTLN)
Si les lignes existent, aller en c); envoi d'un message d'erreur et retour en DOCM sinon.

c) pour toute ligne spécifiée, appel au module PRLNMS pour l'imprimer ainsi que son numéro à l'écran.
Boucle en c) jusqu'à ce que la suite de lignes ait été imprimée.

sortie :
néant ou un envoi d'un message d'erreur.

Module DOCM-S.

entrée :

- commande utilisateur
- identification du fichier temporaire.

objectif :

Substituer un string par un autre dans une suite de lignes référencées par < identificateur suite de lignes >.

Trois phases :

a) contrôle de la syntaxe de la commande utilisateur et de la validité de ses arguments (appel au module CHECK-S-FORMAT). La référence à la ligne fictive de début de texte est illégale. Si erreur, envoi d'un message à l'utilisateur et retour en DOCM, aller en b) sinon.

b) vérification de l'existence des lignes spécifiées dans la commande (appel au module CKEXSTLN).
Si les lignes n'existent pas, retour en DOCM et envoi d'un message à l'utilisateur, aller en c) sinon.

c) pour toute ligne identifiée :

- l'éditeur la recopie en mémoire centrale (module MS-COPY-LINE)

- vérifie si elle contient le string à substituer (module FNSR)
  Si oui : le string est substitué dans la ligne (appel à SHIFT pour permettre la substitution). La longueur maximale imposée par la configuration du terminal ne peut être dépassée. Sinon, l'éditeur passe à la ligne suivante et retourne en c).
  l'ancienne ligne est détruite (module DLLN)
  la nouvelle ligne ainsi constituée est introduite dans une page (appel au module ADLN).

sortie :

néant sauf message à l'utilisateur en cas d'erreur.

Remarque : cette solution peut entraîner du travail inutile. Elle est valable lorsqu'il n'existe plus assez de place libre dans la page pour effectuer la substitution. Elle a cependant été choisie pour sa simplicité d'implémentation.

Module DOCM-V.

entrée :

- commande utilisateur
- identification du fichier temporaire.

objectif :

L'utilisateur veut sauver l'état actuel de son texte dans le fichier [< nom de fichier >].

Deux phases :

a) contrôle du format de la commande "save" (appel au module CHECK-V-FORMAT)
Si le format est correct, aller en b); retour à DOCM et envoi d'un message à l'utilisateur sinon.

b) vérification de la possibilité d'effectuer la commande (appel à CHOOSE-SAFE-FILE).

Si le nom de fichier est absent ou identique au nom du fichier edité, l'état actuel du texte est sauvé dans ce fichier (module SAVE).
Si le nom de fichier fourni correspond à un fichier inexistant ou à un fichier dans lequel l'utilisateur a déjà copié son texte précédemment, le sauvetage a lieu dans ce fichier (module SAVE).
Sinon, l'éditeur refuse la commande. L'utilisateur veut sauver son texte dans un fichier existant avant l'édition.

sortie :

néant sauf l'envoi d'un message à l'utilisateur en cas d'erreur.

4.3. MODULES DE NIVEAUX SUPERIEURS.

4.3.1. Principes de définition.

Nous allons maintenant définir les modules de niveaux supérieurs.

La démarche est "descendante". A partir d'un module, tous les modules activés sont décrits en séquence. Lorsqu'ils sont appelés par la suite, à partir d'autres, leur seul nom est mentionné. L'enchaînement est valorisé grâce au concept de "module appelant".

La notion de pointeur, utilisée par la suite doit être prise au sens large de "référence vers".

La description des modules est la suivante :

- ses arguments en entrée
- son objectif
- ses arguments en sortie
- son (ses) module(s) appelant(s)

4.3.2. Enumération des différents modules.

Module SKWH.

entrée :
pointeur vers une suite de caractères.

objectif :
passer les caractères "blancs" et/ou de tabulation.

sortie :
retourne un pointeur vers le premier caractère différent des deux caractères pré-cités.

module appelant :
DOCM
Tous les modules de vérification de format du type CHECK-X-FORMAT .

Module GETL.

entrée :
- informations sur le fichier origine (édité)
- pointeur vers la zone mémoire centrale où la ligne issue du fichier doit être copiée.

objectif :
copier un ligne extraite d'un fichier vers une zone de mémoire centrale.

sortie :
retourne 1 si l'opération s'est bien déroulée, un code d'erreur sinon.

module appelant :
GENERATE

Module ADLN.

entrée :

- pointeur vers l'élément de la liste des lignes correspondant à la ligne après laquelle l'insertion doit avoir lieu
- numéro de la nouvelle ligne à insérer
- pointeur vers la liste des pages.

objectif :

ajouter une ligne dans la première page contenant la place nécessaire pour cette insertion et mettre à jour les différentes listes.
Pour cela, plusieurs étapes :

- calculer la longueur de la ligne à insérer (appel au module LNLG)
- rechercher la première page contenant suffisamment de place libre (module FNPG)
- ajouter la ligne dans la page (module ADLNPG)
- mettre à jour la liste des lignes (insertion d'un nouvel élément) et la liste des pages : le nombre de caractères occupés a augmenté dans cette page.

sortie :

retourne 0 en cas d'erreur d'entrée-sortie, 1 sinon.

modules appelants :

INSERT-LINES
DOCM-C
DOCM-S

Module LNLG.

entrée :

pointeur vers une suite de caractères terminée par un "carriage return".

objectif :

calcule la longueur, le nombre de caractères, de la suite fournie en entrée.

sortie :

retourne cette longueur.

module appelant :

ADLN

Module FNPG.

entrée :

- nombre de caractères libres à rechercher
- pointeur vers l'élément de tête de la liste des pages.

objectif :

rechercher la première page contenant assez de place libre pour accueillir une nouvelle ligne de longueur donnée. Cette recherche s'effectue par parcours de la liste des pages.

Si la recherche est infructueuse, l'éditeur crée une nouvelle page. Il recopie, ,en mémoire de masse, la page actuellement en mémoire centrale pour la sauver et introduit un nouvel élément dans la liste des pages.
Sinon, il initialise un pointeur vers l'élément de cette liste, correspondant à la page retenue.

sortie :

retourne 0 en cas d'erreur d'entrée-sortie ou s'il n'existe plus de place mémoire disponible, 1 et un pointeur vers l'élément de la liste des pages sinon.

module appelant :

ADLN

Module ADLNPG.

entrée :

- longueur de la ligne à insérer
- pointeur vers la zone contenant la chaîne de caractères à introduire
- pointeur vers l'élément de la liste des pages correspondant à la page dans laquelle l'insertion doit avoir lieu.

objectif :

ajouter une suite de caractères dans la page indiquée.

Pour cela :

- amener la page en mémoire centrale (appel à MSPGMC)
- ajouter la suite de caractères dans cette page. Elle est introduite à partir du premier caractère libre situé en bas de page.

sortie :

ce module retourne 0 en cas d'erreur (entrée-sortie), 1 sinon.

module appelant :

ADLN

Remarque : cette technique est évaluée au chapitre 6.

Module MSPGMC.

entrée :

- pointeur vers la zone réceptrice en mémoire centrale
- pointeur vers l'élément de la liste des pages correspondant à la page qui doit être amenée en mémoire centrale.

objectif :

l'amener en mémoire centrale et la placer dans une zone reservée à cet effet.

Si la page actuellement en mémoire centrale ne correspond pas à la page demandée, l'éditeur la recopie en mémoire de masse et amène la page désirée en mémoire centrale (appel au module INPUT-OUTPUT).

sortie :

retourne 0 en cas d'erreur d'entrée-sortie, 1 sinon.

modules appelants :

ADLNPG
DOCM-F
DDLN
PRLNMS

Module INPUT-OUTPUT.

entrée :
- numéro de la page à lire ou écrire en mémoire de masse
- mode d'input: 0 pour une écriture, 1 pour une lecture.
- pointeur vers la zone réceptrice pour une lecture.

objectif :
lire ou écrire en mémoire de masse la page de numéro donné.

sortie :
ce module retourne la valeur 0 en cas d'erreur d'entrée-sortie, 1 sinon.

module appelant :
MSPGMC

Module GETIN.

entrée :
pointeur vers une zone réceptrice.

objectif :
aller chercher la ligne introduite au terminal par l'utilisateur et la placer dans la zone réceptrice.

sortie :
message à l'utilisateur en cas d'erreur d'entrée - sortie, néant sinon.

modules appelants :
DOCM
INSERT-LINES

Module CHECK-A-FORMAT.

entrée :
commande utilisateur.

objectif :
contrôler le format de la commande "append".

sortie :

retourne 1 si le format est correct , 0 sinon.

module appelant :
DOCM-A

Module INSERT-LINES.

entrée :

- pointeur vers l'élément de la liste des lignes correspondant au lieu d'insertion.
- nombre de lignes à insérer.

objectif :

insérer la suite de lignes introduites au terminal par l'utilisateur après la ligne identifiée .

Cette insertion s'effectue en plusieures étapes :

- vérifier s'il existe encore assez de numéros disponibles pour permettre l' insertion et la numérotation des lignes à introduire (appel au module AVAILABLE).

- calculer le pas pour déterminer la numérotation des lignes.

- aller chercher la nouvelle ligne, terminée par un "carriage return", introduite au terminal par l'utilisateur (module GETIN).

- ajouter la ligne dans la première page contenant suffisamment de place libre pour l'accueillir (appel à ADLN).

Ces deux dernières opérations seront repétées jusqu'à ce que:

- le nombre de lignes à introduire aiteété atteint
- ou que l'utilisateur ait introduit le signal de fin d'insertion de nouvelles lignes. Le seul caractère "." placé en première position d'une ligne.

sortie :

néant ou envoi d'un message à l'utilisateur en cas d'erreur d'entré-sortie.

modules appelants :

DOCM-A
DOCM-I

Module AVAILABLE.

entrée :
pointeur vers un élément de la liste des lignes.

objectif :
calculer le nombre de numéros de lignes disponibles entre la ligne pointée et la ligne suivant cette dernière.
par ex. : soit la ligne de numéro x et la suivante de numéro égal à x+y. Ce module calcule le nombre de numéros encore disponibles comme étant égal à ((x+y) - x) - 1.

sortie :
le module retourne le nombre ainsi calculé.

modules appelants :
INSERT-LINES
DOCM-M
DOCM-C.

Module CHECK-C-FORMAT.

entrée :
- commande utilisateur
- pointeur vers la liste des lignes.

objectif :
Ce module vérifie le format de la commande "copy".
Ce contrôle inclut l'anlyse de la validité des identificateurs.
Pour l'identificateur d'une suite de lignes : appel au module CKIDSTLN.
Le module CKIDLN est appelé lorsqu'il s'agit de l'identificateur d'une seule ligne.

sortie :
ce module retourne 1 ainsi que les numéros de lignes (obtenus par appel de CKIDSTLN et CKIDN) si le format est correct, 0 et envoie un message à l'utilisateur sinon.

module appelant :
DOCM-C

Module CKIDSTLN.

entrée :

- ce module reçoit un pointeur vers < identificateur suite de lignes >
- pointeur vers la liste des lignes.

objectif :

vérifie si la suite de caractères fournie (du type X : Y) correspond bien à celle d'un identificateur d'une suite de lignes.
Le contrôle effectué ici, consiste à vérifier que X et Y sont valides et séparés par " : ". Ils sont transformés en la valeur numérique correspondante (appel à CKIDLN).

sortie :

le module retourne les numéros (valeurs) des lignes correspondantes à X et Y, un pointeur vers le caractère suivant l'identificateur de la suite de lignes s'il est correct. Le pointeur "null" est renvoyé en cas d'erreur.

modules appelants :

CHECK-C-FORMAT
CHECK-D-FORMAT
CHECK-F-FORMAT
CHECK-M-FORMAT
CHECK-P-FORMAT
CHECK-S-FORMAT

Module CKIDLN.

entrée :
- un pointeur vers le premier caractère de l'identificateur supposé d'une ligne < identificateur ligne >
- pointeur vers la liste des lignes.

objectif :
Tout identificateur numérique (ISDIGIT) ou spécial (ISPECIAL) , est transformé en un nombre (appel respectivement à CVDG ou CVSP).

sortie :
le numéro de la ligne ainsi spécifiée, un pointeur vers le caractère suivant l'identificateur s'il est correct, le pointeur "null" sinon.

modules appelants :
CKIDSTLN
CHECK-I-FORMAT
CHECK-M-FORMAT
CHECK-C-FORMAT

Module ISDIGIT.

entrée :
un caractère.

objectif :
contrôler que ce caractère est numérique ou non.

sortie :
le module retourne 1 si le caractère est numérique, 0 sinon.

Module ISPECIAL.

entrée :
un caractère

objectif :
vérifier que le caractère fourni en entrée correspond à un identificateur spécial (b, c, f ou l).

sortie :
si oui : le module retourne la valeur 1, 0 sinon.

Module CVDG.

entrée :
pointeur vers une suite de caractères numériques.

objectif :
convertit une chaîne de caractères numériques en sa valeur numérique..
Le nombre ainsi obtenu doit être inférieur à la valeur maximale d'un entier. Cette contrainte est imposée par la représentation du numéro de ligne en machine (un entier).

sortie :
retourne la valeur du numéro de ligne et un pointeur vers le caractère suivant le dernier caractère numérique si pas d'erreur, le pointeur "null" sinon.

modules appelants :
CKIDLN
CHECK-I-FORMAT

Module POWER.

entrée :
deux nombres x et y.

objectif :
exposer x à la yème puissance.

sortie :
retourne le nombre ainsi obtenu.

Module CVSP.

entrée :

- pointeur vers un identificateur spécial d'une ligne (b, c, f ou l)
- pointeur vers la liste des lignes.

objectif :

parcours de la liste des lignes pour obtenir le numéro de la ligne ainsi identifiée.

sortie :

- le numéro de ligne ainsi trouvé
- un pointeur vers le caractère suivant cet identificateur.

module appelant :

CKIDSTLN

Module CKEXSTLN.

entrée :

- deux numéros de lignes. Ils correspondent aux identifiants d'une suite de lignes.
- pointeur vers l'élément de fin de liste des lignes.

objectif :

ce module vérifie l'existence de ces lignes.
Par exemple, soient x et y, les deux numéros fournis. Le module vérifie que la ligne identifiée par x est bien située avant celle identifiée par y et qu' elles existent toutes deux. Ces opérations s'effectuent par parcours de la liste des lignes.

sortie :

retourne la valeur 0 si les conditions exprimées ci-dessus ne sont pas remplies.
Sinon, le module retourne le nombre de lignes comprises entre les lignes réellement identifiées par x et y inclues ainsi qu' un pointeur vers l'élément de la liste des lignes correspondant à la ligne de numéro x.

modules appelants :

DOCM-C
DOCM-D
DOCM-F
DOCM-I
DOCM-M
DOCM-S

Module MS-COPY-LINE.

entrée :

- un pointeur vers l'élément de la liste des lignes correspondant à la ligne qui doit être copiée
- un pointeur vers la zone dans laquelle la ligne doit être reproduite, temporairement, en mémoire centrale. centrale

objectif :

copier une ligne dans une zone donnée. La page contenant la ligne n'est pas nécessairement en mémoire centrale.

Pour ce faire :

- le module FNPTPGLN positionne un pointeur dans la liste des pages nécessaire à MSPGMC pour amener la page en mémoire centrale.
- copie la ligne en bas de cette page (appel à COPY-LINE).

sortie :

néant sinon un message en cas d'erreur d'entrée-sortie.

modules appelants :

DOCM-C
DOCM-S

Module COPY-LINE.

entrée :
- pointeur vers le premier caractère de la ligne origine
- pointeur vers le début de la zone réceptrice.

objectif :
copie une ligne, terminée par un "carriage return", de la zone d'origine vers la zone réceptrice.

sortie :
néant.

module appelant :
MS-COPY-LINE

Module CHECK-D-FORMAT.

entrée :
- commande utilisateur
- pointeur vers la liste des lignes.

objectif :
contrôler la syntaxe de la commande utilisateur et la validité de ses arguments (appel à CKIDSTLN).

sortie :
retourne 1 ainsi que les numéros des lignes obtenus par appel à CKIDSTLN si le format est correct, 0 sinon.

module appelant :
DOCM-D

Module DLLN.

entrée :

pointeur vers l'élément de la liste des lignes correspondant à la ligne à détruire

objectif :

detruction de la ligne ainsi pointée.

Plusieurs étapes :

- positionnement (FNPTPGLN) du pointeur dans la liste des pages pour permettre à MSPGMC d'amener la page contenant la ligne en mémoire centrale.

- destruction physique de la ligne dans la page par un shift (appel au module SHIFT)

- mise à jour de la liste des lignes :
  suppression de l'élément correspondant à la ligne à détruire
  mise à jour des déplacements des autres lignes de la page (appel au module UPDATE-LINE-LIST)

- mise à jour de la liste des pages : le nombre de caractères occupés dans cette page doit être diminué du nombre de caractères libérés.

sortie :

retourne la valeur 0 et envoie un message à l'utilisateur en cas d'erreur d'entrée - sortie, 1 sinon.

modules appelants :

DOCM-D
DOCM-S

MODULE SHIFT.

entrée :
- pointeur vers le premier caractère à shifter
- direction du shift (vers la gauche ou la droite) et nombre de positions à décaler.

objectif :
décaler de x positions vers la droite ou la gauche les caractères situés après un caractère donné.

sortie :
néant.

modules appelants :
DOCM-S
DLLN

Module UPDATE-LINE-LIST.

entrée :
- pointeur vers la liste des lignes
- nombre de caractères détruits.

objectif :
mettre à jour les déplacements dans la liste des lignes après un shift. Pour toute ligne dont le numéro de page égale celle où il y a eu un shift et dont le déplacement est supérieur à celui du dernier caractère de la ligne décalée, le déplacement est diminué du nombre donné en entrée du module..
Cette mise à jour s'effectue par parcours de la liste des lignes.

sortie :
néant.

modules appelants :
DLLN
DOCM-S

Module JOIN-IN-LINE-LIST.

entrée :
pointeurs vers deux éléments de la liste des lignes.

objectif :
établir le chaînage double entre ces deux éléments.

sortie :
néant.

modules appelants :
DLLN
DOCM-M

Module CHECK-E-FORMAT.

entrée :
commande utilisateur.

objectif :

contrler la syntaxe de la commande "end".

sortie :
retourne 1 si la syntaxe de la commande est exacte, 0 et un message à l'utilisateur sinon.

module appelant :
DOCM-E

Module GET-YES-NO.

entrée :
réponse de l'utilisateur

objectif :
aller chercher la réponse au terminal et la contrôler (module IN-AND-CHECK).
Le module attend que la réponse soit valide.

sortie :
Si elle est valide et égale à "oui", il retourne la valeur 1; 0 si la réponse est "non".

module appelant :
DOCM-E

Module IN-AND-CHECK.

entrée :
réponse de l'utilisateur

objectif :
aller chercher la réponse au terminal et vérifier qu'elle est valide.

sortie :
le module retourne 0 si la réponse est "non", 1 si elle est "oui", 2 et envoie un message si elle est impossible.

module appelant :
GET-YES-NO

Module CHECK-F-FORMAT.

entrée :
commande utilisateur

objectif :
- contrôler la syntaxe de la commande utilisateur et la validité de ses arguments (appel à CKIDSTLN)
- sauver la suite de caractères à rechercher (module SVSR). Cette suite est sauvée à cet instant pour éviter un balayage supplémentaire de la commande par la suite.

sortie :
le module retourne 1 et les valeurs retournées par CKISTLN si la commande est correcte, 0 sinon.

module appelant :
DOCM-F

Module SVSR.

entrée :
- pointeur vers la suite de caractères dans la commande utilisateur
- pointeur vers la zone où cette suite doit être sauvée.

objectif :
- sauver cette suite de caractères dans une zone donnée dont le pointeur est fourni en entrée
- vérifier qu'elle est bien terminée par " ^ ".

sortie :
ce module retourne un pointeur vers le caractère suivant le "^" final si celui-ci a été renconré, le pointeur "null" sinon.

modules appelants :
CHECK-F-FORMAT
CHECK-S-FORMAT

Module FNSR.

entrée :
- pointeur vers le premier caractère de la ligne où la recherche doit avoir lieu
- pointeur vers la suite de caractères demandée ainsi que sa longueur.

objectif :
parcourir une ligne et rechercher si la suite donnée appartient ou non à cette ligne.

sortie :
retourne un pointeur vers le premier caractère de la suite dans la ligne si la recherche a été fructueuse, le pointeur "null" sinon.

modules appelants :
DOCM-F
DOCM-S

Module PRLN.

entrée :
pointeur vers un élément de la liste des lignes.

objectif :
imprimer cette ligne ainsi que son numéro au terminal de l'utilisateur. La page contenant cette ligne est en mémoire centrale.

sortie :
ce module retourne la valeur 0 si il y a eu une erreur au niveau de l' impression au terminal, 1 sinon.

modules appelants :
PRLNMS
DOCM-F

Module CHECK-H-FORMAT.

entrée :
- commande utilisateur
- nom et descripteur des fichiers d'aide.

objectif :
contrôle du format de la commande et de la validité de ses arguments (avec appel au module STRCPY).

sortie :
retourne 1 et le nom du fichier contenant les informations sur la (les) commande(s) demandée(s), 0 sinon.

Module appelant :
DOCM-H

Module STRCPY.

entrée :
pointeur vers les zones émettrice et receptrice.

objectif :
copier les caractères de la première zone dans la seconde.

sortie :
néant.

module appelant :
CHECK-H-FORMAT
CHECK-V-FORMAT

Module CHECK-I-FORMAT.

entrée :
- commande utilisateur
- identification du fichier temporaire.

objectif :
ce module contrôle la syntaxe de la commande "insert" et la validité de ses arguments. Appel au module CKIDLN pour vérifier l'identificateur de ligne spécifié dans la commande. Convertit la chaîne de caractères représentant le nombre de lignes à insérer en un nombre (appel à CVDG).

sortie :
ce module retourne le numéro de la ligne, le nombre de lignes à insérer et la valeur 1 si la commande est correcte, 0 sinon.

module appelant :
DOCM-I

Module CHECK-M-FORMAT.

entrée :
- commande utilisateur
- identification du fichier temporaire.

objectif :
contrôler le format de la commande "move" et de ses arguments (appel à CKIDSTLN pour les lignes à transférer et à CKIDLN pour la ligne après laquelle ce transfert aura lieu).

sortie :
ce module retourne 1 ainsi que les numéros obtenus si la commande est valide, 0 sinon.

module appelant :
DOCM-M

Module CHECK-N-FORMAT.

entrée :
commande utilisateur.

objectif :
contrôle de la commande et de ses arguments. Convertit la suite de caractères représentant le nouveau pas en un nombre (appel à CVDG).

sortie :
le module retourne 1 et la valeur du nouveau pas si la syntaxe est correcte, 0 sinon.

module appelant :
DOCM-N

Module CHECK-P-FORMAT.

entrée :
- commande utilisateur
- identification du fichier temporaire.

objectif :

contrôle syntaxique de la commande "print" et de la validité de ses arguments (appel au module CKIDSTLN).

sortie :
retourne la valeur 0 si la commande est incorrecte, 1 et les numéros de lignes retournés par CKIDSTLN sinon.

module appelant :
DOCM-P

Module PRLNMS.

entrée :
pointeur vers un élément de la liste des lignes.

objectif :
imprimer cette ligne à l'écran.

- La page contenant cette ligne n'est pas nécessairement en mémoire centrale (appel au module FNPTPGLN pour initialiser le pointeur de la liste des pages nécessaire à MSPGMC)
- quand la page est en mémoire centrale, appel au module PRLN pour imprimer la ligne.

sortie :
le module retourne la valeur 0 en cas d'erreur d'entrée - sortie, 1 sinon.

module appelant :
DOCM-P

Module CHECK-S-FORMAT.

entrée ;
- commande utilisateur
- identification du fichier temporaire.

objectif :
- vérifier la syntaxe de la commande utilisateur et de l'identificateur de la suite de lignes (module CKIDSTLN)
- sauver les strings substituant et substitué (appel au module SVSR).

sortie :
- le module retourne 1 et les valeurs retournées par CKIDSTLN si la commande est correcte, 0 sinon
- pointeur vers les string.

module appelant :
DOCM-S

Module CHECK-V-FORMAT.

entrée :
commande utilisateur.

objectif :
contrôler la syntaxe de la commande "save", avec un appel aSKFINA.

sortie :
ce module retourne 1 et le nom du fichier de sauvetage si la commande est correcte, 0 sinon.

module appelant :
DOCM-V

Module SKFINA.

entrée :
pointeur vers une suite de caractères.

objectif :
rechercher le premier caractère qui est soit un blanc, soit un tabulateur, soit un "carriage return".

sortie :
le module retourne un pointeur vers ce dernier.

Module appelant :
CHECK-V-FORMAT.

Module CHOOSE-SAFE-FILE.

entrée :
- nom du fichier de sauvetage [< nom de fichier >]
- pointeur vers le début de la liste des fichiers sauvés
- nom du fichier édité.

objectif :
contrôler que [< nom de fichier >] est valide.

Plusieurs phases :

a) vérifier si [< nom de fichier >] appartient à la liste des fichiers de sauvetage (appel à STRCMP). Si oui, retourner un pointeur vers l'élément correspondant. Sinon, aller en b).

b) si [< nom de fichier >] est celui du fichier edité, renvoyer son descripteur, retour en DOCM-V sinon.

c) si un fichier de ce nom existe déjà, envoyer un message à l'utilisateur lui spécifiant le conflit. Le module retourne alors un descripteur égal à 0. Sinon, aller en d).

d) créer ce fichier, retourner son descripteur, générer et introduire un nouvel élément dans la liste des fichiers de sauvetage (ADD-SAFE-FILE). La valeur 0 est renvoyée s'il n existe plus assez de place pour créer ce nouveau membre.

sortie :
soit un descripteur de fichier soit 0 en cas d'erreur.

module appelant :
DOCM-V

Module STRCMP.

entrée :
pointeurs vers deux suites de caractères.

objectif :
déterminer si elles sont identiques ou non.

sortie :
retourne 0 en cas d'équivalence, une autre valeur sinon.

Modules appelants :
CHOOSE-SAFE-FILE
SEARCH-SAVE

Module SEARCH-SAFE.

entrée :
- le nom de fichier [< nom de fichier >]
- pointeur vers l'élément de tête de la liste des fichiers de sauvetage.

objectif :
rechercher si le fichier appartient ou non à cette liste (avec appel àSTRCMP).

sortie :
si oui, le module retourne un pointeur vers l'élément correspondant, le pointeur "null" sinon.

module appelant :
CHOOSE-SAFE-FILE

Module ADD-SAFE-FILE.

entrée :
- pointeur vers la liste des fichiers de sauvetage
- pointeur vers le nouvel élément crée.

objectif :
ajouter l'élément pointé dans la liste.

sortie :
si oui, le module retourne le descripteur de fichier trouvé dans la liste, 0 sinon.

module appelant :
CHOOSE-SAFE-FILE

Module SAVE.

entrée :
- identification du fichier temporaire (son nom et son descripteur)
- descripteur du fichier de sauvetage.

objectif :
recopier le fichier temporaire dans le fichier de sauvetage.

a) pour toute ligne, la page la contenant est amenée en mémoire centrale (appel à FNPTPGLN et à MSPGMC).

b) elle est injectée dans le fichier (module PUTL).

sortie :
le module retourne 0 en cas d'erreur d'entrée-sortie, 1 sinon.

module appelant :
SAVE

Module PUTL.

entrée :
- descripteur du fichier de réception
- pointeur vers la ligne à transférer (terminée par un "carriage return").

objectif :
copier cette ligne dans le fichier.

sortie :
le module retourne 0 et imprime un message à l'utilisateur en cas d'erreur d'entrée-sortie, 1 sinon.

module appelant :
SAVE

4.4. REMARQUES.

Une telle découpe est caractéristique d'une analyse " top - down ".

Au vu de ce chapitre, le lecteur aura certainement remarqué le grand nombre de modules définis.

Plusieurs raisons nous ont dicté cette démarche :

- les modifications, de format entre autres, sont localisables à un ou plusieurs modules.
- l'insertion de nouvelles commandes est aisée : ajout d'un module de niveau 3 et, si nécessaire, de plus bas niveau.
- ces modules facilitent la compréhension de la structure des traitements.

Les formats vérifiés par les modules CHECK-X-FORMAT sont ceux définis au point 2.11.2.

5. IMPLEMENTATION.

(Michel Lestrade)

5. IMPLEMENTATION.

5.1. Introduction.

Dans ce chapitre, nous allons d'abord donner un bref historique du langage (5.2.1.). Un resumé succint de ses caractèristiques est mentionné en 5.2.2.

Vient ensuite une brève introduction aux programmes pour faciliter la lecture et la comprèhension de ces derniers.

5.2. Le langage d'implémentation.

5.2.1. Historique du langage "C" ([3]).

Le langage C a été développé initialement sur le système d'exploitation UNIX tournant sur les machines DEC PDP-11. IL a été réalisé et implementé par Denis Ritchie.

Le système UNIX, le compilateur C et, en général, tous les programmes d' applications UNIX sont écrits en C. Cependant, des compilateurs existent aussi sur d'autres machines, l' IBM/370, l'HONEYWELL 6000 par exemple.

La plupart des idées reprises pour développer le langage C proviennent du langage BCPL développé par Martin Richards. Cette influence du BCPL sur C a également conduit indirectement au développement du langage B par Ken Thompson en 1970, pour le

premier système UNIX sur PDP-7.

En règle générale, le langage C a été associé au système UNIX puisque ce système ainsi que son software sont écrits en C.

5.2.2. Quelques caractéristiques de ce langage.

Les types de données sont des caractères, des entiers et des nombres en virgule flottante. En plus, il existe une hiérarchie de types de données dérivées créées avec des pointeurs, tableaux, structures et fonctions.

Le langage C dispose d'instructions de contrôle pour permettre une bonne structuration des programmes. Ces instructions sont : IF, WHILE, FOR, DO et SWITCH.

Toutes les routines C peuvent être définies récursivement.

C ne dipose pas de la structure de blocs comme l'ALGOL par exemple. Les procédures ne s'emboitent pas. Un programme C comprend une ou plusieurs procédures. Ceci permet une bonne modularisation du programme. Chaque procédure peut être compilée séparément.

Les arguments des fonctions sont passés par valeur c'est-à-dire en recopiant la valeur de l' (des) argument(s). Il est ainsi impossible à la fonction appelée de modifier la valeur actuelle du paramètre dans la fonction appelante.
Pour réaliser un appel par référence, un pointeur peut être passé. La fonction appelée peut changer l'objet désigné par le pointeur.

Les noms de tableaux sont passés comme la localisation du premier élément du tableau. Il s'agit donc bien d'un appel par référence.

La définition de tableaux dynamiques n'est pas permise. La place réservée est fixe et est connue à la compilation. L'emploi de pointeurs permet des procédures travaillant sur des tableaux de longueurs différentes.

Le langage C cependant, n'est pas un langage de " très haut niveau ". Il ne fournit pas de possibilités de multiprogrammation, pas d'opérations parallèles, pas de mécanismes de synchronisation ni de coroutines.
Tous ces mécanismes de " très haut niveau " peuvent être fournis par la définition de fonctions appropriées.

Pour les raisons énumérées ci-dessus, manipulation d'objets tels caractères , nombres, adresses, une structuration aisée et de plus comme nous disposions d'un système UNIX, le langage d'implémentation est le langage C.

5.3. Implémentation.

Le but de ce paragraphe n'est pas de décrire les progammes (fournis en annexe 1) mais d'expliquer quelque peu les techniques employées.

La découpe du progamme en fonctions est basée sur l'architecture décrite dans le chapitre 4.

Les pointeurs C ont été fréquemment utilisés pour réaliser des appels par référence.

Le concept de "structure" défini dans ce langage nous a permis d'implémenter facilement les différentes listes. La définition dans une structure, d'un pointeur vers une autre étant permise.

Une suite d'informations, associée au fichier temporaire, a été regroupée dans une structure. Elle est constituée de son nom, son descripteur, des pointeurs vers les listes, le numéro de la page en mémoire centrale, un pointeur vers la zone destinée à la recevoir, le nombre de lignes dans le fichier et enfin, le pas de numérotation.

Un pointeur vers cette structure est passé à un grand nombre de procédures. Cette technique permet de "personaliser" le fichier temporaire. Lors d'extensions futures on pourrait ainsi imaginer de ne plus travailler avec un mais plusieurs fichiers temporaires

Une même structure, bien que remplie incompletement, a été créée pour le fichier édité pour les raisons précitées.

Les variables globales de l'éditeur sont constituées d'une suite de paires de caractères. Chacune d'entre elles est significative. (cfr annexe 2).

6. EVALUATION ET EXTENSIBILITE.

(Serge Debacker)

(Michel Lestrade)

6. EVALUATION ET EXTENSIBILITE.

Cette partie a pour but essentiel de dresser un bilan général sur le travail réalisé dans le cadre de ce mémoire.

Nous évaluons, d'une part, les choix proprement dits (6.1.) et, d'autre part, l'apport personnel que peut représenter une telle expérience (6.2.).

6.1. Evaluation du travail.

De la notion même d'éditeur, nous avons pu dégager quatre concepts clés qui ont permis de structurer l'analyse du problème posé.

Un assez grand niveau d'abstraction a été atteint grâce à l'introduction de l'idée de potentiel. Par la suite, il pourrait s'avérer intéressant d'approfondir celle-ci d'un point de vue critique.

Comme nous l'avons mentionné précédemment (2.2.), nous avons choisi de concevoir un outil minimal mais extensible. De fait, il est parfaitement possible, tant au niveau de l'analyse fonctionnelle qu'organique, d'étoffer l'ensemble actuel des commandes.

Dans cet ordre d'idée, citons parmi les nombreuses possibilités l'introduction d'un mode visuel, l'édition simultanée de plusieurs fichiers et l'existence d'informations par défaut dans le format des commandes.

De plus, les effets de certaines d'entre elles peuvent être combinés pour découler sur des mécanismes encore plus puissants. Nous pensons notamment à toutes les formes de modification interactive par recherche.

- - - -

Voyons maintenant l'appréciation de la structure des données.

Nous avons exposé de manière précise (partie 3) la représentation des données en structures de listes. Les raisons principales qui nous ont poussés vers un tel choix sont multiples.

D'une part, il est assez normal d'associer à chaque ligne un élément dans une liste chaînée. Ainsi, à la destruction d'une ligne correspond la suppression d'un membre de la liste. L'enchaînement des éléments dans la liste est calqué sur l'ordre de succession des lignes du texte.

D'autre part, les algorithmes de gestion (ajout, suppression et recherche) sont relativement simples pour de telles structures.

Ce sont les mêmes raisons qui nous ont incités à adopter une organisation similaire pour les informations sur les pages et les fichiers de sauvetage.

- - - -

Malgré de tels avantages, il faut noter certains inconvénients.

Tout d'abord, la structure de liste ne permet pas d'accès par clé et nécessite l'attribution d'une taille de mémoire centrale

assez élevée.

Ensuite, la gestion des pages telle que nous l'avons imaginée est une version très simplifiée de la mémoire virtuelle puisque le buffer prévu ne contient qu'une et une seule page.

Enfin, la répartition des lignes dans les pages est trop éclatée, en ce sens qu'il n'existe pas de proximité physique des lignes correspondant à leur proximité logique.

Néanmoins, nous verrons au point 6.2. que notre approche n'est pas dénuée de tout intérêt.

- - - -

Une solution qui pourrait sembler plus adéquate est l'utilisation d'un dictionnaire travaillant par niveaux d'index.

Dans ce cas, l'accès peut être plus rapide grâce à une clé et on peut s'arranger pour que la table prenne une place moins importante en mémoire centrale.

Notons également une amélioration possible qui pourrait contribuer à un meilleur temps de réponse.

De même qu'un mécanisme permet de savoir si la version finale du texte a été sauvée dans le cadre des commandes "v" et "e", nous aurions pu envisager une méthode analogue pour éviter le recopiage systématique du buffer en cas de défaut de page.

6.2. Evaluation de l'apport personnel.

L'expérience apportée par le travail réalisé concerne trois domaines.

Tout d'abord, c'est pour la première fois que nous avons pu développer complètement une application au travers des différentes phases d'un processus informatique. C'est ainsi que la distinction entre les différentes étapes (analyse fonctionnelle, analyse organique et implémentation) nous est apparue concrètement.

Ensuite, nous avons été confrontés à des problèmes de choix inhabituels pour nous et qui existent souvent dans le développement de projets.

Enfin, l'étude d'un nouveau langage nous a été enrichissante à plusieurs points de vue. En effet, l'utilisation du langage C nous a permis d'employer des types nouveaux de données et d'actions.

BIBLIOGRAPHIE.

(1) Michel E. DEBAR, PRIMER FOR TWENEX, Computing Center, Namur University (BELGIUM), February 7, 1980.

(2) Christopher W. FRASER, "A Generalized Text Editor", Communications of the ACM, Vol. 23, March 1980, pp. 154 - 158.

(3) Brian W. KERNIGHAN & P.J. PLAUGER, SOFTWARE TOOLS, Addison-Wesley, 1976.

(4) Brian W. KERNIGHAN & Dennis M. RITCHIE, THE C PROGRAMMING LANGUAGE, Prentice-Hall, New Jersey, 1978.

(5) Donald E. KNUTH, THE ART OF COMPUTER PROGRAMMING, Addison-Wesley, 2d edit., 1976.

(6) Richard M. STALLMAN, EMACS, THE EXTENSIBLE, CUSTOMIZABLE, SELF-DOCUMENTING DISPLAY EDITOR, Massachusetts Institute of Technology, June 22, 1979.

(7) UNIX PROGRAMMERS MANUAL, University of Warwick, England.

(8) UNIX PROGRAMMERS MANUAL, F.N.D.P., Namur.

(9) Manuel utilisateur TEXT EDITING UTILITIES, I.B.M., éditeurs de texte $EDIT1, $EDIT1N et $FSEDIT.

(10) Documentation sur TECO : TECO.DOC.2 et TECO.REF.2, F.N.D.P., Namur.

(11) Revised Report on the Algorithmic Language ALGOL 60, Communications of the ACM.

FACULTES
UNIVERSITAIRES
N.D. DE LA PAIX

NAMUR

INSTITUT D'INFORMATIQUE

RUE GRANDGAGNAGE, 21, B - 5000 NAMUR (BELGIUM)

FACULTES UNIVERSITAIRES NOTRE-DAME DE LA PAIX - NAMUR,

INSTITUT D'INFORMATIQUE.

ETUDE ET REALISATION D'UN EDITEUR DE TEXTE.

(ANNEXES)

Promoteur : C. Cherton.

Serge Debacker

Michel Lestrade

Mémoire présenté en vue de l'obtention du grade de

LICENCIE ET MAITRE EN INFORMATIQUE.

ANNEE ACADEMIQUE 1980 - 1981.

Annexe 1 : PROGRAMMES.

pages 2 à 86 : EDIT.

pages 86a à 86c : CREAT

Remarque :

Par erreur d'impression, chaque occurrence du caractère "\n" n'apparaît pas correctement.

Par exemple, tout ordre " printf(" XXXXX 0) " doit être lu " printf(" XXXXX \n") ".

```
#define NULL 0                          /* pointer value for error report */

#define MXSZBF 512                              /* max size of buffer  */

#define DL '''                        /* delimit in user's commands */

#define MXLGLN 150                               /* max length of line */

#define MXNMLN 32767                           /* (2 to 15th power) - 1 */

#define MNNMLN 1                                 /* min numeral of line */

#define FIMO 0777                                          /* file mode */

#define EOF -1                                           /* end of file  */

struct line_list
{ int nmln; /* numeral of line */
  struct line_list *ptnxmbln; /* next member */
  struct line_list *ptpvmbln; /* previous member */
  int nmpgln; /* numeral of page */
  int dplnpg; /* displacement of line in page */
} s_line_list;

struct page_list
{ int nmpgpg; /* numeral of page */
  int nbbscr; /* number of busy characters in page */
  struct page_list *ptnxmbpg; /* next member */
} s_page_list;

struct safe_files_inf
{ int fide_f; /* file descriptor */
  char *ptfina_f; /* file name */
  struct safe_files_inf *ptnxmbsv; /* next member */
} s_safe_files_inf;

struct file_inf
{ char *ptfina; /* file name */
  int fide; /* file descriptor */
  struct line_list *pttxli; /* text list */
  struct page_list *ptpgli; /* page list */
  struct line_list *ptmbtxli; /* member of text list */
  struct page_list *ptmbpgli; /* member of page list */
  struct line_list *pt_c; /* current */
  struct line_list *pt_e; /* end */
  struct line_list *pt_b; /* begin */
  int nmpgbf; /* numeral of page in buffer */
  int nmse; /* step */
  char *ptbf; /* buffer */
  int nblnfi; /* number of lines in edited file */
  struct safe_files_inf *ptsffili; /* safe files list */
} s_file_inf;

int tsen; /* test end */
char lntm[MXLGLN + 1]; /* line from terminal */
char lsmd; /* last modifying command */
```

```
char *ptlntm; /* pointer to line terminal */
int lssffide; /* last safe file descriptor */
```

```
main(argc, argv)

int argc; /* count arguments */
char *argv[]; /* pointer to arguments */

{ extern int tsen; /* test_end */
  extern char lntm[]; /* line from user's terminal */
  extern char *ptlntm; /* pointer to lntm */

  int fd; /* edited file descriptor */

  char temp[MXLGLN]; /* temporary (scratch) file name */
  char *ptedit; /* edited file name */
  char *pttemp; /* temporary file name */

  struct file_inf *pt_edited; /* edited file */
  struct file_inf *pt_temp; /* scratch file */
  struct file_inf *pt_file_inf; /* defined for "calloc" */

  int find_name( );
  int init_data_structure( );
  int docm( );
  int tmpnam( );


  ptedit = argv[1]; /* argument = edited file name */

  pttemp = temp;
  ptlntm = lntm;

  if (argc == 1)  /* user error : no argument */
    { printf("not specified edited file0);
      exit(1);
    }

  if (argc != 2)  /* user error : too many arguments */
    { printf("too many specified files0);
      exit(1);
    }

  if ((fd = open(ptedit, 0)) == -1) /* user error : doesn't exist */
    { printf("not existing file0);
      exit(1);
    }
  else
    close(fd);

  if ((pt_file_inf = calloc(2, sizeof( s_file_inf))) == -1)
    { printf("full core memory0);
      exit(1);
    }
  else
    {   /* edited file : */
      pt_edited = pt_file_inf;
      pt_edited -> ptfina = ptedit;
```

```
        /* temporary file : */
      pt_temp = ++pt_file_inf;
      tmpnam(pttemp);
      pt_temp -> ptfina = pttemp;
    }

  tsen = (init_data_structure(pt_edited, pt_temp) == 0)
              ? 1 : 0;

  while (tsen != 1)
              /* execute next command */
    docm(pt_edited, pt_temp);

  exit(0);

}
```

```
int init_data_structure(pt_edited, pt_temp)

                /* init. data structure for editing "pt_edited" file */
                /* with scratch file "pt_temp"                       */

struct file_inf *pt_edited;
struct file_inf *pt_temp;

{ extern char lsmd; /* last modifying command */
  extern int lssffide; /* last safe file descriptor */
  int init_line_list( );
  int init_page_list( );
  int init_safe_list( );
  int ask_buffer( );
  int generate( );

  pt_edited -> fide = open(pt_edited -> ptfina, 2);

  close(creat(pt_temp -> ptfina, FIMO));
  pt_temp -> fide = open(pt_temp -> ptfina, 2);

  if (init_line_list(pt_temp) == 0)
    return(0);

  if (init_page_list(pt_temp) == 0)
    return(0);

  if (init_safe_list(pt_temp) == 0)
    return(0);

  if (ask_buffer(pt_temp) == 0)
    return(0);

  pt_temp -> nmse = 10; /* default value for step */
  pt_temp -> nblnfi = 0; /* initial number of lines in edited file */

                /* initial safe file = edited file : */
  lssffide = pt_edited -> fide;

  lsmd = 'v'; /* not yet modifying command */

  return(generate(pt_edited, pt_temp));

}
```

```
int init_line_list(pt_temp)

                /* create empty line_list for "pt_temp"  */
                /* return 1 if ok, 0 if not ok           */

struct file_inf *pt_temp;

{ struct line_list *pt_line_list; /* pointer to line list */
  struct line_list *pt_head; /* head member */
  struct line_list *pt_begin; /* begin member */
  struct line_list *pt_end; /* end member */

  if ((pt_line_list = calloc(3, sizeof(s_line_list))) == -1)
    { printf("full core memory0);
      return(0);
    }
  else
    { pt_head = pt_line_list;
      pt_begin = ++pt_line_list;
      pt_end = ++pt_line_list;

          /* head member : */
      pt_temp -> pttxli = pt_head;
      pt_head -> ptpvmbln = pt_end;
      pt_head -> ptnxmbln = pt_begin;

          /* begin member : */
      pt_temp -> pt_b = pt_begin;
      pt_begin -> ptpvmbln = pt_head;
      pt_begin -> ptnxmbln = pt_end;
      pt_begin -> nmln = MNNMLN - 1;

          /* end member : */
      pt_temp -> pt_e = pt_end;
      pt_end -> ptpvmbln = pt_begin;
      pt_end -> ptnxmbln = pt_head;
      pt_end -> nmln = MXNMLN + 1;

          /* current member : */
      pt_temp -> pt_c = pt_begin;

      return(1);
    }

}
```

```
int init_page_list(pt_temp)

                /* create empty page_list for "pt_temp"      */
                /* return 1 if ok, 0 if not ok               */

struct file_inf *pt_temp;

{ struct page_list *pt_page_list; /* pointer to safe file list */

  if ((pt_page_list = calloc(sizeof(s_page_list))) == -1)
    { printf("full core memory0);
      return(0);
    }
  else
    { pt_page_list -> nmpgpg = -1;
      pt_page_list -> ptnxmbpg = pt_page_list;

      pt_temp -> ptpgli = pt_page_list;
      pt_temp -> ptmbpgli = pt_page_list;

      return(1);
    }

}
```

```
int init_safe_list(pt_temp)

                /* create empty saved_files_list for "pt_temp" */
                /* return 1 if ok, 0 if not ok                 */

struct file_inf *pt_temp;

{ struct safe_files_inf *pt_safe; /* pointer to safe file list */

  if ((pt_safe = calloc(1, sizeof(s_safe_files_inf))) == -1)
    { printf("full core memory0);;
      return(0);
    }
  else
    { pt_temp -> ptsffili = pt_safe;
      pt_safe -> ptnxmbsv = pt_safe;
      return(1);
    }

}
```

```
int ask_buffer(pt_temp)

            /* ask buffer of MXSZBF characters for "pt_temp"      */
            /* return 1 if ok, 0 if not ok                        */

struct file_inf *pt_temp;

{ char *pt_buffer; /* pointer to buffer */
  char ccc; /* defined for "sizeof" */

  if ((pt_buffer = calloc(MXSZBF, sizeof(ccc))) == -1)
    { printf("full core memory0);
      return(0);
    }
  else
    { pt_temp -> ptbf = pt_buffer;
      pt_temp -> nmpgbf = 0;
      return(1);
    }

}
```

```
int generate(pt_edited, pt_temp)

                /* generate scratch file "pt_temp"          */
                /* from sequential edited file "pt_edited" */
                /* return 1 if ok, 0 if error               */

struct file_inf *pt_edited;
struct file_inf *pt_temp;

{ extern char *ptlntm; /* buffer line for reading "pt_edited" file */
  int counter; /* count number of read lines from pt_edited */
  int status;
  int step; /* step for allocating numerals to lines */
  int num; /* line identifiers */
  struct line_list *pt_end; /* last member */
  struct line_list *pt_member; /* current member */
  int getl( );
  int adln( );

  counter = 0;
  pt_temp -> pt_c = pt_temp -> pt_b;

  while ((status = getl(pt_edited, ptlntm)) == 1)
    { counter = counter + 1;
      if (counter > MXNMLN)
        { printf("too big file : %s0, pt_edited -> ptfina);
          return(0);
        }
      pt_temp -> ptmbtxli = pt_temp -> pt_c;
      if (adln(pt_temp, 0) == 0)
        return(0);
    }

  if (status != 2)
    { printf("wrong format or can't access %s0,
            pt_edited -> ptfina);
      return(0);
    }

  if (counter > 0)
    { step = ((MXNMLN / counter) - ((MXNMLN / counter) % 1) < pt_temp
      ->nmse) ? (MXNMLN/counter)-((MXNMLN/counter)%1) : pt_temp->nmse;
      pt_temp -> nmse = step;
      num = 0;
      pt_member = pt_temp -> pt_b -> ptnxmbln;
      pt_end = pt_temp -> pt_e;
                /* allocate line identifiers : */
      while (pt_member != pt_end)
        { num = num + step;
          pt_member -> nmln = num;
          pt_member = pt_member -> ptnxmbln;
        }
    }

  return(1);
```

)

```
int getl(pt_inf, pt_char)

                /* get next line from file pt_file  */
                /*                 into pt_char     */
                /*   return 1 if ok                 */
                /*          2 if i/o error          */
                /*          3 if too big line       */
                /*          4 if end of file        */
                /*          5 if wrong format       */

struct file_inf *pt_inf;
char *pt_char;

{ int counter;   /* count characters */
  int fd;   /* file descriptor of pt_inf */
  int n_read;   /* answer of read( ) */
  char ch;   /* each character read */

  fd = pt_inf -> fide;

  n_read = read(fd, &ch, 1);
  if (n_read == 0)
                /* end of file */
    return(4);
  *pt_char = ch;
  counter = 1;

  while (ch != "0)
                /* read more */
    { if (counter == MXLGLN)
                /* too big line */
        return(3);
      if (n_read == 0)
                /* wrong format */
               return(5);
      if (n_read == -1)
                /* i/o error */
        return(2);
      n_read = read(fd, &ch, 1);
      *++pt_char = ch;
      counter = counter + 1;
    }

  return(1);

}
```

```
docm(pt_temp,pt_ed_inf)
     /*  switch command  */
     /* pt_temp = pointer to the struct file_inf of the temporary file */
     /* pt_edited = pointer to the struct file_inf of the edited file */

struct file_inf *pt_temp;
struct file_inf *pt_ed_inf;
{
extern char *ptlntm;
int command; /*defined integer because of the switch */
long line_feed;
char *ptcharcm;
char *pt;
char *skwh( );
char *getin( );
int docm_a( );
int docm_c( );
int docm_d( );
int docm_e( );
int docm_f( );
int docm_h( );
int docm_i( );
int docm_m( );
int docm_n( );
int docm_p( );
int docm_s( );
int docm_v( );

line_feed = '0;

printf("");
pt = ptlntm;

if ( getin(pt) != NULL )  {
        ptcharcm = skwh(ptlntm);
        command = *ptcharcm;

        switch (command){
                case 'a':
                case 'A':
                        docm_a(pt_temp,ptcharcm);
                        break;
                case 'c':
                case 'C':
                        docm_c(pt_temp,ptcharcm);
                        break;
                case 'd':
                case 'D':
                        docm_d(pt_temp,ptcharcm);
                        break;
                case 'e':
                case 'E':
                        docm_e(pt_temp,pt_ed_inf,ptcharcm);
                        break;
                case 'f':
                case 'F':
```

```
                        docm_f(pt_temp,ptcharcm);
                        break;
                case 'h':
                case 'H':
                        docm_h(ptcharcm);
                        break;
                case 'i':
                case 'I':
                        docm_i(pt_temp,ptcharcm);
                        break;
                case 'm':
                case 'M':
                        docm_m(pt_temp,ptcharcm);
                        break;
                case 'n':
                case 'N':
                        docm_n(pt_temp,ptcharcm);
                        break;
                case '0:
                        docm_nx(pt_temp,ptcharcm);
                        break;
                case 'p':
                case 'P':
                        docm_p(pt_temp,ptcharcm);
                        break;
                case 's':
                case 'S':
                        docm_s(pt_temp,ptcharcm);
                        break;
                case 'v':
                case 'V':
                        docm_v(pt_temp,pt_ed_inf,ptcharcm);
                        break;
                default:
                        printf("unrecognised command 0);
                        break;
        }
   }
else printf("i-o error  0");

return;
}
```

```
docm_a(pt_temp,ptcharlncm)
      /* - append command : append a line introduced by the user */
      /* - pt_temp = pointer to the  struct file_inf of the      */
      /*   temporary file                                        */
      /* - the user's command is in ptcharlncm                   */

struct file_inf  *pt_temp;
char *ptcharlncm;

{
int digit;
int *pt_digit;
struct line_list *pt_last;
int insert_lines( );
int check_a_format( );

pt_digit = &digit;

if (check_a_format(ptcharlncm) == 0)
  return;

pt_last = pt_temp->pt_e->ptpvmbln;
*pt_digit = (( MXNMLN - pt_last->nmln) / pt_temp->nmse) -
            ((( MXNMLN - pt_last->nmln) / pt_temp->nmse) % 1);
pt_temp->ptmbtxli = pt_last;
insert_lines(pt_temp,pt_digit);
return;

}
```

```
int check_a_format(ptcharlncm)

        /* - check the format of the append command              */
        /*   return 1 if ok                                      */
        /*          0 if not                                     */

char *ptcharlncm;

{ char *pt_char;
  char *skwh( );

  pt_char = skwh(++ptcharlncm);
  if (*pt_char != '0) {
        printf("format error 0);
        return(0);
  }
  return(1);
}
```

```
char *skwh(pt_char)
          /*  - skip blancs and tab characters from pt_char  */
          /*  - return a pointer to the first character non  */
          /*    equal to blanc or tab                        */

char *pt_char;

{

char c;

while ((c = *pt_char++) == ' ' || c == '\t')
       ;
return(--pt_char);
}
```

```
insert_lines(pt_temp,pt_digit)

            /* - try to insert *pt_digit lines after line  */
            /*   pt_temp->ptmbtxli                         */

struct file_inf *pt_temp;
int *pt_digit;

{

int number_line; /* number of available identifiers */
int new_step; /* local new step */
int new_num_line; /* identifier of new line */
extern char lsmd; /* last modifying command */
extern char *ptlntm; /* terminal line */
char *pt; /* work pointer */
int available( );

  if ((number_line = available(pt_temp))  <  *pt_digit  ||  *pt_digit == 0)
    {   printf("impossible 0);
        return;
    }

  new_step = ((number_line / *pt_digit) - ((number_line / *pt_digit)
          % 1) < pt_temp -> nmse)
          ? (number_line / *pt_digit) - ((number_line / *pt_digit)
              %1)
          : pt_temp -> nmse;
  new_num_line = pt_temp->ptmbtxli->nmln ;

  for ( ; ; )

    {  if ((*pt_digit)-- == 0 )
          return;

       new_num_line =+ new_step;
       printf("%5d",new_num_line);

       pt = ptlntm;
       if (getin(pt) == NULL)
              /* get new line from terminal */
         { printf ("i-o error 0);
           return;
         }

       if (*ptlntm == DL)
         return;

       if (adln(pt_temp,new_num_line) != 1)
                /* add line */
         return;

       pt_temp-> ptmbtxli = pt_temp->pt_c;
       lsmd = 'i';

    }
```

)

```
int available(pt_file_inf)

            /* -  compute the number of possible lines between   */
            /* -  line pointed by pt_file_inf->ptmbtxli          */
            /*    and the following                              */
            /* -  return this number                             */

struct file_inf *pt_file_inf;

{

  return((pt_file_inf->ptmbtxli->ptnxmbln->nmln) - (pt_file_inf->
         ptmbtxli->nmln) - 1);

}
```

```
int adln(pt_temp,num_line)

        /*  - add a line of number num_line after the line  */
        /*    pointed by pt_temp->ptmbtxli                  */
        /*  - return 1 if ok , 0 if not                     */

struct file_inf *pt_temp;
int num_line;

{
int length;
struct line_list *ptnewmemb;
struct line_list *ptwork;
int lnlg( );
int fnpg( );
char *calloc( );

length = lnlg(ptlntm);
if (fnpg(pt_temp, length) == 0) {
        printf("i-o error  0);
        return(0);
}
if (adlnpg(pt_temp,length) == 0 ) {
        printf("i-o error  0);
        return(0);
}
ptnewmemb = calloc(1,sizeof(s_line_list)); /* update texte_list */
if (ptnewmemb ==NULL) {
        printf("i-o error  0");
        return(0);
}
ptwork = pt_temp->ptmbtxli;
ptnewmemb->dplnpg = pt_temp->ptmbpgli->nbbscr;
ptnewmemb->nmln = num_line;
ptnewmemb->ptpvmbln = ptwork;
ptnewmemb->ptnxmbln = ptwork->ptnxmbln;
ptnewmemb->nmpgln = pt_temp->ptmbpgli->nmpgpg;
ptwork->ptnxmbln->ptpvmbln = ptnewmemb;
ptwork->ptnxmbln = ptnewmemb;
pt_temp->ptmbpgli->nbbscr =+ length;  /*  update page_list  */
pt_temp->nblnfi =+ 1;                   /*  update number of lines  */
pt_temp->pt_c = ptnewmemb;                 /*  update current line  */
return(1);
}
```

```
int lnlg(pt_char)

        /*  - compute the length of a string beginning at pt_char  */
        /*  - return this length                                   */

char *pt_char;

{

int length ;

  length = 0;

  while (*pt_char++ != '0)
        {  length =+ 1;
        }

 return(++length);

}
```

```
int fnpg(pt_temp, length)

        /*  - the first page containing at least "length" free  */
        /*    chararcters is searched                           */
        /*  - if no existing creation of a new one , update     */
        /*    of page_list                                      */
        /*  - after : update of pt_temp->ptmbpgli          */
        /*  - return 1 if ok , 0 if not                         */

struct file_inf *pt_temp;
int length;

{

struct page_list *fnpt;
struct page_list *pttopvmb;
int input_output( );
char *calloc( );

  pttopvmb = pt_temp->ptpgli;
  fnpt = pttopvmb->ptnxmbpg;

  while (fnpt != pt_temp->ptpgli)
        {  if ((MXSZBF - (fnpt->nbbscr)) >= length)
                  { pt_temp->ptmbpgli = fnpt;
                    return(1);
                  }
           pttopvmb = fnpt;
           fnpt = fnpt->ptnxmbpg;
        }

  fnpt = calloc(1,sizeof(s_page_list));

  if (fnpt == NULL)
        return(0);

  pttopvmb->ptnxmbpg = fnpt;
  fnpt->nbbscr = 0;
  fnpt->ptnxmbpg = pt_temp->ptpgli;
  fnpt->nmpgpg = (pttopvmb->nmpgpg) + 1;

  if (input_output(pt_temp, fnpt->nmpgpg, '1') == 0)
                return(0);

  pt_temp->ptmbpgli = fnpt;
  return(1);

}
```

```
adlnpg(pt_temp,length)

      /*  - add a line in a page                          */
      /*  - length of the line = "length"                 */
      /*  - page pointed by pt_temp->ptmbpgli    */
      /*  - return 1 if ok , 0 if not                     */

struct file_inf *pt_temp;
int length;

{

extern char *ptlntm;
char *free_char, *pt_line;
int mspgmc( );

  if ( mspgmc(pt_temp) == 0 )
        return(0);

 free_char = pt_temp->ptbf + pt_temp->ptmbpgli->nbbscr;
  pt_line = ptlntm;

  while (length != 0 )
        { *(free_char++) = *(pt_line++);
          length =- 1;
        }

  return(1);

}
```

```
int mspgmc(pt_temp)

          /*  - bring a page in central memory          */
          /*  - page pointed by pt_temp->ptmbpgli   */
          /*  - return 1 if ok , 0 if not               */

struct file_inf *pt_temp;

{

int input_output( );

  if (pt_temp->ptmbpgli->nmpgpg == pt_temp->nmpgbf)
        return(1);

  if (input_output(pt_temp, pt_temp->nmpgbf, '1' ) == 0)
        return(0);

  if (input_output(pt_temp, pt_temp->ptmbpgli->nmpgpg, '0' ) == 0)
        return(0);

  return(1);

}
```

```
int input_output(pt_temp, num_page, rwmode)

                /* - read (rwmode = 0) or write (rwmode = 1) page   */
                /* - "num_page" of file "pt_file_inf"               */
                /* - buffer pointed by pt_file_inf->ptbf            */
                /* - return 1 if ok, 0 if not ok                    */

struct file_inf *pt_temp;
int num_page;
int rwmode;

{

long offset;
char *pt_char;

  offset = num_page * MXSZBF;

  switch(rwmode)

    { case '0' :

        seek(pt_temp->fide,offset,0);

        if (read (pt_temp->fide,pt_char,MXSZBF) != MXSZBF)
            { printf(" i-o error 0);
              return(0);
            }

        pt_temp->nmpgbf = num_page;
        break;

       case '1' :

        seek(pt_temp->fide,offset,0);

        if ( write (pt_temp,pt_char,MXSZBF) != MXSZBF)
            { printf (" i-o error 0);
              return(0);
            }

        pt_temp->nmpgbf = num_page;
        break;

    }

  return(1);

 }
```

```
docm_c(pt_temp,ptcharlncm)

        /*  -  copy command : copy the lines identified by the      */
        /*       user's command                                     */

struct file_inf *pt_temp;
char *ptcharlncm;

{

extern char lsmd; /* last modidying command */
extern char *ptlntm;
int lower; /* pointer to first identified line */
int upper; /* pointer to last identified line */
int idln; /* pointer to insertion numeral line */
int possible; /* number of available numerals */
int copied;                /* number of lines to be copied */
int new_num_line; /* new numeral of line */
int new_step; /* local step */
int *pt_lower;
int *pt_upper;
int *pt_idln;
struct line_list *pt_copy;
struct line_list *pt_add_where;
int check_c_format( );
int ckexstln( );
int ms_copy_line( );

  pt_lower = &lower;
  pt_upper = &upper;
  pt_idln = &idln;

  if(check_c_format (ptcharlncm,pt_temp,pt_lower,pt_upper,pt_idln) == 0)
        return;

  if ( (copied = ckexstln(pt_temp,pt_lower,pt_upper)) == 0 )
    {   printf("non existent line 0);
        return;
    }

  if (*pt_lower < MNNMLN || *pt_upper > MXNMLN )
    {   printf ("impossible number 0);
        return;
    }

  pt_copy = pt_temp->ptmbtxli;

  if (ckexstln (pt_temp,pt_idln,pt_idln) == 0)
    {   printf("non existent line 0);
        return;
    }

  if (*pt_idln > MXNMLN )
    {   printf("impossible number 0);
        return;
    }
```

```
  if (copied > (possible = available(pt_temp)))
    {   printf("too many lines to be copied 0);
        return;
    }

  new_step = ((possible / copied) - ((possible / copied) % 1 ) < pt_temp->nmse)
         ? (possible / copied) - ((possible / copied) %1)
         : pt_temp->nmse;

  pt_add_where = pt_temp->ptmbtxli;
  new_num_line = pt_temp->ptmbtxli->nmln;

  while ( --copied >= 0 )
    {   pt_temp->ptmbtxli = pt_copy;
        ms_copy_line(pt_temp,ptlntm);
        pt_temp->ptmbtxli = pt_add_where;
        new_num_line =+ new_step;

        if (adln(pt_temp,new_num_line) != 1)
                return;

        pt_add_where = pt_temp->pt_c;
        pt_copy = pt_copy->ptnxmbln;
    }

  lsmd = 'c';

  return;

}
```

```
int check_c_format(ptcharlncm,pt_temp,pt_lower,pt_upper,pt_idln)

        /*  -  check the format of the copy command                 */
        /*  -  return 1 if ok                                       */
        /*             0 if not                                     */



char *ptcharlncm;
struct file_inf *pt_temp;
int *pt_lower; /* pointer to numeral first identified line */
int *pt_upper; /* pointer to numeral last identified line */
int *pt_idln; /* pointer numeral insertion line */

{

char *pt_char;
char *ckidstln( );
char *ckidln( );

  pt_char = skwh(++ptcharlncm);
  pt_char = ckidstln(pt_temp,pt_char,pt_lower,pt_upper);

  if (pt_char == NULL)
        return(0);

  pt_char = skwh(pt_char);

  if (*pt_char != DL)
    {   printf("format error 0);
        return(0);
    }

  pt_char = skwh(++pt_char);
  pt_char = ckidln(pt_temp,pt_char,pt_idln);

  if (pt_char == NULL)
        return(0);

  pt_char = skwh(pt_char);

  if(*pt_char != '0)
    {   printf("format error 0);
        return(0);
    }

return(1);

}
```

```
char *ckidstln(pt_file_inf,pt_char,pt_low,pt_up)

              /*  - check the set of lines identifier          */
              /*  - return a pointer to the next character of  */
              /*    the identifier if ok , null if not         */

struct file_inf *pt_file_inf;
char *pt_char; /* pointer to  set of lines identifier */
int *pt_low; /* pointer to first line numeral identified */
int *pt_up; /* pointer to last line numeral identified */

{


  pt_char = ckidln(pt_file_inf,pt_char,pt_low);

  if (pt_char == NULL)
        return(NULL);

  pt_char = skwh(pt_char);

  if (*pt_char == ':')
    {  pt_char = skwh(++pt_char);
       pt_char = ckidln(pt_file_inf,pt_char,pt_up);

       if (pt_char == NULL)
               return(NULL);
    }

else

    {   *pt_up = *pt_low;
        return(pt_char);
    }

}
```

```
char *ckidln(pt_file_inf,pt_char,pt_bound)

            /*  - check the identifier of a line                    */
            /*  - return a pointer to the character just after      */
            /*    the identifier if ok , null if not                */

struct file_inf *pt_file_inf;
char *pt_char; /* pointer through the line */
int *pt_bound; /* numeral of identified line */

{

char c;
char *cvdg( ),*cvsp( );

  if (isdigit ( *pt_char) == 1)
    {   pt_char = cvdg(pt_char,pt_bound);
        return(pt_char);
    }

  else if (isspecial(*pt_char) != 0)

            {   pt_char = cvsp(pt_file_inf,pt_char,pt_bound);
                return(pt_char);
            }

        else
            {   printf ("impossible number 0);
                return(NULL);
            }

   }
```

```
int isdigit(c)

        /*  -  return 1 if c is a digit                          */
        /*             0 if not                                  */

char c;

{
  return ( ( c =='0' || c == '1' || c == '2' || c == '3' || c == '4'
             || c == '5' || c == '6' || c == '7' || c == '8'
             || c == '9' ) ? 1 : 0);

}
```

```
int isspecial(c)
        /*  -  check if the caracter c is a special identifier  */
        /*     b , c , f ou l                                   */
        /*  -  return 1 if c ia a special identifier            */
        /*            0 if not                                  */

char c; /* input character */

{

  return( (c == 'B' || c == 'b' || c == 'C' || c == 'c' || c == 'F'
          || c == 'f' || c == 'L' || c == 'l') ? 1 : 0);

}
```

```
char *cvdg(pt_char,pt_int)

          /*  - convert a chain.    of digits pointed by pt_char to */
          /*    an integer                                          */
          /*  - the integer will be in pt_int                       */
          /*  - return a pointer to the next character after        */
          /*    the chain if ok , null if not                       */

char *pt_char; /* numeric character */
int *pt_int; /* the numeral */

{

int count ; /* digit counter */
char c; /* numeric character */
long test ; /* test variable */
char *pt_saved; /* pointer to the first digit */
int power( );

count = 0;
test = 0;

  pt_saved = pt_char;
   while ( isdigit( c = *pt_char) == 1)
    {   ++pt_char;
        ++count;
    }

  if (count > 5)

    {   printf("impossible number 0);
        return(NULL);
    }

  while (--count >= 0)
    {  test =+ (*pt_saved++ - '0' ) * power(10,count);

        if (test > MXNMLN)
            {   printf("number too big 0);
                return(NULL);
            }
    }

  *pt_int = test;
  return(pt_char);

}
```

```
int power(x,n)

        /* raise x to n-th power , x not equal 0 */

int x,n;

{

int i,p;

  p = 1;

  for(i=1;i<=n;++i)
        p = p * x;

return(p);

}
```

```
char *cvsp(pt_file_inf,pt_char,pt_int)

          /*  - convert a special identifier (b,c,f or l) to  */
          /*    a number of line                              */
          /*  - return a pointer to the next character        */

struct file_inf *pt_file_inf;
char *pt_char; /* special identifier */
int *pt_int; /* numeral identified line */

{

char c;

                /* begin line */

  if ((c = *pt_char) == 'b' || c =='B' )
    {   *pt_int = pt_file_inf->pt_b->nmln;
        return(++pt_char);
    }

                /* current line */

  else if (c == 'c' || c == 'C' )
    {   *pt_int = pt_file_inf->pt_c->nmln;
       return(++pt_char);
    }

                /* first line */

     else if (c == 'f' || c == 'F' )
            {   *pt_int = pt_file_inf->pt_b->ptnxmbln->nmln;
                return(++pt_char);
            }

          else         /*   last line   */
                  {   *pt_int =pt_file_inf->pt_e->ptpvmbln->nmln;
                       return(++pt_char);
                  }

}
```

```
int ckexstln(pt_file_inf,ptlower,ptupper)

          /*  - check the existence of a set of lines  */
          /*  - return the number of lines between the */
          /*    lower and the upper if these two lines */
          /*    exist and pt_file_inf->ptmbtxli point  */
          /*    to the line identified by the lower ,  */
          /*    0 if not                               */

struct file_inf *pt_file_inf;
int *ptlower; /* first identified line numeral */
int *ptupper; /* last identified line numeral */

{

int num; /* numeral last identified line */
int compteur ; /* number of identified lines */
struct line_list *pt_list;

  compteur = 1;
  pt_list =pt_file_inf->pt_e->ptpvmbln;

  while ((num = pt_list->nmln) != *ptupper)
    {   if(num > *ptupper)
                pt_list = pt_list->ptpvmbln;
        else return(0);
    }
while ((num = pt_list->nmln) != *ptlower)

    {   if (num > *ptlower)

            {   pt_list = pt_list->ptpvmbln;
                compteur =+ 1;
            }

        else return(0);

    }

  pt_file_inf->ptmbtxli = pt_list;

  return(compteur);

}
```

```
int ms_copy_line (pt_temp,pt_to)

        /* copy the line identified by pt_temp->ptmbtxli to      */
        /* pt_to                                                 */

struct file_inf *pt_temp;
char *pt_to;

{

int fnptpgln( );
int copy_line( );

                        /* bring page in central memory */

  fnptpgln(pt_temp);

  if (mspgmc(pt_temp) == 0)
        return;

  copy_line(pt_temp->ptbf + pt_temp->ptmbtxli->dplnpg , pt_to);

  return;

}
```

```
fnptpgln(pt_file_inf)

        /* - find the member of the page_list corresponding to the */
        /*   line pointed by pt_file_inf->ptmbtxli                 */
        /* - after : pt_file_inf->ptmbpgli points to this page     */

struct file_inf *pt_file_inf;

{

int research; /* numeral of searched page */
struct page_list *pt_work;

  pt_work = pt_file_inf->ptpgli;
  research = pt_file_inf->ptmbtxli->nmpgln;

                        /* search through the page list */

  while (pt_work->nmpgpg != research)
        pt_work = pt_work->ptnxmbpg;

  pt_file_inf->ptmbpgli = pt_work;

  return;

}
```

```
int copy_line(pt_from,pt_to)

        /*  -  copy the string beginning at pt_from to an area     */
        /*     beginning at pt_to                                  */

char *pt_from;
char *pt_to;

{

  for( ; ; )

     {  *pt_to = *pt_from;

        if (*pt_to == '0)
                return;

        ++pt_to;
        ++pt_from;

     }

}
```

```
docm_d(pt_temp,ptcharlncm)

      /* - delete command : delete the line specified by  */
      /*   the user's command                             */

struct file_inf *pt_temp;
char *ptcharlncm; /* user's command */

{

extern char lsmd; /* last modifying command */
int lower; /* first identified numeral line */
int upper; /* last identified numeral line */
int counter; /* number of identified lines */
int *pt_lower,*pt_upper;
struct line_list *pt_member;
int dlln( );
int check_d_format( );

  pt_lower = &lower;
  pt_upper = &upper;

  if (check_d_format(ptcharlncm, pt_temp, pt_lower, pt_upper) == 0)
    return;

  if (*pt_upper > MXNMLN   ||  *pt_lower < MNNMLN)
    {   printf("impossible number 0);
        return;
    }

  if ((counter = ckexstln(pt_temp,pt_lower,pt_upper)) == 0)
    {   printf("unexisting line(s) 0);
        return;
    }
  pt_member = pt_temp->ptmbtxli;

                        /* delete line(s) */

  while ( --counter >= 0 )
    {   if (dlln(pt_temp) != 1)
                return;

        pt_temp->ptmbtxli = pt_temp->pt_c;
        lsmd = 'd';          /*  update lsmd  */
    }

  return;

}
```

```
int check_d_format(ptcharlncm, pt_temp, pt_lower, pt_upper)
        /*  -  check the format of the delete command          */
        /*     return 1 if ok                                  */
        /*            0 if not                                 */

char *ptcharlncm; /* user's command */
struct file_inf *pt_temp;
int *pt_lower; /* first line identified numeral */
int *pt_upper; /* last line identified numeral */

{

  char *pt_char; /* pointer trough user's command */

  pt_char = skwh(++ptcharlncm);
  pt_char = ckidstln(pt_temp,pt_char,pt_lower,pt_upper);

  if (pt_char == NULL)
        return(0);
  pt_char = skwh(pt_char);

  if (*pt_char != '0)
    {   printf("format error 0);
        return(0);
    }

  return(1);

  }
```

```
int dlln(pt_temp)

        /*  - delete a line pointed by pt_temp->ptmbtxli  */
        /*  - free the place of the line in the page      */
        /*    and in  the line list.                      */
        /*  - return 1 if ok , 0 if not                       */

struct file_inf *pt_temp;

{

int pageincm; /* numeral page in central memory */
int length; /* deleted line length */
char *pt_char; /* first deleted character */
struct line_list *pt_member;
int dllnpg( );
int cfree( );
int update_line_list( );

                        /* bring page in central memory */

  fnptpgln(pt_temp);

  if (mspgmc(pt_temp) != 1)
    {   printf("i-o error  0);
        return(0);
    }

  pt_char = pt_temp->ptbf + pt_temp->ptmbtxli->dplnpg;

  length = lnlg(pt_char);

                                /* delete and update */

  shift (pt_char, (pt_temp->ptbf+pt_temp->ptmbpgli->nbbscr)-pt_char,length);
  update_line_list(pt_temp,length);          /* update line_list */

  join_in_line_list(pt_temp->ptmbtxli->ptpvmbln,pt_temp->ptmbtxli->ptnxmbln);

  pt_temp->ptmbpgli->nbbscr =- length;  /* update page_list  */

  pt_temp->pt_c = pt_temp->ptmbtxli->ptnxmbln; /* update current line */

  pt_temp->nblnfi =- 1; /*  update number of lines in the file */

   cfree (pt_temp->ptmbtxli,1,sizeof(pt_temp->ptmbtxli));
   /* free place ptmbtxli of the deleted line */

  return(1);

}
```

```
update_line_list(pt_file_inf,length)

                /* - update the displacement of the lines in    */
                /*   the page after a shift of length positions*/

struct file_inf *pt_file_inf;
int length;

{

int pageinmc; /* numeral page in central memory */
struct line_list *pt_member;

  pageinmc = pt_file_inf->nmpgbf;
  pt_member = pt_file_inf->pt_b->ptnxmbln;

  while (pt_member != pt_file_inf->pt_e)
    {   if (pt_member->nmpgln == pageinmc && pt_member->dplnpg > pt_file_inf->ptmbtxli
                pt_member->dplnpg =- length;
        pt_member = pt_member->ptnxmbln;
    }

  return;

}
```

```
docm_e(pt_temp, pt_edited, ptcharlncm)

                /* execute "e" command pointed by ptcharlncm      */
                /* with temporary file "pt_temp" and edited file  */
                /* "pt_edited                                     */

struct file_inf *pt_temp;
struct file_inf *pt_edited;
char *ptcharlncm;

{ extern char lsmd;
  extern int lssffide;
  extern int tsen;
  int get_yes_no( );
  int check_e_format( );

  if (check_e_format(ptcharlncm) == 0)
    return;

  if (lsmd != 'v')  /* not saved text */
    { printf("not saved text! do you want to quit ?0);

      if (get_yes_no( ) == 1)  /* answer is 'y' */
        { tsen = 1;
          unlink(pt_temp -> ptfina);
          return;
        }
       else                    /* answer is 'n' */
          return;
     }

  if (lssffide != pt_edited -> fide)
                /* file not saved in "pt_edited" */
     { printf ("file not saved in %s 0,pt_edited->ptfina);
       printf (" do you want to quit ?0);

       if (get_yes_no( ) == 1) /* answer is 'y' */
         { tsen = 1;
           unlink(pt_temp -> ptfina);
           return;
         }
       else                    /* answer is 'n' */
         return;
      }

    /* file saved in "pt_edited" : */

    printf("file saved in %s0,pt_edited->ptfina);
    unlink(pt_temp -> ptfina);
    tsen = 1;

    return;

  }
```

```
int check_e_format(ptcharlncm)

                /* check format of "e" command pointed by ptcharlncm */
                /* return 1 if ok, 0 if format error                 */

char *ptcharlncm;

{

  if (*skwh(++ptcharlncm) != '0)
    { printf ("format error0);
      return(0);
    }
  else
    return(1);

}
```

```
int get_yes_no( )

                /* get line from user          */
                /* return 0 if 'n', 1 if 'y'  */

{ int status; /* kind of answer */
  int in_and_check( );

  while ((status = in_and_check( )) == 2)  /* user error */
       ;

  return(status);

}
```

```
int in_and_check( )

                /* get line from user                          */
                /* return 0 if 'n', 1 if 'y', 2 if 'error'     */

{ int status; /* kind of character */
  int ch; /* first character of line */
  char line[MXLGLN +1]; /* line from user */
  char *pt_line; /* pointer to line */
  char *pt_through; /* pointer through line */

  pt_line = line;
  pt_through = pt_line;

  do
    { if (read(0,pt_through,1) == -1)
        return(2);
    }
  while (*pt_through++ != '0);

  pt_line = skwh(pt_line);

  ch = *pt_line;

  switch(ch)
    { case 'y':
      case 'Y':
        status = 1;
        break;
      case 'n':
      case 'N':
        status = 0;
        break;
      default:
        printf("answer again0);
        return(2);
        break;
    }

  pt_line = skwh(++pt_line);

  if (*pt_line != '0)
    { printf ("answer again0);
      return(2);
    }

  return(status);

}
```

```
char *getin(ptwhere)

        /* - put a line terminated by a carriage return from the */
        /*   standart input to an array pointed by ptwhere       */


char *ptwhere;

{

int nbcar;
char c;
nbcar = 0;

  do
    {
        c = getc (ptwhere++) ;
        nbcar =+ 1;

        if (c == -1)
          { printf ("i-o error 0);
            return(NULL);
          }

  }

  while ( c != "0 || nbcar > MXLGLN - 1);


 return(--ptwhere);

}
```

```
docm_f(pt_temp,ptcharlncm)

      /* - find command                                          */
      /* - print the line(s) where the find succeeds             */

struct file_inf *pt_temp;
char *ptcharlncm; /* pointer to user's command */

{

int length; /* searched string length */
int lower; /* first line identifier */
int upper; /* last line identifier */
int counter; /* count number of identified lines */
int fail; /* success status */
int *pt_length,*pt_lower,*pt_upper;
char string[MXLGLN]; /* searched string */
char *pt_char; /* pointer through line */
char *pt_find; /* pointer to string in line */
char *pt_string; /* pointer to searched string */
struct line_list *pt_member;
int prln( );
int check_f_format( );
char *fnsr( );

  pt_length = &length;
  pt_lower = &lower;
  pt_upper = &upper;
  pt_string = string;
  fail = 1;

  if (check_f_format(ptcharlncm, pt_temp, pt_string, pt_lower,
                  pt_upper, pt_length) == 0)
    return;

  if ((counter = ckexstln(pt_temp,pt_lower,pt_upper)) == 0)
    {   printf("unexisting line(s) 0);
        return;
    }

  if (*pt_lower < MNNMLN  ||  *pt_upper > MXNMLN)
    {   printf("impossible number of line 0);
        return;
    }

  printf("0);
  pt_member = pt_temp->ptmbtxli;

  while (--counter >= 0 )
    {   fnptpgln(pt_temp);

        if (mspgmc(pt_temp) == 0)
            {   printf("i-o error  0);
                return;
            }
```

```
        pt_char = pt_member->dplnpg + pt_temp->ptbf;
        pt_find = fnsr(pt_char,pt_string,pt_length);

        if (pt_find != NULL)
            {   pt_temp->pt_c = pt_member;/* update current line */

                if (prln(pt_temp) == 0)
                        return;
                fail = 0;
            }

        pt_member = pt_member->ptnxmbln;
        pt_temp->ptmbtxli = pt_member;

    }

  if (fail == 1)
    {   printf("search failed 0);
        return;
    }

  return;

}
```

```
int check_f_format(ptcharlncm, pt_temp, pt_string, pt_lower,
                   pt_upper, pt_length)

        /*  -  check the format of the find command            */
        /*  -  return 1 if ok                                  */
        /*             0 if not                                */

char *ptcharlncm; /* user's command */
struct file_inf *pt_temp;
int *pt_lower; /* first identified line numeral */
int *pt_upper; /* last identified line numeral */
int *pt_length; /* pointer to searched string length */

{

  char *pt_char; /* pointer through user's command */
  char *svsr( );

  pt_char = skwh(++ptcharlncm);

  if (*pt_char != DL )
    {   printf("format error 0);
        return(0);
    }

  pt_char = svsr(++pt_char,pt_string,pt_length);
                       /* save searched string */

  if (pt_char == NULL)
    {   printf("format error 0);
        return(0);
    }

                       /* empty string */
  if (*pt_length == 0)
    {   printf("impossible search 0);
        return(0);
    }

  pt_char = skwh(pt_char);
  pt_char = ckidstln(pt_temp,pt_char,pt_lower,pt_upper);

  if (pt_char == NULL)
        return(0);

  pt_char = skwh(pt_char);

  if (*pt_char !='\n')
    {   printf("format error 0);
        return(0);
    }

  return(1);

}
```

```
char *svsr(pt_char,pt_string,pt_length)

        /*  -  save a string beginning at pt_char and terminated */
        /*     by sep to an area beginning at pt_string          */
        /*  -  the length of the string shuold be in pt_length   */
        /*  -  return a pointer to the character after the sep   */
        /*     if it exist                                       */
        /*           null if not                                 */

char *pt_char,*pt_string;
int *pt_length;

{

  *pt_length = 0;

  while ( *pt_char != DL)
    {   if (*pt_char == '0)
                return(NULL);

        *pt_length =+ 1;
        *pt_string++ = *pt_char++;
    }

  return(++pt_char);

}
```

```
char  *fnsr(pt_char,pt_string,pt_length)

          /* - search a string in a line beginning at pt_char */
          /* - the searched string is in pt_string            */
          /* - the length of this string is in pt_length      */
          /* - return a pointer to the first position of the  */
          /*   string in the line , null if the string is not */
          /*   in the line                                    */

int *pt_length;
char *pt_char,*pt_string;

{

int length; /* work variable */
char *pttostring; /* pointer to searched string */
char *pt_work; /* pointer through line */

  while ( *pt_char != '\n' )

    {   pt_work = pt_char;
        pttostring = pt_string;
        length = *pt_length;

        while(--length >=0 && *pt_work++ == *pttostring++)

            {   if (length == 0)
                        return(pt_char);
            }

   ++pt_char;

    }

  return(NULL);

}
```

```
int prln(pt_file_inf)

        /* - print a line from the central memory            */
        /* - the line is pointed by pt_file_inf->ptmbtxli    */

struct file_inf *pt_file_inf;

{

char c;
char *pt_char; /* pointer first character of line */

  pt_char = pt_file_inf->ptmbtxli->dplnpg + pt_file_inf->ptbf;
  printf("%5d  ",pt_file_inf->ptmbtxli->nmln);
  c = *pt_char;

  do

    {   if (putc((c = *pt_char++),1) == EOF)
            {   printf("i-o error  0);
                return(0);
            }

    } while (c  != '0);

  return(1);

}
```

```
int docm_h(ptcharlncm)

          /* execute "h" command pointed by ptcharlncm          */

char *ptcharlncm;

{
int check_h_format( );
char *help_name[9]; /* name of  "help file "  */
char *pt_help_name; /* name to "help file" */
int fd; /* help file descriptor */
int end_file_status; /* status for end file */
char c; /* each read char from "help file" */

pt_help_name = help_name;

if ( check_h_format(ptcharlncm,pt_help_name) == 0) /* wrong format */
   return;

fd = open(pt_help_name,1);

for ( ; ; )
  { end_file_status = read (fd,&c,1);
    if ( end_file_status == 0 )
                  /* EOF */
        break;

    write (fd,&c,1);
   }

return;

}
```

```
int check_h_format(ptcharlncm,pt_help_name)

        /*  - check the format of command "h" pointed by        */
        /*     ptcharlncm                                       */
        /*  - return 1 if ok and put name of "help file"        */
        /*    file required                                     */
        /*  -  return 0 if wrong format                         */


char *ptcharlncm;
char *pt_help_name;


{

char * pt_through; /* through "h" command */
char ch; /* help required */
int strcpy( );

pt_through = skwh(ptcharlncm);

if ( *pt_through == '0 )
              /* all information required */
  {
   strcpy ( pt_help_name,"help_all");
   return(1);
  }

if (*skwh(++pt_through) !='\n')
  { printf("wrong format0);
    return(0);
  }

if((ch = *pt_through) != '  ' && (ch != '    && (ch != ') && (ch != 'h') && (ch != 'i'
   (ch != '0) && (ch != '') && (ch != ') && (ch != 'v'))
   { printf ("doesn't exist 0);
     return(0);
   }
 *pt_help_required = 'h';
*++pt_help_required = 'e';
*++pt_help_required = 'l';
*++pt_help_required = 'p';
*++pt_help_required = '_';
*++pt_help_required = *pt_through;
*++pt_help_required = '0';

return(1);

}
```

```
int strcpy(s,t)

          /* copy s to t */

char *s,*t;

{
  while (( *s = *t ) != '0' )
    {  s++;
       t++;
    }

}
```

```
docm_i(pt_temp,ptcharlncm)

      /* - insert command for the insertion of line(s)     */
      /* - the line is pointed by pt_temp -> ptmbtxli      */

struct file_inf *pt_temp;
char *ptcharlncm; /* user's command */

{

int idln,digit;
int *pt_idln; /* insertion line numeral */
int *pt_digit; /* number of insertion lines */
int check_i_format( );

  pt_idln = &idln;
  pt_digit = &digit;

  if (check_i_format(ptcharlncm, pt_temp, pt_idln, pt_digit)  == 0)
  return;

  if (ckexstln(pt_temp,pt_idln,pt_idln) == 0)
    {   printf(" unexisting line 0);
        return;
    }

  if ( *pt_idln > MXNMLN )
    { printf ("impossible number of line 0);
      return;
    }

                                /* insert lines */

  insert_lines(pt_temp,pt_digit);

  return;

}
```

```
int check_i_format(ptcharlncm, pt_temp, pt_idln, pt_digit)


        /*  -  check the format of the insert command          */
        /*  -  return 1 if ok                                  */
        /*             0 if not                                */

char *ptcharlncm; /* pointer to user's command */
struct file_inf *pt_temp;
int *pt_idln; /* numeral of insertion line */
int *pt_digit; /* number of inserted lines */

{

char * pt_char; /* work pointer through command */


  pt_char = skwh(++ptcharlncm);
  pt_char = cvdg(pt_char,pt_digit);

  if (pt_char == NULL)
    return(0);

  pt_char = skwh(pt_char);

  if (pt_char != DL)
    { printf ("format error 0);
      return(0);
    }

  pt_char = skwh(pt_char);
  pt_char = ckidln(pt_temp,pt_char,pt_idln);

  if (pt_char == NULL)
    return(0);

  pt_char = skwh (pt_char);

  if (*pt_char !='\n')
    {   printf("format error 0);
        return(0);
    }

  return(1);

}
```

```
docm_m(pt_temp,ptcharlncm)

        /*  move command : move the lines specified by the      */
        /*  user's command                                      */

struct file_inf *pt_temp;
char *ptcharlncm; /* pointer to user's command */

{

extern char lsmd; /* last modifying command */
int moved; /* number of lines to be moved */
int possible; /* number of available identifiers */
int previous_num; /* insertion line identifier */
int new_step; /* local step */
int lower; /* first identified line numeral */
int upper; /* last identified line numeral */
int idln; /* insertion line numeral */
int *pt_lower,*pt_upper,*pt_idln;
struct line_list *pt_through; /* through line list */
struct line_list *pt_moved; /* first moved line */
struct line_list *pt_before_moved; /* before first moved line */
struct line_list *pt_after_insertion; /* after insertion */
struct line_list *pt; /* work pointer */
int check_m_format( );
int join_in_line_list( );

  pt_lower = &lower;
  pt_upper = &upper;
  pt_idln = &idln;

  if (check_m_format(ptcharlncm,pt_temp,pt_lower,pt_upper,pt_idln) == 0)
        return;

  if ((moved = ckexstln(pt_temp,pt_lower,pt_upper)) == 0)
    {   printf("unexistent line(s) 0);
        return;
    }

  pt_moved = pt_temp->ptmbtxli;

  if  (*pt_lower < MNNMLN || *pt_upper > MXNMLN )
    {   printf ("impossible number 0);
        return;
    }

  if(ckexstln(pt_temp,pt_idln,pt_idln) == 0)
    {   printf("unexistent line 0);
        return;
    }

  if ( *pt_idln > MXNMLN )
    {   printf ("impossible number\n");
        return;
    }
```

```
  if ( moved > (possible = available(pt_temp)))
    {   printf("too many lines to be moved 0);
        return;
    }

  new_step = ((possible / moved) - ((possible / moved) %1) < pt_temp->nmse)
           ? (possible / moved) - ((possible / moved) %1)
           : pt_temp->nmse;
  pt_after_insertion = pt_temp->ptmbtxli->ptnxmbln;
  pt_before_moved = pt_moved->ptpvmbln;
  pt_through = pt_moved;
  previous_num = pt_temp->ptmbtxli->nmln ;

  while (--moved >= 0)
    { pt_through->nmln = previous_num + new_step;

      previous_num = pt_through->nmln;
      pt_through = pt_through->ptnxmbln;
    }

  pt = pt_through -> ptpvmbln;

  join_in_line_list(pt_before_moved, pt_through);
  join_in_line_list(pt_temp->ptmbtxli,pt_moved);
  join_in_line_list(pt, pt_after_insertion);

  pt_temp->pt_c = pt;
  lsmd = 'm';

}
```

```
int check_m_format(ptcharlncm,pt_temp,pt_lower,pt_upper,pt_idln)


        /*  -  check the format of the move command                */
        /*  -  return 1 if ok                                      */
        /*             0 if not                                    */

char *ptcharlncm; /* pointer to user's command */
struct file_inf *pt_temp;
int *pt_lower; /* first specified line */
int *pt_upper; /* second specified line */
int *pt_idln; /* insertion line */

{

char *pt_char;

  pt_char = skwh(++ptcharlncm);
  pt_char = ckidstln(pt_temp,pt_char,pt_lower,pt_upper);

  if (pt_char == NULL)
        return(0);

  pt_char = skwh(pt_char);

  if (*pt_char != DL)
    {   printf("format error 0);
        return(0);
    }

  pt_char = skwh(++pt_char);
  pt_char = ckidln(pt_temp,pt_char,pt_idln);

  if (pt_char == NULL)
        return(0);

  pt_char = skwh(pt_char);

  if(*pt_char != '0)
    {   printf("format error 0);
        return(0);
    }

  return(1);

}
```

```
docm_n(pt_file_inf,ptcharlncm)

       /* - number command                                              */
       /* - change the step (default = 10) and/or renumber all          */
       /*   the lines of the text                                       */

struct file_inf *pt_file_inf;
char *ptcharlncm;

{

extern char lsmd; /* last modifying command */
int count; /* count new step */
int digit; /* new step */
int *ptdigit;
struct page_list *pt_list; /* pointer to last member */
struct page_list *pt_work; /* pointer through list */
int check_n_format( );

  ptdigit = &digit;

  if (check_n_format(ptcharlncm, ptdigit) == 0)
    return;

  if ((MXNMLN / *ptdigit) < pt_file_inf->nblnfi)
    {   printf("impossible,choose a smaller step please 0);
        return;
    }

  pt_file_inf->nmse = *ptdigit;
  count = 1;
  pt_list = pt_file_inf->pt_e;
  pt_work = pt_file_inf->pt_b;

  while (pt_work->ptnxmbln != pt_list)
    {   pt_work = pt_work->ptnxmbln;
        pt_work->nmln = count * *ptdigit;
        count =+1;
    }

  lsmd = 'n';
  return;

}
```

```
int check_n_format(ptcharlncm, ptdigit)

        /*  -  check the format of the number command          */
        /*  -  return 1 if ok                                  */
        /*             0 if not                                */

char *ptcharlncm; /* pointer to user's command */
int *ptdigit; /* pointer to step */

{

  char *pt_char; /* pointer through user's command */

  pt_char = skwh(++ptcharlncm);
  pt_char = cvdg(pt_char,ptdigit);

  if (pt_char == NULL)
        return(0);

  if (*ptdigit == 0)
    {   printf("impossible step 0);
        return(0);
    }

  pt_char = skwh(pt_char);

  if (*pt_char != '0)
    {   printf("format error 0);
        return(0);
    }

  return(1);

}
```

```
docm_nx(pt_temp,ptcharcm)

      /* print the line after the current line to the standard    */
      /* output.                                                  */


struct file_inf *pt_temp;
char *ptcharcm;

{

pt_temp->ptmbtxli = pt_temp->pt_c->ptnxmbln;

if (pt_temp->ptmbtxli = pt_temp->pt_e)
    { printf ("end of file");
      return;
    }

if (prlnms(pt_temp) == 0)
      return;

pt_temp->pt_c = pt_temp->ptmbtxli;

return;

}
```

```
docm_p(pt_temp, ptcharlncm)

        /*  -  print command : print the set of lines identified */
        /*     by the user's command                             */

struct file_inf *pt_temp;
char *ptcharlncm;

{

int lower;
int upper;
int counter; /* number of identified lines */
int *pt_lower; /* numeral first identified line */
int *pt_upper; /* numeral last identified line */
struct line_list *pt_member;
int prlnms( );
int check_p_format( );

  pt_lower = &lower;
  pt_upper = &upper;

  if (check_p_format(ptcharlncm, pt_temp, pt_lower, pt_upper) == 0)
        return;

  if ((counter = ckexstln(pt_temp,pt_lower,pt_upper)) == 0)
    {   printf("unexisting line(s) 0);
        return;
    }

  if (*pt_lower < MNNMLN  ||  *pt_upper > MXNMLN)
    {   printf ("impossible number of line 0);
        return;
    }

  pt_member = pt_temp->ptmbtxli;

  while (--counter >= 0)

                                    /* print set of lines */

     {  if (prlnms(pt_temp) == 0)
                return;

        pt_temp->pt_c = pt_member; /*  update current line  */
        pt_member = pt_member->ptnxmbln;
        pt_temp->ptmbtxli = pt_member;

     }

return;

}
```

```
int check_p_format(ptcharlncm, pt_temp, pt_lower, pt_upper)

        /*  -  check the format of the print command          */
        /*  -  return 1 if ok                                 */
        /*             0 if not                               */

char *ptcharlncm; /* pointer to the user's command */
struct file_inf *pt_temp;
int *pt_lower; /* numeral of the first identified line */
int *pt_upper; /* numeral of the last identified line */


{

char *pt_char;

  pt_char = skwh(++ptcharlncm);
  pt_char = ckidstln(pt_temp, pt_char, pt_lower, pt_upper);

  if (pt_char == NULL)
    return(0);

  pt_char = skwh(pt_char);

  if (*pt_char != '0)
    { printf("format error0);
      return(0);
    }

  return(1);

}
```

```
int prlnms(pt_temp)

        /* - print the line pointed by pt_temp->ptmbtxli      */
        /* - the page containing the line is not necessary in */
        /*   central memory                                   */
        /* - return 1 if ok , 0 if not                        */

struct file_inf *pt_temp;

{

                                    /* copy page to central memory */

  fnptpgln(pt_temp);

  if (mspgmc(pt_temp) !=1)
    {   printf("i-o error 10 0);
        return(0);
    }

                                    /* print line */

  if (prln(pt_temp) != 1)
        return(0);

  return(1);

}
```

```
int join_in_line_list(pt_first, pt_second)

            /* join the first and the second in line list */

struct line_list *pt_first;
struct line_list *pt_second;

{

  pt_first -> ptnxmbln = pt_second;
  pt_second -> ptpvmbln = pt_first;

  return;

}
```

```
docm_s(pt_temp,ptcharlncm)

      /* - substitute command : substitute an old string by a */

      /*    new one in the set of lines identified by the user's */
      /*    command ( pointed by ptcharlncm )                    */


struct file_inf *pt_temp;
char *ptcharlncm;

{

extern char *ptlntm;
extern char lsmd; /* last modifying command */
int lower;
int upper;
int lg_old_line; /* old length line */
int lg_new_line; /* new length line after substitution */
int work_length_new;
int num_saved; /* numeral of substituted line */
int lg_old; /* old string length */
int lg_new; /* new string length */
int number; /* number of identified lines */
int fail; /* status : can't substitute */
int *pt_lower;
int *pt_upper;
int *pt_lg_old;
int *pt_lg_new;
char old[MXLGLN]; /* old string */
char new[MXLGLN]; /* new string */
char *pt_old;
char *pt_find; /* first  character of the string in the line */
char *pt_line;
char *pt_new_string;
char *pt_new;
struct line_list *pt_member;
struct line_list *pt_saved;
int check_s_format( );

pt_new = new;
pt_old = old;
pt_lower = &lower;
pt_upper = &upper;
pt_lg_old = &lg_old;
pt_lg_new = &lg_new;
fail = 1;

  if (check_s_format(ptcharlncm,pt_temp,pt_old,pt_lg_old,pt_new,
                     pt_lg_new,pt_lower,pt_upper) == 0 )
        return;

  if ((number = ckexstln(pt_temp,pt_lower,pt_upper)) == 0 )
    {   printf ("not existent line 0);
        return;
    }
```

```
if (*pt_lower < MNNMLN || *pt_upper > MXNMLN)
  {   printf ("impossible  0");
      return;
  }


pt_member = pt_temp->ptmbtxli;
pt_line = ptlntm;

while (--number >= 0 )

                                    /* substitution */


  {   pt_temp->ptmbtxli = pt_member;

                                  /* copy line in central memory */

      if (ms_copy_line(pt_temp,pt_line) == 0)
              return;

        if ((pt_find = fnsr(pt_line,pt_old,pt_lg_old)) != NULL) /* search */
         {    fail = 0;
              lg_old_line = lnlg(pt_line);
              lg_new_line = lg_old_line + *pt_lg_new - *pt_lg_old;

              if (lg_new_line <= MXLGLN)              /* substitution */
                  { shift(pt_find + *pt_lg_old ,
                       lg_old_line - ((pt_find + *pt_lg_old) - pt_line)
                       , *pt_lg_new - *pt_lg_old);
                    work_length_new = *pt_lg_new;
                    pt_new_string = pt_new;

                    while (--work_length_new >= 0)
                       *pt_find++ = *pt_new_string++;

                    pt_saved = pt_member->ptpvmbln;
                    num_saved = pt_member->nmln;
                    pt_temp->ptmbtxli = pt_member;

                    if (dlln(pt_temp) == 0)
                          return;

                    pt_temp->ptmbtxli = pt_saved;

                    if (adln(pt_temp,num_saved) == 0)
                          return;

                    lsmd = 's';

                  }

              else

                                       /* too big line */
```

```
                printf("can't substitute in %d 0,pt_member->nmln);

            }

        pt_member = pt_member->ptnxmbln;

    }

                            /* no substitution */

  if (fail == 1)
        printf ("search failed 0);

  return;

}
```

```
int check_s_format(ptcharlncm,pt_temp,pt_old,pt_lg_old,pt_new,
                   pt_lg_new,pt_lower,pt_upper)

char *ptcharlncm; /* user's command */
struct file_inf *pt_temp;
char *pt_old; /* substitued string */
int *pt_lg_old; /* length of this string */
char *pt_new; /* new string */
int *pt_lg_new; /* length of the new string */
int *pt_lower; /* first numeral identified */
int *pt_upper; /* last line numeral identified */


{
char *pt_char;

  pt_char = skwh(++ptcharlncm);

  if (*pt_char != DL)
    {   printf ("format error 0);
        return(0);
    }

                          /* save old string */

  pt_char = svsr(++pt_char,pt_old,pt_lg_old);

  if (pt_char == NULL)
    {   printf("format error 0);
        return(0);
    }

  if (*pt_lg_old == 0 )
    {   printf("can't substitute 0);
        return(0);
    }

                           /* save new string */

  pt_char = svsr(pt_char,pt_new,pt_lg_new);

  if (pt_char == NULL)
    {   printf("format error 0);
        return(0);
    }

  pt_char = skwh(pt_char);
  pt_char = ckidstln(pt_temp,pt_char,pt_lower,pt_upper);

  if(pt_char == NULL)
        return(0);

  pt_char = skwh(pt_char);

  if(*pt_char != '0)
    {   printf ("format error 0);
```

```
            return(0);
        }

    return(1);

}
```

```
shift (pt_char,length,direction)

      /* - shift of the "length" characters after pt_char  */
      /*   to direction characters                         */
      /*         to left if direction < 0                  */
      /*         to right if direction > 0                 */
      /*         no shift if == 0                          */

char *pt_char;
int length;
int direction;

{

char *pt_to;
char *pt_from;

  if (direction < 0 )           /* left shift */
    {   pt_to = pt_char + direction ;
        while (--length >= 0)
                *pt_to++ = *pt_char++ ;
        return;
    }

  if (direction > 0 )           /* rigth shift */

    {   pt_to = pt_char + length + direction - 1;
        pt_from = pt_char + length - 1;

        while (--length >= 0 )
                *pt_to-- = *pt_from--;

        return;
    }

  return;                           /* no shift */

}
```

```
int docm_v(pt_temp, pt_edited, ptcharlncm)

                /* execute "v" command pointed by ptcharlncm     */
                /* with temporary file "pt_temp"                 */
                /* and edited file "pt_edited"                   */

struct file_inf *pt_temp;
struct file_inf *pt_edited;
char *ptcharlncm;

{ extern char lsmd; /* last modifying command */
  extern int lssffide; /* last safe file descriptor */
  char name[MXLGLN]; /* field for safe file name */
  char *pt_name; /* pointer to safe file name */
  int fd; /* safe file descriptor */
  int save( );
  int check_v_format( );
  int choose_safe_file( );

  pt_name = name;

  if (check_v_format(ptcharlncm, pt_edited, pt_name) == 0)
    return;

  if ((fd = choose_safe_file(pt_temp, pt_edited, pt_name))
      == NULL)
    return;

  if (save(pt_temp, fd) == 1)
    { lssffide = fd;
      lsmd = 'v';
    }
  return;

}
```

```
int check_v_format(ptcharlncm, pt_edited, pt_name)

                /* check format of "v" command pointed by ptcharlncm */
                /* copy name of safe file to *pt_name                */
                /* pt_edited : edited file                           */
                /* return 1 if correct format, 0 if wrong format     */

char *ptcharlncm;
struct file_inf *pt_edited;
char *pt_name;

{ char *pt_end;

  ptcharlncm = skwh(++ptcharlncm);

  if (*ptcharlncm == '0)
                /* safe file = edited file */
    strcpy(pt_name, pt_edited -> ptfina);
  else
    { pt_end = skfina(ptcharlncm);
      if (*skwh(pt_end) != '0)
        { printf("format error0);
          return(0);
        }
      *pt_end = '0';
      strcpy(pt_name, ptcharlncm);
    }

  return(1);

}
```

```
char *skfina(pt_char)          /* skip file name */
char *pt_char;

{ char ch;

  while ((ch = *pt_char) != ' ' && ch != '	' && ch != '0')
    pt_char++;

  return(pt_char);

}
```

```
int choose_safe_file(pt_temp, pt_edited, pt_char)

                /* return file descriptor of choosen safe file */
                /* or 0 if error                               */
                /* pt_temp : temporary file                    */
                /* pt_edited : edited file                     */
                /* pt_char : pointer to name of safe file      */

struct file_inf *pt_temp;
struct file_inf * pt_edited;
char *pt_char;

{ int fd; /* file descriptor for created new file */
  char ccc; /* defined for "sizeof" */
  char *pt_name; /* name of created safe file */
  int choosen; /* file descriptor for choosen safe file */
  int strcmp( );
  struct file_inf *search_safe( );

  if ((fd = search_safe(pt_temp -> ptsffili, pt_char)) != 0)
                        /* re-used safe file */
    return(fd);

  if (strcmp(pt_edited -> ptfina, pt_char) == 0)
                        /* safe file = edited file */
    return(pt_edited -> fide);

  /* create new safe file : */

  if ((fd = open(pt_char, 0)) != -1)
    { printf("error : exist file : %s0, pt_char);
      return(0);
    }

  if ((pt_name = calloc(MXLGLN, sizeof(ccc))) == -1)
    { printf("full core memory0);
      return(0);
    }
  else
      strcpy(pt_name, pt_char);

  close(creat(pt_name, FIMO));
  fd = open(pt_name, 2);

  if (add_safe_file(pt_temp -> ptsffili, fd, pt_name) == 0)
    return(0);
  else
    return(fd);

}
```

```
int strcmp(s,t)

      /* return <0if s<t, 0 if s == t, > 0 if s > t)  */

char *s,*t;

{
  for  ( ; *s == *t; s++,t++)

         if (*s == '0')
              return(0);

  return( *s - *t );

}
```

```
int search_safe(pt_safe_list, pt_char)

                /* seek for file "pt_char" in safe list "pt_safe_list" */
                /* return file_inf pointer if success, NULL if not     */

struct safe_files_inf *pt_safe_list;
char *pt_char;

{ struct safe_files_inf *pt_member; /* current member */

  pt_member = pt_safe_list -> ptnxmbsv;

  while (pt_member != pt_safe_list)
                /* search more */
    { if (strcmp(pt_char, pt_member -> ptfina_f) != 0)
        pt_member = pt_member -> ptnxmbsv;
      else
        return(pt_member -> fide_f);
    }

  return(0);
}
```

```
int add_safe_file(pt_safe_list, fd, pt_name)

                /* add new safe file (fd & pt_name) in "pt_safe_list" list */
                /* return 1 if ok, 0 if not                                 */

struct safe_files_inf *pt_safe_list;
int fd;
char *pt_name;

{ struct safe_files_inf *pt_member;
                /* member of safe files list */

  if ((pt_member = calloc(1, sizeof(s_safe_files_inf))) != -1)
    { pt_member -> ptnxmbsv = pt_safe_list -> ptnxmbsv;
      pt_member -> fide_f = fd;
      pt_member -> ptfina_f = pt_name;
      pt_safe_list -> ptnxmbsv = pt_member;
      return(1);
    }
  else
    { printf("full core memory 0);
      return(0);
    }

}
```

```
int save(pt_temp, fd)

                /* save text of scratch file "fd" to  */
                /* sequential file "pt_safe"          */
                /* return 1 if ok, 0 if i/o error     */

struct file_inf *pt_temp;
int fd;

{ struct line_list *pt_end; /* last member */
  struct line_list *pt_line_mb; /* current member */
  int putl( );

  pt_end = pt_temp -> pt_e;
  pt_line_mb = pt_temp -> pt_b -> ptnxmbln;

  while (pt_line_mb != pt_end)
                /* write more to safe file */
    { pt_temp -> ptmbtxli = pt_line_mb;
      fnptpgln(pt_temp);
      if (mspgmc(pt_temp) == 0)
        { printf("i/o error: %s0);
          return(0);
        }
      if (putl(fd, pt_temp -> ptbf + pt_line_mb -> dplnpg) == 0)
        { printf("i/o error0);
          return(0);
        }
      pt_line_mb = pt_line_mb -> ptnxmbln;
    }

  return(1);

}
```

```
int putl(fd, pt_char)

                /* copy line "pt_char" to file "fd" */
                /* return 1 if ok, 0 if not ok      */

int fd;
char *pt_char;

{ char ch;

  do
    { if (write(fd, pt_char++, 1) == -1)
        { printf("i/o error 0);
          return(0);
        }
    }
  while
    ( ch != '0 );

  return(1);

 }
```

```
#define PMODE 0644      /* file mode RW owner, R group, R others */

#define MXLGLN 150
```

```
main(argc, argv)
int argc;
char *argv[];

{ char cmd[MXLGLN];        /* command editor */
  char file_name[MXLGLN];   /* created file */
  char *pt_cmd; /* pointer to command */
  int strcpy( );



  if (argc == 1)   /* user error : no argument */
    { printf("not specified created file0);
      exit(1);
    }

  if (argc != 2)  /* user error : too many arguments */
    { printf("too many files0);
      exit(1);
    }

  pt_cmd = cmd ;
  strcpy(file_name, argv[1]);   /* save file name */

  if (open(file_name, 0) != -1)  /* user error : existent file */
    { printf("existent file0);
      exit(1);
    }

  if (creat(file_name, PMODE) == -1)  /* can't create */
   { printf("can't create0);
     exit(1);
  }

  strcpy(pt_cmd, "edit ");
  pt_cmd = pt_cmd + 5 ;  /* 5 = length of "edit " */
  strcpy(pt_cmd, file_name);
  while( *pt_cmd++ != '0' )
                        /* search '0' again */
                ;
  *--pt_cmd = '0;

  system(cmd);            /* call editor "edit" */

  exit(0);

}
```

```
int strcpy(s,t)

          /* copy s to t */

char *s,*t;

{
  while (( *s = *t ) != '0')
    {  s++;
       t++;
    }

}
```

Annexe 2 : LISTE DES VARIABLES.

Abréviations :

BF	buffer
BS	busy
CD	command
CR	character
DE	descriptor
DL	delimitor
DP	displacement
EN	end
FI	file
IN	information
LG	length
LI	list
LN	line
LS	last
MB	member
MD	modify
MN	minimum
MO	mode
MX	maximum
NA	name
NB	number
NM	numeral
NX	next
PG	page
PT	pointer
PV	previous
SE	step
SF	safe
SZ	size
TM	terminal
TS	test
TX	text

_b	begin (line)
_c	current (line)
_e	end (line)
NX	next
LS	last

Variables :

DL	delimitor
FIMO	file mode
LNTM	line terminal
LSMD	last modifying (command)
MNNMLN	minimal numeral of line
MXLGFINA	maximal length of file name
MXLGLN	maximal length of line
MXNMLN	maximal numeral of line
MXSZBF	maximal size of buffer
PTLNTM	pointer to line terminal
PTLSSFFI	pointer to last safe file
TSEN	test end

```
( file_inf : )

FIDE      file descriptor of  file
FIDE_     file descriptor of  corresponding file of numerals
NBLNFI    number of lines in  file
NMPGBF    numeral of  page in th buffer
NMSE      numeral = step
PTBF      pointer to buffer
PTFINA    pointer to  name of the file
PTFINA_   pointer to  name of the corresponding file of numerals
PTMBPGLI  pointer to member of page list
PTMBTXLI  pointer to member of text list
PTPGLI    pointer to page list
PTSFFILI  pointer to safe files list
PTTXLI    pointer to text list
pt_b      pointer to first line (begin)
pt_c      pointer to current line
pt_e      pointer to last line (end)

( lines list : )

DPLNPG    displacement of line in page
NMLN      numeral of line
NMPGLN    numeral of page
PTNXMBLN  pointer to next member
PTPVMBLN  pointer to previous member

( pages list : )

NMPGPG    numeral of page
NBBSCR    number of busy characters
PTNXMBPG  pointer to next member

( safe files list : )


PTFIIN    pointer to file information
PTNXMBSF  pointer to next member
```

Annexe 3 : FICHIERS D'AIDE.

Les "fichiers d'aide" sont des fichiers utilisés par notre éditeur. Chacun d'entre eux contient des informations susceptibles d'être demandées par l'utilisateur.

Ces fichiers sont repris dans cette annexe. Voici leur liste, avec, pour chacun d'entre eux :

- le nom du fichier,
- la commande d'édition que l'utilisateur doit envoyer à l'éditeur pour être informé sur son contenu, c'est-à-dire sur la commande d'édition dont les renseignements se trouvent dans le fichier,
- la référence du fichier dans cette annexe.

nom :	commande utilisateur :	page :
help_a	h a	92
help_c	h c	93
help_d	h d	94
help_e	h e	95
help_f	h f	96
help_h	h h	97
help_i	h i	98
help_m	h m	99
help_n	h n	100
help_p	h p	101
help_s	h s	102
help_v	h v	103
help_all	h	104

Name : a - append
Format : a
Definition :

The append command reads the given text and appends it after the last line.

Name : c - copy
Format : c ‹id. set lines› ^ ‹id. line›
Definition :

The copy command recopies the lines identified by ‹id. set lines› after the line identified by ‹id. line›.

Name : d - delete
Format : d ‹id. set lines›
Definition :

The delete command deletes all the lines identified by ‹id. set lines›.

Name : e - end
Format : e
Definition :

The end command causes edition to exit. Edition exits if the text has been asked to be saved. Otherwise, the text editor converses with its user to know if he really wants the editing process to be ended.

Name : f - find
Format : f <string>^<id. set lines>
Definition :

All the lines which belong to <id. set lines> and contain <string> are printed. An error message is sent if search fails.

Name : h - help
Format : h [<command name>]
Definition :
1) "h" gives information about the editor.
2) "h <command name>" gives particulars about the specified command.

Name : i
Format : i <id. line> ^ <int>
Definition :

The insert command inserts the given text after the line addressed by <id. line>. <int> is the maximum number of inserted lines. Insertion stops before this normal end if "^" is sent by the user as unique and first character of a line.

Name : m - move
Format : m ‹id. set lines› ^ ‹id. line›
Definition :

The m command moves the lines addressed by ‹id. set lines› after the line identified by ‹id. line›.

Name : n - number
Format : n <int>
Definition :

The number command gives the value <int> to the step and changes all the line identifiers, according to the new step.

Name : p - print
Format : p ‹id. set lines›
Definition :

The print command prints all the lines identified by ‹id. set lines›.

Name : s
Format : s ‹string a›^‹string b›^ ‹id. set lines›
Definition :
In every line of the set ‹id. set lines›, the substitute command puts ‹string b› in the place of each ‹string a›.

Name : v
Format : v [<file name>]
Definition :

1) "v" saves the edited text in the file the name of which is given when asking for edition ("privileged file").

2) "v <file name>" saves the edited text in the file <file name>. This file must be either the privileged file, or a file which didn't exist before asking for edition.

Terminology :

<int>	integer
<id. line>	identifier of line
<id. set lines>	identifier of a set of lines
<file name>	file name
<command name>	command name
^	separating character
[ ]	option

1) command n.

Name : n - number
Format : n <int>
Definition :
The number command gives the value <int> to the step and changes all the line identifiers, according to the new step.

2) commands searching text : f, p.

Name : f - find
Format : f <string>^<id. set lines>
Definition :
All the lines which belong to <id. set lines> and contain <string> are printed. An error message is sent if search fails.

Name : p - print
Format : p <id. set lines>
Definition :
The print command prints all the lines identified by <id. set lines>.

3) commands modifying text : a, c, d, i, m, s.

Name : a - append
Format : a
Definition :
The append command reads the given text and appends it after the last line.

Name : c - copy
Format : c <id. set lines> ^ <id. line>
Definition :
The copy command recopies the lines identified by <id. set lines> after the line identified by <id. line>.

Name : d - delete
Format : d <id. set lines>
Definition :
The delete command deletes all the lines identified by <id. set lines>.

Name : i
Format : i <id. line> ^ <int>
Definition :
The insert command inserts the given text after the line addressed by <id. line>. <int> is the maximum number of inserted lines. Insertion stops before this normal end if "^" is sent by the user as unique and first character of a line.

Name : m - move
Format : m <id. set lines> ^ <id. line>
Definition :
The m command moves the lines addressed by <id. set lines> after the line identified by <id. line>.

Name : s
Format : s <string a>^<string b>^ <id. set lines>
Definition :
In every line of the set <id. set lines>, the substitute command puts <string b> in the place of each <string a>.

4) commands v, e.

Name : v
Format : v [<file name>]
Definition :
1) "v" saves the edited text in the file the name of which is given when asking for edition ("privileged file").

2) "v <file name>" saves the edited text in the file <file name>. This file must be either the privileged file, or a file which didn't exist before asking for edition.

Name : e - end
Format : e
Definition :
The end command causes edition to exit. Edition exits if the text has been asked to be saved. Otherwise, the text editor converses with its user to know if he really wants the editing process to be ended.

5) command h.

Name : h - help
Format : h [<command name>]
Definition :
1) "h" gives information about the editor.
2) "h <command name>" gives particulars about the specified command.