Titre: Title:	Linéarisation de document hypertexte
Auteur: Author:	Martin Gagnon
Date:	1997
Type:	Mémoire ou thèse / Dissertation or Thesis
Référence: Citation:	Gagnon, M. (1997). Linéarisation de document hypertexte [Mémoire de maîtrise, École Polytechnique de Montréal]. PolyPublie. https://publications.polymtl.ca/6724/

UNIVERSITÉ DE MONTRÉAL

LINÉARISATION DE DOCUMENT HYPERTEXTE

MARTIN GAGNON
DÉPARTEMENT DE GÉNIE ÉLECTRIQUE
ET DE GÉNIE INFORMATIQUE
ÉCOLE POLYTECHNIQUE DE MONTRÉAL

MÉMOIRE PRÉSENTÉ EN VUE DE L'OBTENTION
DU DIPLÔME DE MAÎTRISE ÈS SCIENCES APPLIQUÉES
(GÉNIE ÉLECTRIQUE)
MAI 1997

UNIVERSITÉ DE MONTRÉAL

ÉCOLE POLYTECHNIQUE DE MONTRÉAL

Ce mémoire intitulé:

LINÉARISATION DE DOCUMENT HYPERTEXTE

présenté par: GAGNON Martin

en vue de l'obtention du diplôme de: Maîtrise ès sciences appliquées

a été dûment acceptée par le jury d'examen constitué de :

M. LANCTÔT Bernard, ing., président

M. DAGENAIS Michel, Ph.D.. membre et directeur de recherche

M. GRANGER Louis, M.Sc., membre

À ma mère.

Remerciements

J'aimerais remercier toutes les personnes qui, de près ou de loin, m'ont témoigné leur encouragement tout au long de ce travail.

Ces remerciements s'adressent plus particulièrement au professeur Michel Dagenais qui m'a témoigné toute sa confiance, tout en partageant sa précieuse expérience durant ces deux dernières années.

L'entraide et la complicité des membres de l'équipe formée de Benoit, de Jérôme et de Louis-Dominique, ont été bénéfiques pour l'accomplissement de ce projet.

Résumé

Les documents de type hypertexte HTML (Hypertext Markup Language) sont de plus en plus utilisés comme source d'information sur les médias électroniques. Cependant, ils sont basés sur une structure hiérarchique non-linéaire se prêtant mal à la mise en forme d'un texte linéaire respectant le contenu du document original. L'opération de linéarisation peut donc s'avérer longue et résulter en un manuscrit ayant une structure difforme. Cette recherche propose donc des règles permettant de développer un système capable de faire une linéarisation acceptable d'un document hypertexte spécifié par l'usager, tout en respectant ses intentions pour le texte final. Cette linéarisation doit comprendre certaines sections de synthèse comme une table des matières, un index et une liste des références. L'implantation d'un prototype ainsi qu'une analyse de ses performances sont aussi présentées.

Abstract

One of the major sources of information on the internet are HTML (Hypertext Markup Language) documents. These documents use a nonlinear structure, which cannot easily be converted to a linear representation. Consequently, the linearisation can take a long time and the final process may result in an incoherent text. This research presents a system and a set of rules to perform the linearisation of HTML documents. Furthermore, the linearisation process builds a table of contents, an index and a list of references from the HTML document. A prototype and its analysis are presented.

Table des matières

Liste des tableaux

Liste des figures

Liste des sigles et abbréviations

ASCII American Standard Code for Information Interchange

CPU Central Processor Unit

DTD Document Type Definition

FTP File Transfert Protocol

GUI Graphical User Interface

HTML HyperText Markup Language

HTTP HyperText Transfert Protocol

ISO International Standard Organization

Ko Kilooctets

Mo Megaoctets

NNTP News Network Transfer Protocol

RAM Random Access Memory

sec Seconde

SGML Standard Generalized Markup Language

URI Uniform Resource Identificator

URL Uniform Resource Locator

WAIS Wide Area Information System

WWW World Wide Web

Liste des annexes

Introduction

Au cours des dernières années, l'autoroute électronique s'est grandement développée. Ce fut certes à cause d'une plus grande rapidité des ordinateurs, et d'une augmentation des performances des réseaux de communication, mais surtout en raison de l'augmentation significative du nombre d'usagers grâce à une plus grande facilité d'accès.

Cette augmentation a engendré une plus grande utilisation des médias électroniques comme source d'information. En particulier, un des médias de l'autoroute électronique en pleine croissance est l'organisation World Wide Web (WWW). Cette organisation propose à ses usagers une quantité impresssionnante d'information provenant de divers points du globe. Cette information est accessible par l'intermédiaire de logiciels utilisant un réseau de communication pour transmettre la documentation. Le World Wide Web structure l'information sous forme de documents hypertextes. L'avantage des documents hypertextes est de développer de gros documents sous la

forme de petits documents liés ensemble par des liens logiques. Un document peut donc être vu comme un graphe où les noeuds sont associés aux fragments de texte, et les liens qui relient des points d'ancrage dans ces noeuds comme des relations entre ces fragments de texte. De cette façon, un gros document peut couvrir tous les aspects d'un langage informatique, alors qu'un de ses sous-documents ne s'interressera qu'aux librairies graphiques du langage.

Cette méthode permet à l'usager, lorsqu'il est à une station de travail, de rapidement consulter, par exemple, un détail précis dans un manuel de référence informatique. Par contre, s'il désire consulter cette information sous la forme d'un manuel imprimé, que ce soit pour l'apprentissage ou bien seulement pour éviter la restriction d'avoir à utiliser une station de travail, cette méthode peut exiger un effort supplémentaire de la part de l'usager pour produire le format désiré.

Qui de nous n'a jamais essayé d'imprimer l'information contenue dans un site World Wide Web qui nous intéressait? Même si certains logiciels, donnant accès au World Wide Web, nous permettent de traduire l'information dans un format directement imprimable, aucun de ces logiciels ne nous permet de parcourir automatiquement toute l'information contenue dans le site et d'en imprimer ainsi chacune des parties dans un ordre logique. De toute manière, le document produit ne contiendrait pas une table des matières, une liste des figures, une liste des références ou un index qui facilitent l'utilisation de gros documents.

1.1 Obectifs

L'objectif est donc de partir d'un noeud (fichier) faisant partie d'un site World Wide Web, de récursivement de suivre les liens appropriés pour générer un document linéaire cohérent, et de lui ajouter les sections suivantes : table des matières, liste de références, liste de figures et un index. Pour finir, il faut convertir ce document dans un autre format (ASCII, LaTeX, Postscript) pour en faciliter la mise en page.

1.2 World Wide Web

L'organisation World Wide Web peut se définir comme un ensemble de serveurs régis par des conventions et un protocole de communication HTTP (HyperText Transfert Protocole) (Berners-Lee, 1993). Cette organisation est accessible par différents types de logiciels disponibles sur le marché informatique.
Le langage utilisé par le WWW est le langage HTML (HyperText Markup Language) (Berners-Lee, 1993). Ce langage dérivé de la norme SGML (Standard Generalized Markup Language) (Nordin, Barnard et Macleod, 1993; Goldforbs, 1990) permet de structurer les documents tout en ayant la possibilité de créer des liens internes et externes entre eux.

1.3 Problématique

Plusieurs questions se posent cependant pour produire un document linéaire à partir d'un site World Wide Web:

- quels liens hypertextes doivent être suivis pour parcourir tout le graphe et briser les références cycliques;
- les niveaux d'entête (section, sous-section, etc.) doivent être ajustés selon la profondeur relative au noeud sélectionné par l'usager;
- le nom de chaque identificateur HTML doit être unique dans un noeud. Lorsque plusieurs noeuds sont joints ensembles dans un document linéaire, leurs identificateurs doivent être modifiés pour en assurer l'unicité;
- les liens à des identificateurs HTML inclus dans le noeud doivent être modifiés en fonction des changements effectués. Les liens à des noeuds auparavant externes, qui sont inclus dans le document linéaire, doivent être transformés en des liens internes et leurs identificateurs modifiés en conséquence;

- une table des matières, une liste des figures, un index ainsi qu'une liste de références bibliographiques peuvent être générés pour le document linéarisé.

Ce mémoire de maîtrise se présentera comme suit : Le chapitre suivant contient une revue bibliographique qui fait le point sur la recherche effectuée dans le domaine de la linéarisation de documents hypertextes. Ce chapitre définit aussi les normes et les protocoles utilisés. Il présente enfin les outils utilisés dans le cadre de ce projet. Le second chapitre présente les règles et l'algorithme de linéarisation que nous avons développé pour effectuer la linéarisation d'un document hypertexte. Le troisième chapitre met en évidence les avantages et désavantages du système proposé et présente une implantation du sytème. La performance de cette implantation est étudiée et discutée. La conclusion termine le mémoire.

Revue de la littérature

La recherche bibliographique se compose de trois parties. Le premier volet traite de la recherche faite sur la linéarisation de documents hypertextes. L'organisation World Wide Web étant très récente, la majorité des travaux se concentrent sur des documents créés par des outils permettant de structurer l'information sous forme hypertexte (HyperCard, gIBIS, etc.), plutôt que sur des documents construits à l'aide du langage HTML, spécifique au World Wide Web. La deuxième partie présente les normes et certains protocoles utiles à la compréhension du projet. Enfin, la dernière partie traite des outils utilisés pour implanter le prototype.

2.1 Linéarisation de documents hypertextes

La tâche de produire une "vue" linéaire d'un document hypertexte a attiré l'attention de plusieurs chercheurs dans le domaine. Selon Bench-Capon, Dunne et Staniford (1992,1993), la linéarisation est un cas spécial du problème de la naviguation dans

un environnement hypertexte. Ce problème comprends 2 difficultés majeures; tout d'abord la structure de liens d'un document hypertexte peut être extrêment complexe et ainsi augmenter le niveau de difficulté pour parcourir tout le texte. La seconde difficulté est que plusieurs documents linéaires peuvent être produits à partir d'un seul document hypertexte; certains de ces documents seront plus spécifiques à un groupe de lecteurs, d'où l'utilité de créer des classes de lecteurs pour répondre adéquatement aux besoins de chaque usager. Pour résoudre ces problèmes, les auteurs ont donc décidé d'ajouter des attributs à la structure du document et ainsi créer une description du document. Furuta (1988) définit la description d'un document comme étant la définition du contenu d'un document et la définition de la structure du document. Le contenu d'un document est formé d'objets atomiques tels que mots, figures, tables, etc., structurés de façon à communiquer le sens et l'intention de l'auteur au lecteur. La structure du document définit la forme du document en décrivant la façon dont les objets atomiques sont assemblés dans une structure de niveau supérieur. Dans un autre article, Bench-Capon, Dunne et Staniford (1992) ont proposé une manière de décrire la structure d'un document par des graphes acycliques. L'imposition de l'absence de cycle simplifie le problème de linéarisation. Ces graphes ont un nombre fini de noeuds et définissent explicitement un noeud d'entrée et de sortie qui permettent de parcourir tout le graphe. De plus, il est impossible de visiter le même noeud deux fois, puisque dans un document linéaire chaque fragment de texte doit apparaître seulement une fois. La linéarisation peut donc être vue comme une tâche qui s'effectue en deux temps. Premièrement, il faut trouver une façon de modéliser l'information hypertexte dans un document sous la forme d'un graphe acyclique. Deuxièmement, il faut produire une linéarisation du modèle construit. Les mêmes auteurs ont aussi été les premiers à proposer une définition formelle d'un document hypertexte :

Définition 2.1 : Un document hypertexte H, utilisant un ensemble de caractères (ou

alphabet). Σ est défini par le quintet de variables suivant :

$$H \equiv (V; E; \lambda_V; \lambda_E; \chi)$$

où $V = 1, 2, \ldots, n$ un ensemble fini de noeuds appartenant au graphe:

$E \subseteq V \times V$ est un ensemble fini de liens appartenant au graphe:

$\lambda_V : V \rightarrow \Sigma^*$ est une fonction d'étiquetage de noeuds:

$\lambda_E : E \rightarrow \Sigma^*$ est une fonction d'étiquetage de liens:

$\chi : V \rightarrow \Sigma^*$ est la correspondance décrivant le contenu de chaque noeud.

Définition 2.2: Pour une correspondance λ_R provenant d'un ensemble quelconque R sur Σ^*, on définit l'ensemble des *Noms(λ_R)* par :

$$\text{Noms } \lambda_R \stackrel{\text{def}}{=} \alpha : \exists x \in R \Rightarrow \lambda_R(x) = \alpha$$

Nous obtenons donc, *Noms(λ_V)* est l'ensemble des étiquettes pour les noeuds utilisés et *Noms(λ_E)* l'ensemble des étiquettes pour les liens utilisés. Nous assumons une étiquette nulle pour les deux ensembles.

Avec cette définition, les auteurs ont adopté une approche se basant sur l'utilisation d'expressions régulières provenant d'un langage formel et théorique. Cette approche peut se résumer de la façon suivante : En introduisant le concept d'expressions régulières étendues et en définissant comment ces expressions peuvent être utilisées pour définir une famille de graphes acycliques, on peut définir une correspondance entre les chemins d'un noeud source à un autre noeud situé dans le graphe et un ensemble de caractères générés par l'expression régulière correspondante. Finalement, en visualisant la linéarisation comme un sous-ensemble ordonné de noeuds hypertextes, et en admettant commme spécification une expression régulière, une linéarisation peut être définie comme étant la séquence de noeuds qui génère une chaîne de caractères pouvant correspondre à l'expression régulière spécifiée pour une certaine classe de lecteurs.

Cette approche comporte un certain nombre d'avantages et de désavantages. Parmi les avantages on peut citer la probabilité nulle de se perdre pendant la naviguation et surtout l'assurance d'extraire toute l'information essentielle même à partir d'un réseau de noeuds complexes. Par contre, certaines faiblesses peuvent survenir dans une application pratique. Tout d'abord la spécification de l'ordre d'apparition des noeuds dans un document ne peut être produite qu'à partir des liens qui sont explicitement définis. Une séquence ne pourrait pas utiliser des liens implicites pour exécuter la linéarisation puisque ceux-ci sont impossibles dans un environement hypertexte. Ensuite, les systèmes hypertextes sont construits sous la forme d'une hiérarchie de noeuds, alors que les graphes opèrent seulement au niveau du noeud, ils ne peuvent pas combiner plusieurs sous-niveaux de hierarchie simultanément. Dans un autre article, Subbotin et Subbotin (1992) ont développé un logiciel qui sert d'agent orienteur pour structurer et organiser de façon linéaire de l'information construite sous forme hypertexte. Leur logiciel traite l'information hypertexte en se basant sur les concepts suivants :

- Un noeud hypertexte se définit comme étant un texte indépendant véhiculant une idée, un fait ou une affirmation, i.e. une unité monosémantique.

- Un mot-clé peut être assigné à un noeud pour en définir son contenu. L'usage principal des mots-clé se situe dans le processus automatique de création de liens entre les noeuds. Ils jouent le rôle de descripteur externe d'un noeud et de son contenu.

- Les liens hypertextes s'établissent entre les noeuds en se basant uniquement sur leurs similitudes sémantiques. Une similitude sémantique s'établit lorsqu'une paire de noeuds peuvent être connectés par un lien du type "par conséquence", "par exemple", "en se basant sur", etc. Pour éviter une approche arbitraire, tous les liens pouvant être établis doivent l'être. Les liens sont bidirectionnels et ne comportent aucun poids puisque la cohérence d'un texte ne dépend aucunement

du type de relation qui unit les phrases qui le composent.

Les auteurs ont aussi identifié l'importance de développer une interface flexible entre le linéarisateur et n'importe quel type de base données ou de réseau, ceci pour permettre plus facilement l'importation de l'information vers le linéarisateur et l'exportation du document linéaire vers l'usager.

Dans un autre article, Sharples et Godlet (1994) ont présenté une comparaison de deux algorithmes permettant d'effectuer une linéarisation sur un réseau de noeuds hypertextes.

Les auteurs ont d'abord identifié les contraintes pour un algorithme général de linéarisation.

- L'algorithme doit s'appliquer à n'importe quel type de graphe, cyclique ou acyclique, comprenant des liens étiquetés ou non. La linéarisation doit donc pouvoir s'effectuer sur un graphe cyclique avec des liens sans étiquette, tout en prenant avantage de l'information sur les liens hypertextes si celle-ci est disponible.

- Il doit être en mesure de pouvoir effectuer une linéarisation à partir de n'importe quel noeud de départ, et parcourir l'ensemble des noeuds.

- Il doit garantir la fin de la linéarisation lorsque tous les noeuds du graphe ont été ajoutés à la liste linéaire. Cette condition peut être omise si on a de bonnes raisons d'inclure un noeud plusieurs fois dans la liste (par exemple si ceux-ci sont séparés par une grande distance dans le graphe).

- Le temps pour traverser un réseau de noeuds doit être acceptable pour un usager lorsque le graphe ne comporte pas plus d'une centaine de noeuds. La complexité de l'algorithme devrait être généralement de l'ordre $O(n)$ où n est le nombre de noeuds dans le graphe.

- L'algorithme doit être en mesure d'utiliser toute l'information disponible pour effectuer la linéarisation. Il ne devra pas exiger d'aide de la part de l'usager pour guider la linéarisation, sauf pour indiquer le noeud de départ. L'information que l'algorithme peut utiliser est le nom des noeuds, le contenu des noeuds, l'étiquette des liens, la connectivité des noeuds dans le graphes, le temps de création et l'espace entre les noeuds dans le graphe.
- L'algorithme doit être déterministe, l'ordre de la linéarisation doit être entièrement déterminé par l'information contenue dans le réseau, et non pas par des caractéristiques arbitraires du programme de linéarisation.
- Il doit être en mesure de produire une linéarisation acceptable pour un lecteur.

Tel qu'expliqué précédemment, l'organisation World Wide Web s'étant développée récemment, la majorité de la littérature que nous venons de parcourir se concentre sur la linéarisation de base de données créées à partir d'outils spécifiques à certaines plate-formes et développées dans un but particulier. Ainsi la majorité de ces outils ne suivent aucune norme pour construire leurs documents hypertextes et utilisent des langages qui leur sont spécifiques. L'avantage du World Wide Web est d'utiliser un langage qui est dérivé d'une norme, SGML, et qui rend la structure de l'information identique pour tous les utilisateurs.

2.2 Normes et protocoles

2.2.1 SGML

Le langage SGML (Standard Generalized Markup Language) (Nordin, Barnard et Macleod, 1993; Goldforbs, 1990) permet de représenter la structure d'un document. Ce langage agit comme une norme dans la définition de document. Développé par un groupe de travail composé d'experts internationaux, SGML fut défini comme la norme ISO 8879. SGML définit les commandes textuelles qui sont ajoutées aux

données du document. Ces commandes sont ensuite utilisées pour traiter l'information contenue dans le document. et sont appelées délimiteurs (markup). Il existe 4 types de délimiteurs :

- délimiteurs descriptifs (mnemonics ou tags):
- références:
- délimiteurs déclaratifs;
- instructions.

Le langage SGML est donc un outil qui permet de définir et d'implanter certains aspects de l'ensemble des processus qui sont appliqués sur un document. Cet ensemble de processus est appelé application SGML. Une application SGML inclut une spécification formelle des délimiteurs utilisés pour structurer le document.
Une application SGML inclut normalement une définition du type du document. un ensemble d'entités. et une notation du contenu de l'information. La définition du type de document (DTD) est l'ensemble des règles d'une application qui applique le système SGML aux délimiteurs d'un type de document en particulier. Les paragraphes suivant montrent la relation entre une instance d'un document SGML et sa DTD.
Une instance d'un document SGML:

<section>
Summary of &plan; Plan Elements
<section>
Highway System
<para>
A base network of roads for people
goods movement designed to operate at maximum effiency
</para>

```
<list>
 *Completion of Measure &oquote; A&cquote;
 *Emphasis on commuter lanes
 *Capacity improvements in 101
</list>
</section>
</section>
```

et la DTD correspondante:

```
<! ENTITY % text "(# PCDATA — xref)*" >
<! ELEMENT section – (head, para—list)*,section*)>
<! ELEMENT (head, para) - O (%text)>
<! ELEMENT list – (item+) >
<! ATTLIST list type (bullet—number)bullet)>
<! ELEMENT item - O (%text; ,list? )>
<! ENTITY strtitem STARTTAG "item">
<! SHORTREF listmap "*" stritem>
<! USEMAP listmap list>
<! ELEMENT xref - O EMPTY>
```

Dans l'exemple précédent on peut remarquer dans la DTD, la définition d'une liste pouvant contenir un ou plusieurs items. Le délimiteur de l'item peut aussi être abrégé en utilisant le caractère "*" (défini à l'aide de la commande SHORTREF). C'est ce qui est utilisé dans l'instance du document SGML associé à cette DTD, lors de la définition d'une liste d'items.

SGML peut représenter des documents de structure arbitraire en modélisant ceux-ci sous forme d'arbre contenant des connexions supplémentaires entre les noeuds. Cette technique s'applique bien puisqu'en pratique la majorité des documents conventionnels sont en fait des arbres structurés, à l'exception des noeuds terminaux (feuilles)

qui eux contiennent l'information textuelle. Chaque noeud dans un document SGML est un élément où le descendant du noeud est le contenu de cet élément. (voir figure 2.1)

Un élément est une composante hiéarchique définie par le DTD. Il est identifié dans l'instance d'un document par un délimiteur descriptif. Un type d'élément est une classe d'éléments ayant des caractéristiques similaires (i.e. paragraphes, chapitre, résumé ou bibliographie). Un élément peut aussi se caractériser par certains attributs qui lui sont spécifiques.

Une application SGML est une abstraction qui peut être réalisée sous différentes formes. Pour des raisons pratiques, l'implantation d'une application SGML se fait souvent à l'aide d'un système informatique comprenant un logiciel spécifique au type de document. Cette implantation se nomme *système SGML* et comprend un analyseur lexical SGML et un outil qui gère les entités. Par exemple, FrameMaker offre un environnement intégré permettant de développer des applications SGML. Cet environnement comprend un outil permettant de définir les éléments du langage, un analyseur lexical et des filtres permettant de convertir des documents SGML dans un format particulier. Une entité est une collection de caractères qui peut être référencée comme étant une unité. Un analyseur lexical SGML est un programme (ou une portion de programme, ou une combinaison de plusieurs programmes) qui reconnait les délimiteurs dans un document SGML. L'analyseur lexical SGML exécute les mêmes fonctions que l'analyseur lexical dans un compilateur de langage informatique. L'outil qui gère les entités est aussi un programme qui, comme un système de fichier ou une table de symboles, maintient et donne accès aux multiples entités. Bien que SGML établit la représentation des documents et les spécifications du processus, il ne codifie pas la sémantique du processus. Comme résultat, une application s'exécutant sur un document SGML peut faire appel à un analyseur lexical SGML pour reconnaître les spécifications du processus, mais l'application elle-même devra

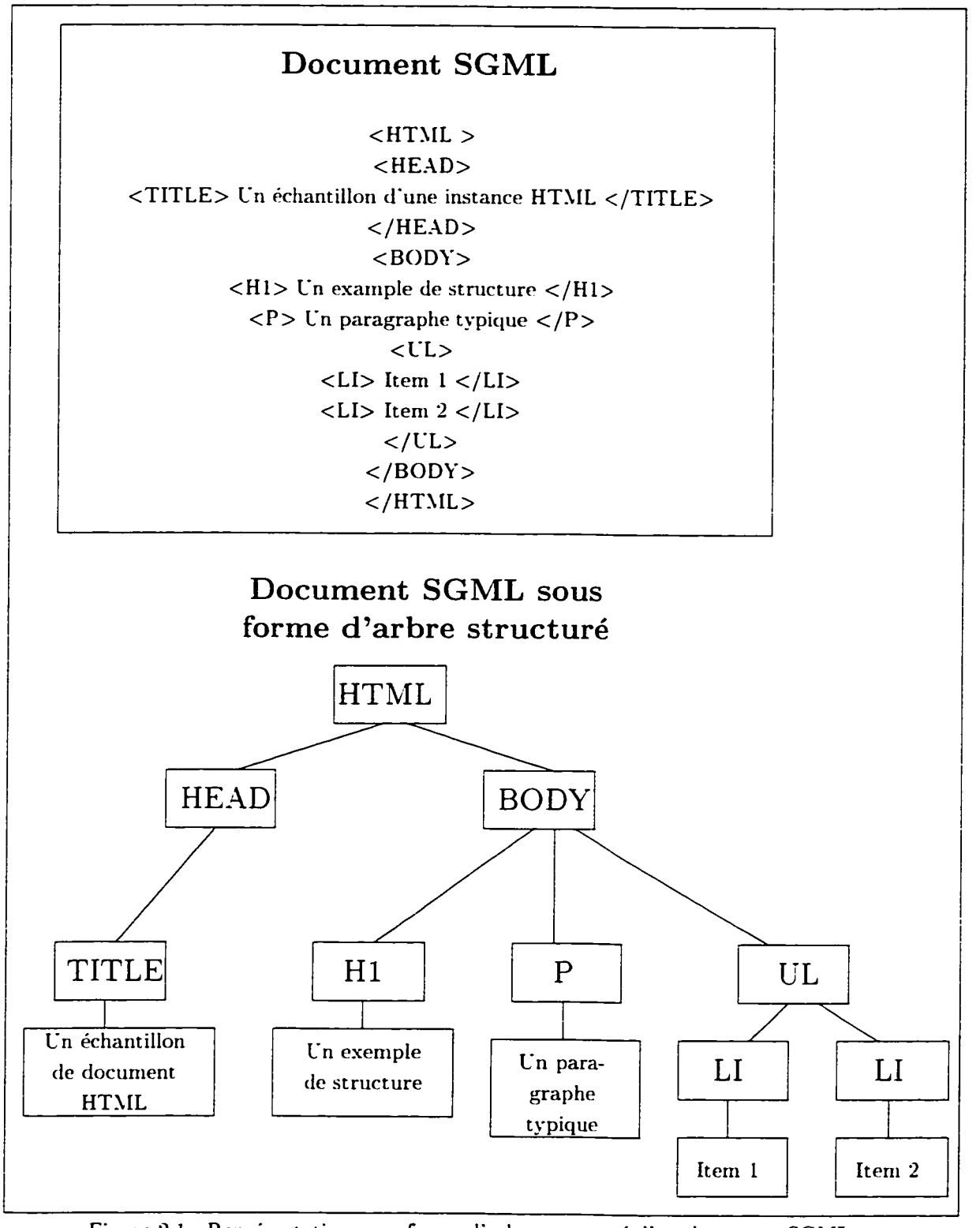


Figure 2.1: Représentation sous forme d'arbre structuré d'un document SGML

les interpréter et agir en conséquence selon la commande reconnue. Cette partie est appelée l'architecture du document et contrôle tous les aspects de l'application.
Les principaux avantages d'utiliser SGML pour représenter un document sont :

- l'utilisation de programmes communs qui sont indépendants de l'architecture du document et qui opèrent directement sur la structure du document:
- l'usager peut diviser un document en plusieurs sous-documents transparents à l'architecture:
- des liens hypertextes peuvent être établis dans le document et entre les documents, tout en étant transparents à l'architecture.

On peut donc résumer qu'un document SGML est représenté comme une séquence de caractères, physiquement organisés dans une entité, et logiquement organisés dans un élement tel que décrit plus haut. Il comprend des caractères de données qui représentent le contenu de l'information et des caractères délimiteurs qui représentent la structure des données.
L'organisation logique d'un document consiste en une structure d'élément d'une ou plusieurs hiérarchies (arbre), chacune se conformant à sa définition de type de document (DTD). Les entités comprises dans le document représentent l'organisation physique.
SGML coordonne donc tous les aspects de la définition d'un type de document. Ainsi un langage peut utiliser cet outil pour définir ses propres spécifications. C'est ce que le langage HTML (Hypertext Markup Language) effectue pour établir ses règles de définition.

2.2.2 HTML

Le langage HTML (Hypertext Markup Language) (Berners-Lee, 1993) est un système simple de délimiteurs utilisés pour créer des documents hypertextes et qui est portable

d'une plate-forme à une autre. Les documents HTML sont des documents SGML avec une sémantique générique appropriée pour représenter l'information pour une vaste variété d'aplications. Il peut servir de medium pour représenter le courrier électronique, des groupes de nouvelles (*newsgroup*) hypertextes, de la documentation hypermédia , de la documentation conventionnelle, des menus avec options ou des résultats de recherche dans des bases de données.

HTML est utilisé par le groupe d'information globale World Wide Web depuis 1990. Le langage est défini comme une application du Standard International ISO8879: 1986 Standard Generalized Markup Language (SGML). Ce langage n'est pas encore un standard formel pour le World Wide Web bien que le groupe de travail IETF HTML travaille pour peaufiner le langage pour qu'il puisse le devenir. Le langage HTML est un médium pour lier ensemble l'information provenant de différentes plates-formes. Il permet d'éviter les problèmes de compatibilité entre différents types de format de documents et offre une interface simple aux propriétaires de systèmes d'information.

Une instance HTML ressemble à un fichier texte, à l'exception que certains caractères sont interprétés comme des délimiteurs. L'instance représente une hiérarchie d'éléments. Chacun de ces éléments a un nom, des attributs, et un contenu. La plupart des éléments ont un délimiteur de départ qui contient le nom ainsi que les attributs, un contenu et un délimiteur terminal. Par exemble un document HTML peut ressembler à ceci :

<HTML>

<HEAD>

<TITLE> Un échantillon d'une instance HTML

</TITLE>

</HEAD>

<BODY>

<H1>

Un exemple de structure

</H1>

<P> Un paragraphe typique </P>

<UL>

<LI>

L'item 1 a un <A NAME="lien">

lien

</A> </LI>

<LI>

Un deuxième item </LI>

</UL>

</BODY>

</HTML>

Chaque élément débute avec un délimiteur, et tous les éléments qui ne sont pas vides se terminent avec un délimiteur. Les délimiteurs de début sont marqués par un < et un >, et les délimiteurs terminaux par un </ et un >. Le nom de l'élément suit immédiatemment le délimiteur de début. Les attributs d'un élément consistent en un nom suivi d'un signe égal et d'une valeur. La valeur est habituellement spécifiée par une chaîne de caractères incluse entre des guillemets.

Une propriété importante du langage est l'ancrage. L'ancrage est un morceau de texte qui identifie le début ou la fin d'un lien hypertexte. Le texte situé entre le début et la fin des délimiteurs est soit la destination ou le départ du lien. Les liens hypertextes dans le langage HTML sont exprimés à l'aide d'un identificateur de

ressource universelle (URI) (Berners-Lee, 1993). Cet identificateur définit le protocole de communication avec lequel le lien hypertexte sera accédé, ainsi que la localisation de la destination. Les protocoles de communication supportés habituellement par les logiciels qui donnent accès au WWW sont :

- HTTP (Hypertext Transfert Protocol) (Berners-Lee, 1993);
- FTP (File Transfert Protocol) (Postel et Reynolds, 1985);
- Gopher (Ankleseria, McCahill, Lindner, Jonshon, Torrey et Alberti, 1993);
- NNTP (Network News Transfert Protocol) (Kantor et Lapsley, 1986);
- WAIS (Wide Area Information Service) (Davis, Kahle, Morris, Salem, Shen, Wang, Sui et Grinbaum, 1990).

Ces protocoles de communication fonctionnent sur la base d'un paradigme requête / réponse. Le programme requérant (appelé client) établit une connexion avec un programme répondant (appelé serveur) et envoie une requête au serveur sous la forme d'une commande contenant une version du protocole, un identificateur de ressource universelle, de l'information sur le client et possiblement de l'information. Le serveur répond par une ligne statuant sa version du protocole ainsi qu'un code d'erreur ou de succès, suivi du contenu de l'information.

Certains de ces protocoles (HTTP et FTP) supportent la présence de serveur proxy (*proxy server*) et de passerelle (*gateway*). Un serveur proxy est un programme intermédiaire qui agit comme client et comme serveur permettant de rediriger les requêtes. Ce type de programme est habituellement utilisé comme porte d'entrée dans un réseau protégé par un mur coupe-feu (*firewall*). Une passerelle est un serveur proxy qui permet, en plus de rediriger les requêtes, de les traduire dans un protocole différent.

2.3 Environnement de développement

Pour développer ce projet nous avons utilisé l'environnement de développement Modula-3, qui offrait une grande variété de librairies, ainsi que plusieurs outils permettant de déverminer et d'analyser la performance d'un logiciel en exécution. Cet environnement de développement offrait aussi une librairie graphique, FormsVBT, qui facilite la tâche pour la réalisation de l'interface graphique du prototype. Les sections suivantes décrivent brièvement le langage Modula-3 ainsi que sa librarie graphique FormsVBT. Puisque les résultats de la linéarisation pourront être présentés selon deux types de format ASCII et LaTeX, la dernière section présentera rapidement le format ASCII et le langage LaTeX.

2.3.1 Modula-3

La connaissance des notions de langages orientés objets sera essentielle au développement de notre prototype. Un avantage intéressant des langages orientés objets est leur facilité de modéliser un problème réel. Par exemple, une entité physique tel qu'un fichier, représentant un fragment de texte dans un document hypertexte, deviendra désormais un objet nommé fichier qui aura ses attributs tel que le nom du fichier, son chemin relatif, son extension, et qui possédera ses propres fonctions spécifiques (méthodes). La programmation modulaire facilite la manipulation des objets. Les principales caractéristiques des langages orientés objets s'énumèrent comme suit:

- structure modulaire;
- abstraction des données;
- gestion automatique de la mémoire (Ramasse-miette automatique);
- notions de classes et d'héritage;

- liaisons dynamiques;
- polymorphisme;
- héritage multiple.

Le langage Modula-3 (Harbinson, 1992) ne possède pas la notion d'héritage multiple. Cependant, le langage orienté objet Modula-3, possède en plus, les caractéristiques suivantes:

- contextes d'exécution multiple;
- un mécanisme facilitant la récupération des erreurs;
- une excellente librairie d'outils graphiques.

2.3.2 FormsVBT

FormsVBT est un système conçu pour construire des interfaces usagers graphiques (GUI Graphical User Interfaces). Développé par Marc Brown et James Meehan (1993), FormsVBT consiste en un langage permettant de décrire l'interface entre l'usager et l'application, un interpréteur permettant de construire l'interface, et une librairie dynamique permettant la communication entre le code de l'application et l'interface usager.
L'interpréteur de FormsVBT (*formsedit*) offre un éditeur de texte ainsi qu'une vue de l'interface usager lorsque celle-ci est interprétée correctement.

2.4 ASCII

Le format ASCII (American Standard Code for Information Interchange) est un système permettant d'encoder les caractères pour que ceux-ci puissent être lus en format binaire par les ordinateurs.

2.5 LaTeX

Conçu par Leslie Lamport (1994), LaTeX est un langage permettant de construire un système pour préparer et mettre en page une grande variété de documents. Basé sur le programme TeX, créé par Donald Knuth (1994), la première version apparut vers 1985, et depuis ce temps LaTeX s'imposa rapidement comme le langage pour écrire un document ou un article dans la communauté scientifique et académique. L'aspect bibliographique du projet de recherche venant d'être couvert, le prochain chapitre présente le système proposé en détail.

Définition du système

Le système présenté dans cette recherche se base sur une série de règles qui sont intégrées dans le document lors de sa structuration afin d'en simplifier la linéarisation. La linéarisation s'effectue d'abord en extrayant l'information relative aux règles de linéarisation et ensuite en les interprétant pour construire le document linéaire. Cette méthode n'utilise donc aucune heuristique, tel que vu précédemment dans la recherche bibliographique. Par contre, dans le futur, le système pourrait incorporer certaines fonctions heuristiques permettant la linéarisation de documents n'ayant pas été construits selon ces règles.

En général, un document hypertexte situé sur un site World Wide Web comporte toujours un noeud d'entrée communément appelé *index.html*. Cet index sert habituellement à introduire au lecteur le sujet général traité par le site, et ensuite le référer aux sujets spécifiques à l'aide de liens aux sous-documents. Ces liens sont habituellement ordonnés dans une liste: il est donc facile de linéariser ce genre de document puisque le noeud d'entrée sert d'introduction au document linéaire, et liste les liens

aux éléments qui forment les sections ou chapitres. Malheureusement, tous les sites ne sont pas structurés de cette façon, d'où l'utilité d'utiliser certaines heuristiques permettant de diriger la linéarisation dans le graphe de noeuds pour produire un document linéaire cohérent.

L'adoption d'un système utilisant des règles prédéfinies repose sur le fait qu'il est plus simple de produire un document linéaire qui respecte les intentions premières du rédacteur si celui-ci a déjà spécifié à la construction du document hypertexte une vue linéaire de celui-ci. En effet l'utilisation d'heuristique exige une complexité beaucoup plus grande pour produire un document linéaire cohérent sans intervention humaine.

Un site World Wide Web peut être modélisé selon la figure 3.1. Cette structure représente un réseau de noeuds où chaque noeud représente un fichier HTML, qui peut contenir un document ou une partie de document HTML. Les liens entre les noeuds représentent les références que chaque document hypertexte (fichier HTML) établit avec les autres documents. Deux types de références existent : les références externes (exemple : lien A et B pour le noeud # 1 sur la figure 3.1) qui sont des liens dont le point d'ancrage de la destination se situe dans un autre noeud (fichier HTML), et les références internes (exemple : lien C pour le noeud # 2 sur la figure 3.1) qui sont des liens où le point d'ancrage de la destination est situé dans le même fichier HTML.

Tel que vu dans le chapitre précedent, un fichier HTML (correspondant à un noeud dans un site Web) est habituellement structuré de cette façon :

<HTML>

<HEAD>

<TITLE> Un échantillon d'une instance HTML </TITLE>

</HEAD>

<BODY>

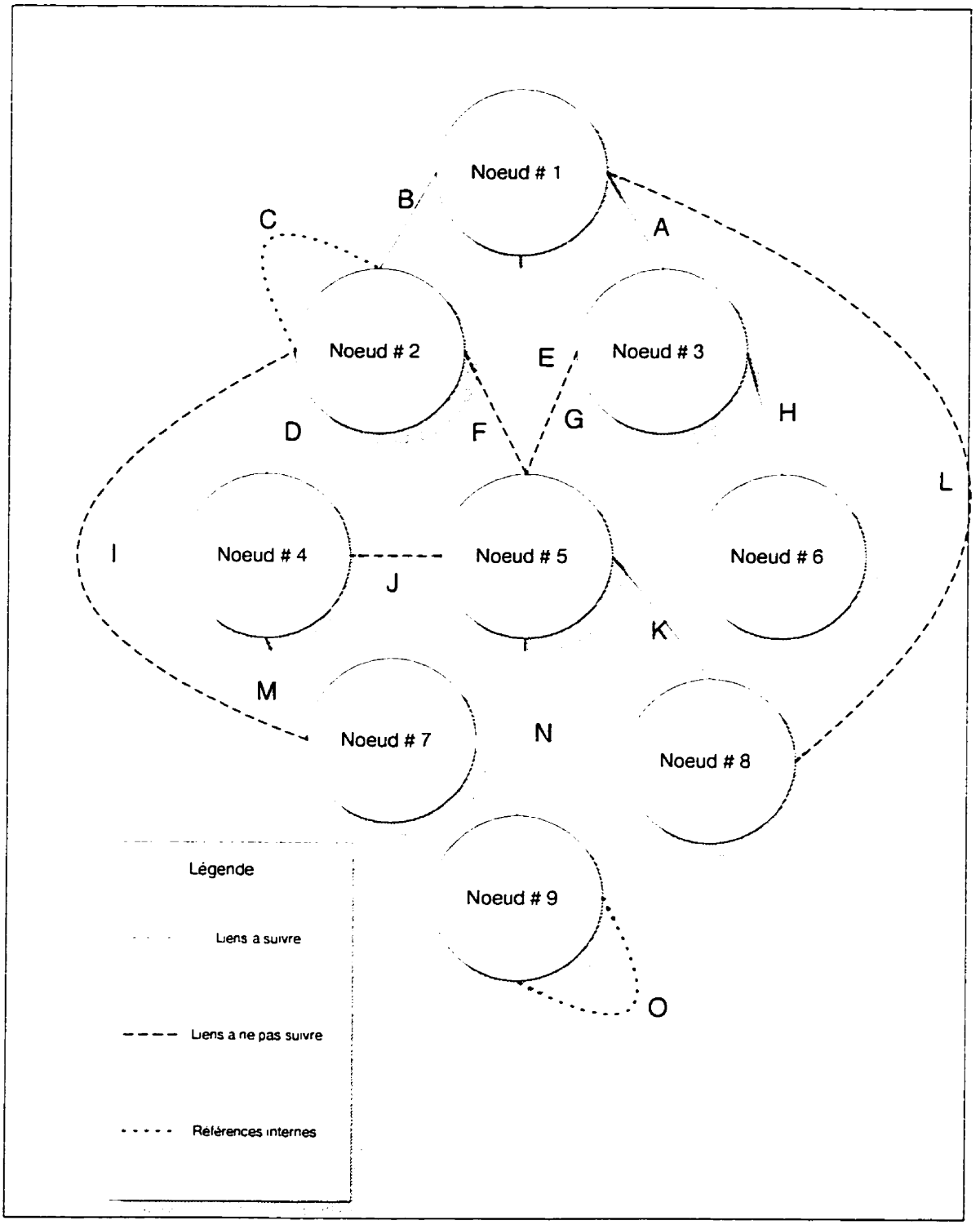


Figure 3.1: Structure d'un site World Wide Web

<H1>

Un example de structure

</H1>

<P> Un paragraphe typique </P>

<UL>

<LI>

L'item 1 a un <A NAME="lien">

lien

</A>

</LI>

<LI>

Un deuxième item

</LI>

</UL>

</BODY>

</HTML>

Dans cette structure, seulement le titre est obligatoire. La structure <BODY> débute implicitement après le <TITLE>, si le <HEAD> et le <BODY> n'ont pas été spécifiés. Lors d'une linéarisation, la progression à travers le *Web* débute au noeud spécifié comme étant la racine. À partir de ce point, les liens hypertextes sont suivis dans une recherche en profondeur en suivant les règles définies par le système et l'algorithme de linéarisation.

3.1 Définition des règles

Les règles définies pour effectuer la linéarisation sont les suivantes :

- À chaque noeud atteint, on assigne un identificateur unique dérivé de son URL (Uniform Resource Locator). Une liste des noeuds visités, appelée liste VISITE, est construite. La liste contient pour chaque noeud visité : l'identificateur du noeud , son URL et son chemin relativement au noeud spécifié comme étant la racine.

- À chaque noeud atteint, on vérifie si ce noeud n'a pas été précédemment visité pour identifier toute référence cyclique. Cette vérification s'effectue à l'aide de la liste de noeuds VISITE construite à la première règle. Lorsqu'un noeud déjà visité est rencontré de nouveau, il peut ou ne pas être suivi selon l'option de l'usager, mais un avertissement est émis.

- Dans le langage HTML, les sections (chapitres, sections, sous-sections, ...) sont délimitées par les éléments <DIV> et </DIV>. Ces éléments peuvent être utilisés comme récipients pour encapsuler les niveaux, et peuvent être caractérisés par l'attribut CLASS du langage. Le niveau des entêtes (<H1>) identifiera le titre de la section alors que sa profondeur sera relative au nombre de divisions dans lequel il est inclus. En pratique, peu de documents utilisent l'élément <DIV >pour spécifier la profondeur. La convention proposée ici est que la profondeur effective d'un entête soit le nombre de divisions ayant la classe SECTION, dans lequel elle est incluse, plus la valeur de son entête.

- L'élément DIV peut être aussi utilisé comme identificateur de section de document (<DIV CLASS=ABSTRACT >... </DIV> <DIV CLASS=APPENDIX> ... </DIV>). Si un élément DIV sert à la fois de délimiteur hiérarchique et d'identificateur, sa syntaxe sera la suivante <DIV CLASS=SECTION,ABSTRACT>.

- Les liens avec l'attribut REL=INCLUDE sont automatiquement suivis. Le lien sera remplacé par l'élément DIV avec son chemin comme identificateur. La

valeur de ses entêtes ne sera pas modifiée. L'opération se fera comme suit :
<A REL=INCLUDES HREF="sub/toto.html">toto</A>
sera remplacé par
<DIV ID="sub/toto.html">
le BODY de toto.html
</DIV>

- Les liens avec l'attribut REL= SUBDOCUMENT sont aussi automatiquement suivis.

Définition 3.1 : Pour obtenir le niveau d'un en-tête en fonction de sa position dans la structure du document et de son type, on utilise la définition suivante:

$$N_{linéaire} = N_{hiérarchique} + N_{div}$$

où $N_{linéaire}$ représente le niveau de l'en-tête dans le document linéaire.
$N_{hiérarchique}$ représente le niveau de l'en-tête dans le document situé sur le *Web*.
N_{div} le nombre d'inclusion de l'en-tête dans une division (élément DIV) ayant l'attribut CLASS=SECTION.
La valeur des en-têtes sera dimimuée à cause de l'attribut CLASS=SECTION lors de la conversion en LaTeX.

- Dans les deux cas précédents, il se peut que le noeud inclus contienne un titre inapproprié dans le contexte de l'inclusion. Si le lien contient l'attribut CLASS=SKIPTITLE, alors le premier en-tête sera enlevé, par contre s'il contient l'attribut CLASS=REPLACETITLE, le texte utilisé dans le lien (<A ...> texte dans le lien </A>) sera utilisé pour remplacer le premier en-tête dans le noeud inclus.

- Lorsqu'un noeud est inclus, tous ses identificateurs seront remplacés par la fusion de l'identificateur du noeud avec l'ancien identificateur. De cette façon, tous les identificateurs resteront globalement uniques.

- Tous les autres liens ne sont pas suivis; toutefois s'ils font référence à des identificateurs internes au noeud, cette référence doit être remplacée par la fusion de l'identificateur du noeud et l'ancien identificateur.

- Les liens qui réfèrent à des sous-documents sont souvent placés dans une liste. Toutefois, lorsqu'on remplace ces liens par des noeuds, la liste doit être retirée. Une liste qui contient seulement ces liens est retirée; par contre une liste qui n'en contient que quelques-uns n'est pas retirée mais un avertissement est émis. De la même manière, si certains éléments ne sont pas désirés dans la version linéaire, ils sont identifiés avec l'attribut CLASS de la façon suivante CLASS=SKIPLINEAR.

- Comme utilité, il sera possible de spécifier à la linéarisation un ensemble de règles qui affectent les liens à suivre. Il est possible de modifier ou de remplacer les attributs HREF et REL si ceux-ci correspondent au patron défini par l'usager. Cette utilité permettra de modifier les relations entre les liens (remplacer REL=SUBDOCUMENT par REL="" par exemple) ou bien de rediriger une référence vers un autre document.

- Une fois la linéarisation terminée, une deuxième passe sera nécessaire pour ajuster les liens internes. Ce réajustement est nécessaire comme le démontrent les figures 3.2 et 3.3. La figure 3.2 montre la structure du site avant la linéarisation et la figure 3.3 la structure du site après la linéarisation.
  Lorsqu'on linéarise le réseau de noeuds vers le document linéaire, on suit tous les liens qui portent l'étiquette REL=SUBDOCUMENT ou REL=INCLUDE. Dans notre cas, il s'agit des liens a, b, c, et i. Les liens d, e, f, g, h, j, k et l qui

n'ont pas d'étiquette sont traités comme des références du document linéaire. Or, comme vu précédemment, deux types de références existent: les références internes et les références externes. Dans notre cas, il faut déterminer si ces liens sont des références internes ou externes et ajuster leur lien en conséquence dans la version linéaire de la structure. Ceci est effectué en vérifiant que tous les liens vers les noeuds qui apparaissent dans la liste de noeuds visités, ont leur attribut HREF remplacé par la fusion de l'identificateur du noeud ainsi que de l'ancien identificateur. Dans l'exemple, les liens f et g qui étaient des références externes dans la structure non-linéaire (ils pointent vers les noeuds 3 et 5) deviennent des références internes lors du processus de linéarisation. Les liens k et h sont aussi des références externes dans la structure non-linéaire; par contre, ils pointent vers les noeuds 6 et 7 qui n'ont pas été visités (ils ne sont pas présents dans la liste des noeuds visités), et restent donc des références externes au document linéaire.

- Quelquefois, de l'information supplémentaire pourra être emmagasinée dans les noeuds. Cette information pourrait contenir les données nécessaires à la création d'un éventuel index, ou une courte description à ajouter à la table des matières, et même une référence bibliographique au document. Le langage HTML 3 ne contient aucun élement (à part le champs FORMS) qui permet de stocker de l'information qui ne sera pas affichée. Pour résoudre ce problème, de telles informations peuvent être encodées dans le champ ID de l'élément SPOT de cette façon:

  < SPOT CLASS=INDEX ID="xxx_key_text" >

  Dans cet exemple, la donnée encodée est interprétée comme suit pour un index:

  xxx: la chaine de caractères à utiliser dans l'index,

  key: la clé de tri,

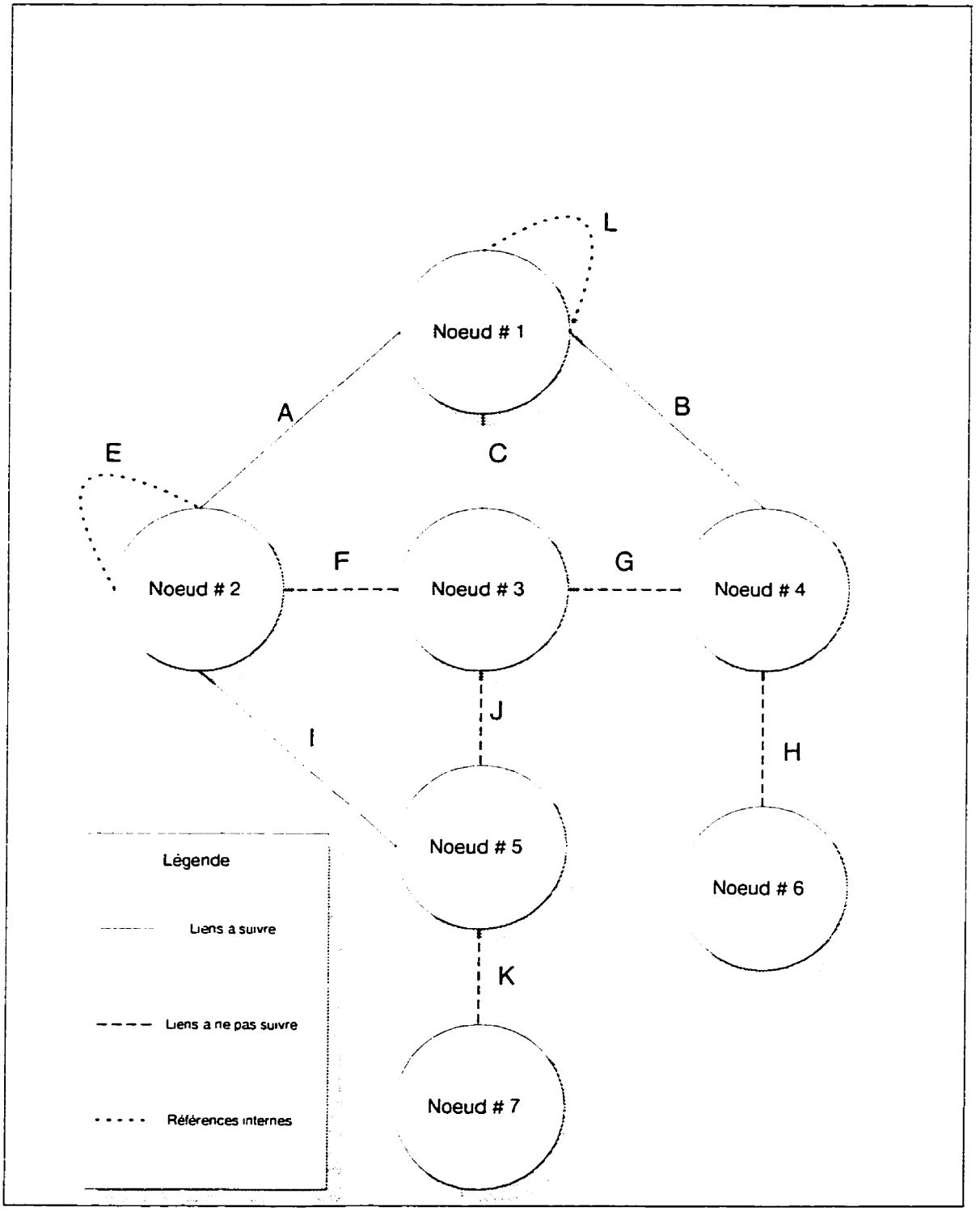


Figure 3.2: Structure non-linéaire

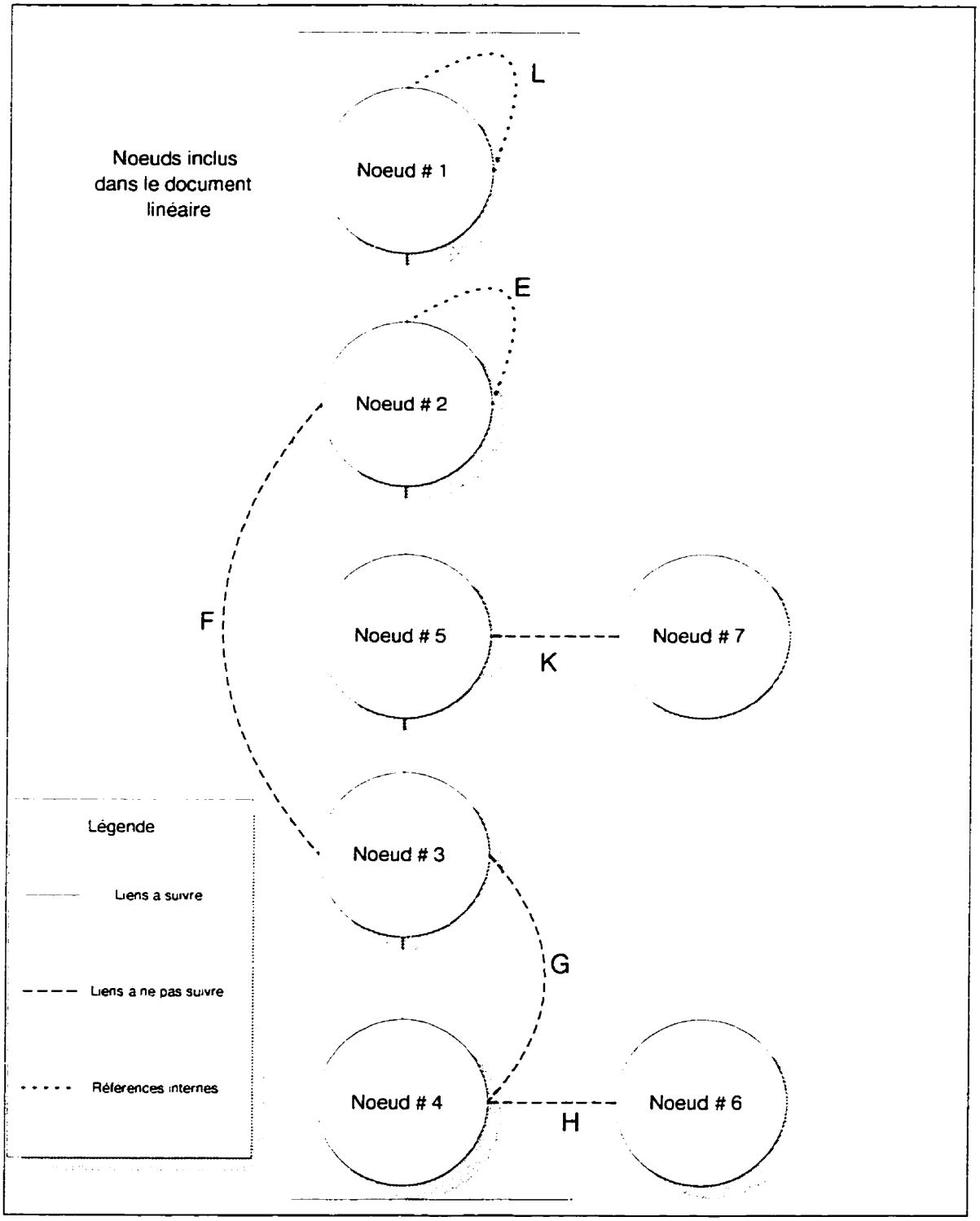


Figure 3.3: Structure linéaire

text : l'identificateur de l'index.

Cette information servira donc à créer l'index d'un site Web.

3.2 Algorithme de linéarisation

L'autre aspect important du système est l'algorithme de linéarisation. Contrairement aux algorithmes de Mike Sharples et James Goodlet (1994), qui reposent sur des heuristiques pour savoir quel liens suivre. notre algorithme repose sur des règles (énoncées plus haut) bien définies pour savoir quels liens parcourir.

Plusieurs alternatives s'offraient à nous pour le développement de l'algorithme de linéarisation. Tout d'abord la recherche en largeur: la recherche en largeur consiste à examiner tous les noeuds se rattachant au noeud en traitement avant de descendre d'un niveau plus bas dans l'arbre (voir figure 3.4). Cette méthode possède le désavantage de ne pas parcourir l'arbre de la même façon qu'un document linéaire, c'est-à-dire qu'elle parcourt d'abord toutes les sections, pour ensuite parcourir toutes les sous-sections (les noeuds d'un niveau plus bas) et ainsi de suite, contrairement à un document linéaire qui parcourt en profondeur une section avant de passer à la prochaine section. Lorsqu'on utilise cette méthode, on doit donc insérer des noeuds dans le milieu de la liste linéaire (des sous-sections qui s'insèrent entre des sections). Ceci a le fâcheux inconvénient d'avoir l'obligation de garder en mémoire toute la liste pour effectuer les insertions. Si le document linéaire est de grande dimension, des problèmes d'espace mémoire peuvent surgir. Cette méthode fût donc rejetée pour cette raison.

L'autre alternative était la recherche en profondeur. La recherche en profondeur consiste à parcourir l'arbre à partir de son noeud d'entrée et de descendre jusqu'au niveau le plus bas et ensuite de remonter au prochain niveau où se trouve un embranchement non visité (voir figure 3.5). Ce mécanisme permet de parcourir l'arbre de la même façon qu'un document linéaire est construit. Cette méthode a l'avantage

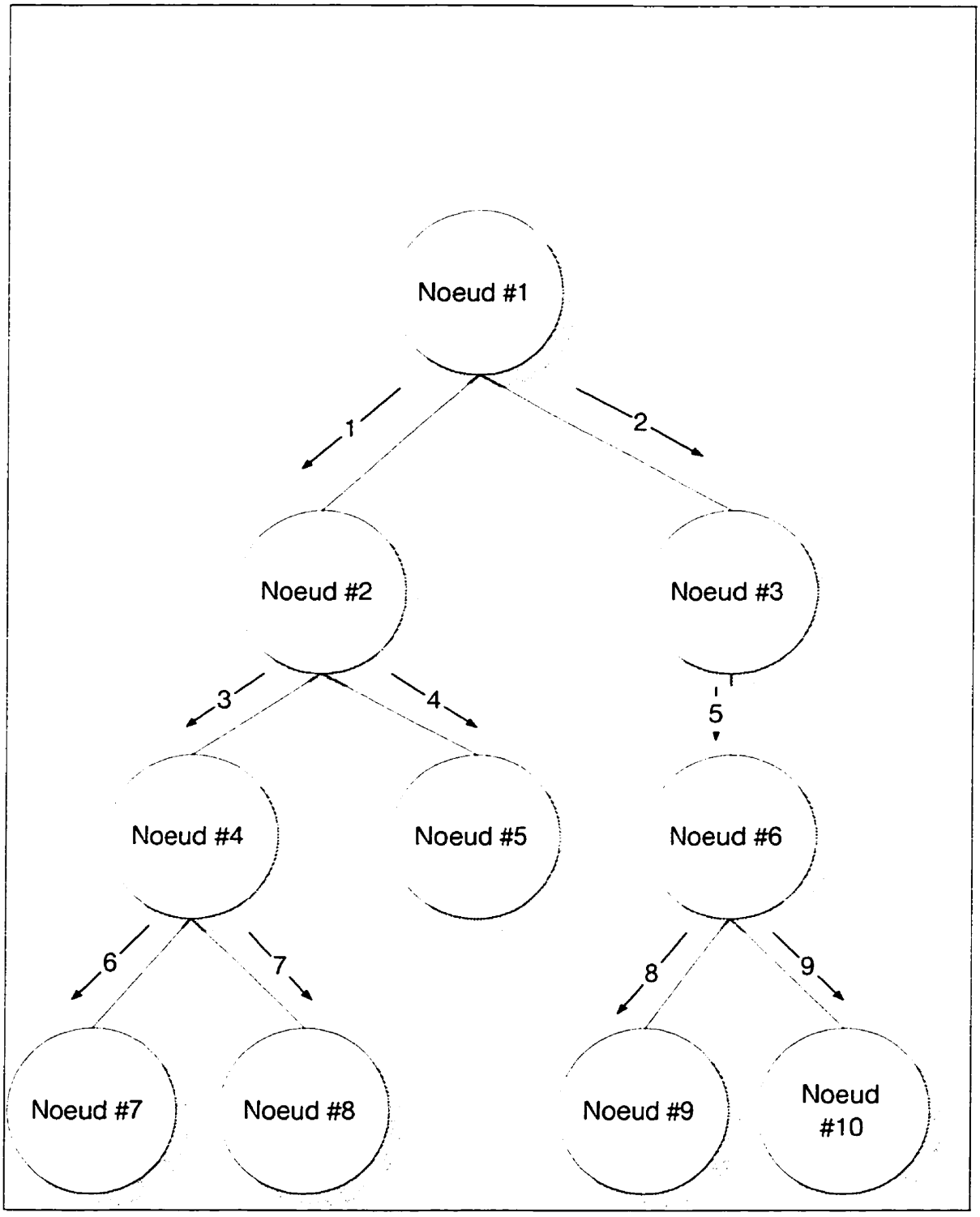


Figure 3.4: Recherche en largeur

de construire la liste sans faire de retour en arrière pour insérer des noeuds. les ajouts de noeuds se font directement en fin de liste. Ceci permet d'écrire le document au fur et à mesure que l'arbre est visité et ainsi de minimiser le nombre de fichiers ouverts simultanément, et ainsi d'utiliser la mémoire efficacement. C'est donc sur cette méthode de recherche que nous avons basé notre algorithme de linéarisation. La définition de l'algorithme est la suivante:

Créer 3 listes simples, VISITÉ, NON-VISITÉ et RÉFÉRENCÉ, initialement vides.

1. Commence au noeud d'entrée du graphe.
2. Place tous les noeuds référés dans la liste NON-VISITÉ.
3. Prend le premier noeud dans la liste NON-VISITÉ. Est-ce un lien à suivre ? Si oui, aller à l'étape 4, sinon aller à l'étape 6.
4. Enlever le noeud de la liste NON-VISITÉ et le mettre dans la liste VISITÉ, ensuite suivre le lien et placer tous les noeuds référés au début de la liste NON-VISITÉ.
5. Est-ce que la liste NON-VISITÉ est vide ? Si oui, aller à l'étape 8, sinon retourner à l'étape 3.
6. Enlever le noeud de la liste NON-VISITÉ et le mettre dans la liste RÉFÉRENCÉ.
7. Est-ce que la liste NON-VISITÉ est vide ? Si oui, va à l'étape 8, sinon, retourne à l'étape 3.
8. Linéarisation completée.

On peut remarquer que cet algorithme utilise 3 listes et que le processus se termine au moment où la liste des noeuds NON-VISITÉ est vide. Ensuite, il ne nous reste

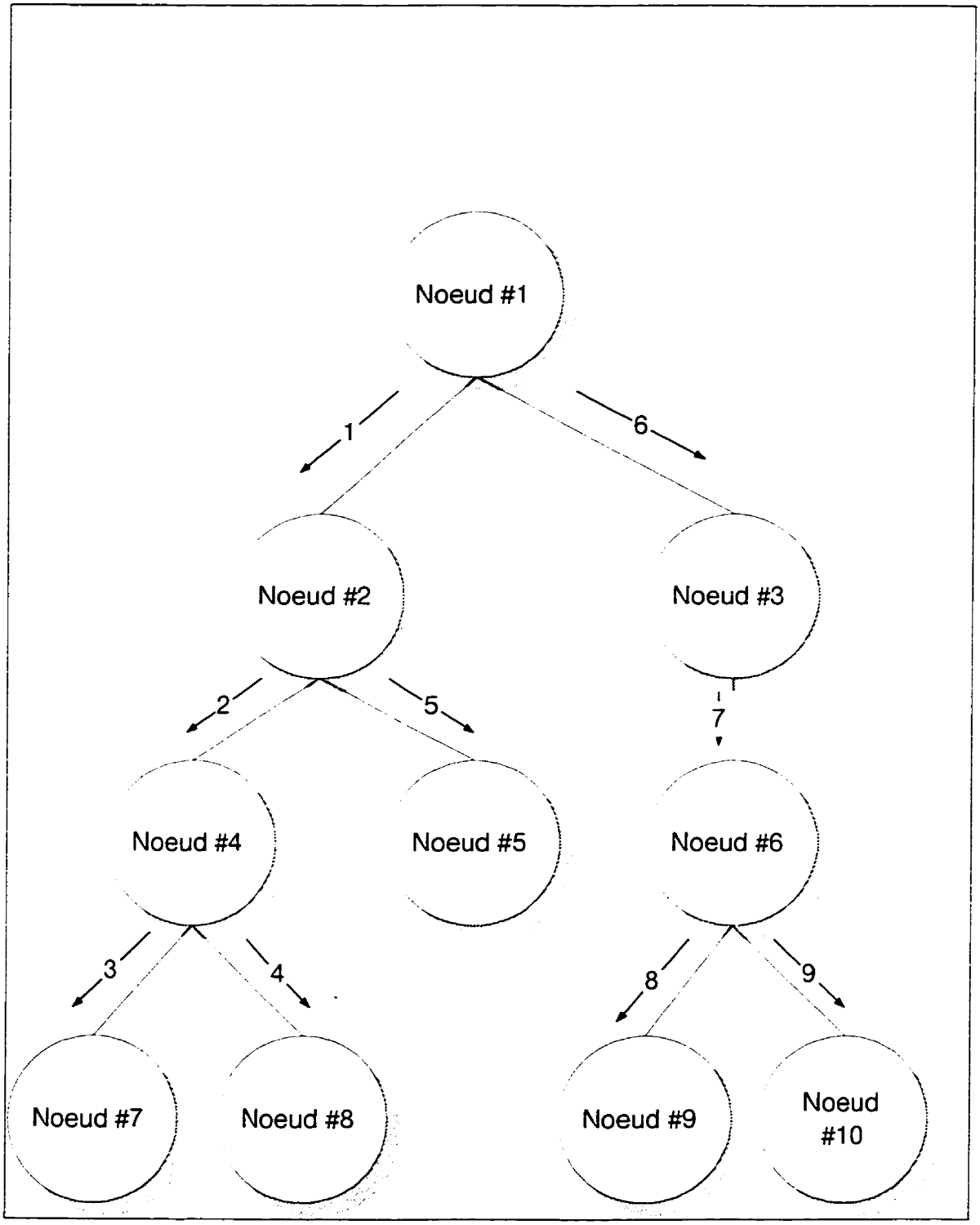


Figure 3.5: Recherche en profondeur

qu'à prendre la liste des noeuds référencés et d'en extaire les informations nécessaires pour construire la liste des références bibliographiques.

3.3 Construction de noeuds dérivés

Lorsque la linéarisation est complétée, il pourrait être intérressant de construire des noeuds dérivés contenant de l'information sur le document linéaire, ou le Web visité. Dans le premier cas, l'adresse dans le document linéaire sera utilisée alors que dans le deuxième cas l'adresse originale devra être utilisée. Ces noeuds dérivés pourront être les suivants :

- Une table des matières contenant toutes les en-têtes du document. Chaque entrée de la liste sera un lien contenant le titre et pointant vers sa position dans le document linéaire ou dans le Web visité. Ceci implique que chaque entête contienne un attribut ID.
- Une liste des figures pourrait être construite similairement à la précédente.
- Une liste des références serait aussi intéressante. Cette liste pourrait contenir toutes les références externes au document linéaire. Dans le langage HTML, ces références sont habituellement sous la forme de URL alors que les documents pour impression préfèrent les références traditionnelles. Pour régler ce problème, le lien indiqué par le URL pourrait pointer directement vers une entrée bibliographique :

  <DIV ID="Here" CLASS=BIBITEM >

  Author,<I>book title</I> , O'Connel, 1995

  </DIV>

  Alors que dans les autres cas où le URL pointera vers un document ordinaire, une entrée bibliographique pourra en être déduite :

  <TITLE> book title </TITLE>

<H1> alternate book title </H1>

⋮

<P>

<DIV CLASS=AUTHOR > Author </DIV>

<P>

<DIV CLASS=BIBITEM.TAIL > O'Connel, 1995, Editor </DIV>

⋮

Si aucune de ces informations n'est présente, alors la référence sera simplement le URL.

3.4 Production de document ASCII

Pour produire un document ASCII, la table des matières doit être ajoutée au début du document alors que la liste des références et l'index sont ajoutés à la fin.
Les références dans la table des matières doivent être remplacées si possible par le numéro de page correspondant de la section.
Les entrées de la liste de références devront avoir un identificateur correspondant à l'ordre de leur première occurence dans le document. Les références externes au document sont ensuite remplacées par l'identificateur correspondant.
Les références internes aux figures, sections, et tables doivent être remplacées par le numéro de la figure, section ou table. Dans certains cas, l'utilisateur voudra le numéro de page plutôt que le numéro de l'item, ceci pourra être accompli par les identificateurs REL=PAGEREF ou CLASS=PAGEREF dans l'élément <A HREF=...>.

3.5 Production de document LaTeX

L'outil LaTeX permet de générer automatiquement la table des matières et l'index, et traiter les items qui réfèrent à des sections internes du document.
La seule particularité où une attention sera requise est dans la génération de la liste

de références qui devra être en partie réalisée par la linéarisation. LaTeX s'occupant de faire les connexions avec la liste à l'aide des commandes \cite et \bibitem.

Discussion

4.1 Analyse du système

Le système ici proposé se distingue de plusieurs façons de ceux exposés dans la recherche bibliographique. D'abord, pour que le système soit effectif, il faut que le document ait été construit dans le but d'une éventuelle linéarisation. L'intégration des règles se fait donc au niveau du document et non au niveau de l'outil lui-même. L'outil ne fait qu'extraire l'information contenue dans le document.

Le système tel que défini présentement ne peut s'appliquer que sur des documents HTML, bien que dans sa définition générale, il pourrait s'appliquer à n'importe quelle base d'information hypertexte.

Une autre de ses limites est lorsque la linéarisation s'effectue sur un document qui n'intègre pas les règles définies par le système. La linéarisation se base alors seulement sur certaines indications provenant de l'usager. Dans notre cas, ces informations se limitent à une expression régulière qui, selon sa correspondance avec le

chemin des noeuds relativement au noeud d'entrée, permet d'inclure ou d'exclure ce noeud. Cette seule information provenant de l'usager ne peut pas être considérée comme une heuristique tel que défini par les systèmes présentés dans la recherche bibliographique. En effet, cette sélection n'influence aucunement la cohérence du texte linéaire final, elle ne fait que limiter le champ d'action de la linéarisation.
Ce système est aussi le seul qui offre la possibilité de générer des noeuds dérivés qui sont directement liés au document hypertexte. Ces noeuds dérivés étant la table des matières, l'index ou la liste des références. Il offre aussi la capacité de présenter les résultats sous plusieurs formats.

4.2 Processus de linéarisation

Le processus de linéarisation est un processus itératif qui débute par la spécification d'un noeud d'entrée par l'usager. Chaque étape du processus sera expliquée en détail par un diagramme et mis en relation avec la définition du système. Un scénario complet sera présenté pour identifier toutes les étapes qui permettent la génération d'un document à partir d'un site WWW.
Le scénario consiste à spécifier un noeud d'entrée d'un site WWW et exiger un document en format ASCII et PostScript (à partir du fichier LaTeX) comprenant une liste des références, un index et une table des matières.
La description du processus général du scénario prédéfini est montré à la figure 4.1. Ce shéma se veut un point de référence global pour chaque étape décrite plus spécifiquement dans les figures suivantes.
Le processus global commence par l'entrée de données par l'usager. Ces données comportent habituellement un noeud d'entrée, des expressions régulières pour limiter la recherche si nécessaire, des options telles que le format exigé (PostScript, LaTeX et ASCII) et la génération de noeuds dérivés (index, table des matières et listes des références). L'initialisation (figure 4.2) permet de valider ces données, de

les initialiser et de créer les structures nécessaires à la linéarisation (listes et fichiers). Ensuite viennent les 4 étapes du processus itératif: analyse du noeud, extraction des références externes, linéarisation des références et sélection du prochain noeud. L'analyse du noeud consiste à identifier la nature du fichier (local, HTTP, FTP, etc), normaliser le chemin de son répertoire, identifier et valider le format de l'information contenue dans le fichier, vérifier si le noeud a déjà été visité pour éviter les références cycliques et émettre un avertissement en conséquence. Pour finir, analyser lexicalement et grammaticalement le noeud pour construire l'arbre binaire contenant l'information.

L'extraction des références externes du noeud et la linéarisation sont des processus itératifs qui permettent d'identifier les références et de les trier en fonction des règles définies. Les noeuds qui doivent être suivis sont insérés dans la liste des noeuds NON-VISITÉ et ceux qui ne le sont pas sont insérés dans la liste des noeuds RÉFÉRENCÉ.

La sélection du prochain noeud se fait lors du parcours de la liste des noeuds NON-VISITÉ en sélectionnant le noeud qui suit immédiatemment le noeud qui vient d'être analysé (recherche en profondeur). Les itérations se terminent lorsque la liste des noeuds NON-VISITÉ est vide.

La linéarisation terminée, une redéfinition des liens est nécessaire comme expliqué précédemment (voir page 28). Cette redéfinition des liens permet d'identifier les références qui sont toujours externes au document linéaire.

Le fichier linéarisé, qui contient une copie des noeuds originaux visités, sert aussi à produire les noeuds dérivés. Le processus de construction des noeuds dérivés se définit de la même manière qu'il s'agisse d'une table des matières ou bien d'un index. D'abord, le fichier d'où l'index ou la table des matières doivent être générés est ouvert. Les en-têtes dans le cas d'une table des matières et les mots indexés, dans le cas d'un index, sont recherchés séquentiellement. Ensuite, si le document duquel

on veut extraire le noeud dérivé a été généré par notre système, on peut extraire l'information relative aux documents sources utilisés pour produire le document et ainsi créer un fichier externe contenant soit l'index, soit la table des matières, pointant vers les documents originaux. Par contre, si le document n'a pas été généré par notre système, aucune information ne peut être déduite et l'index ou la table des matières ne pointera que sur le document cible.

Pour finir, la production de document ASCII, LaTeX et PostScript exige une analyse lexicale et grammaticale pour bien identifier la structure de l'information et ainsi pouvoir la convertir dans le format exigé. La conversion s'effectue à l'aide d'un filtre qui utilise une fonction de correspondance pour convertir chacunes des commandes.

4.3 Implantation

Les règles définies précédemment ont été implantées dans un prototype pour en vérifier la validité et l'applicabilité. La seule exigence pour le prototype fût son aspect modulaire, nécessaire pour la modification ou l'intégration de nouvelles règles dans le futur.

4.3.1 Librairies

L'avantage de la modularité au niveau de la conception est de pouvoir créer le prototype sous forme de librairies, qui elles pourront être réutilisées pour d'autres applications similaires.

Les librairies sont au nombre de 3, chacune étant composée d'un certain nombre de modules regroupés ensemble selon le niveau de fonctionnalité qu'ils apportent au sytème de linéarisation.

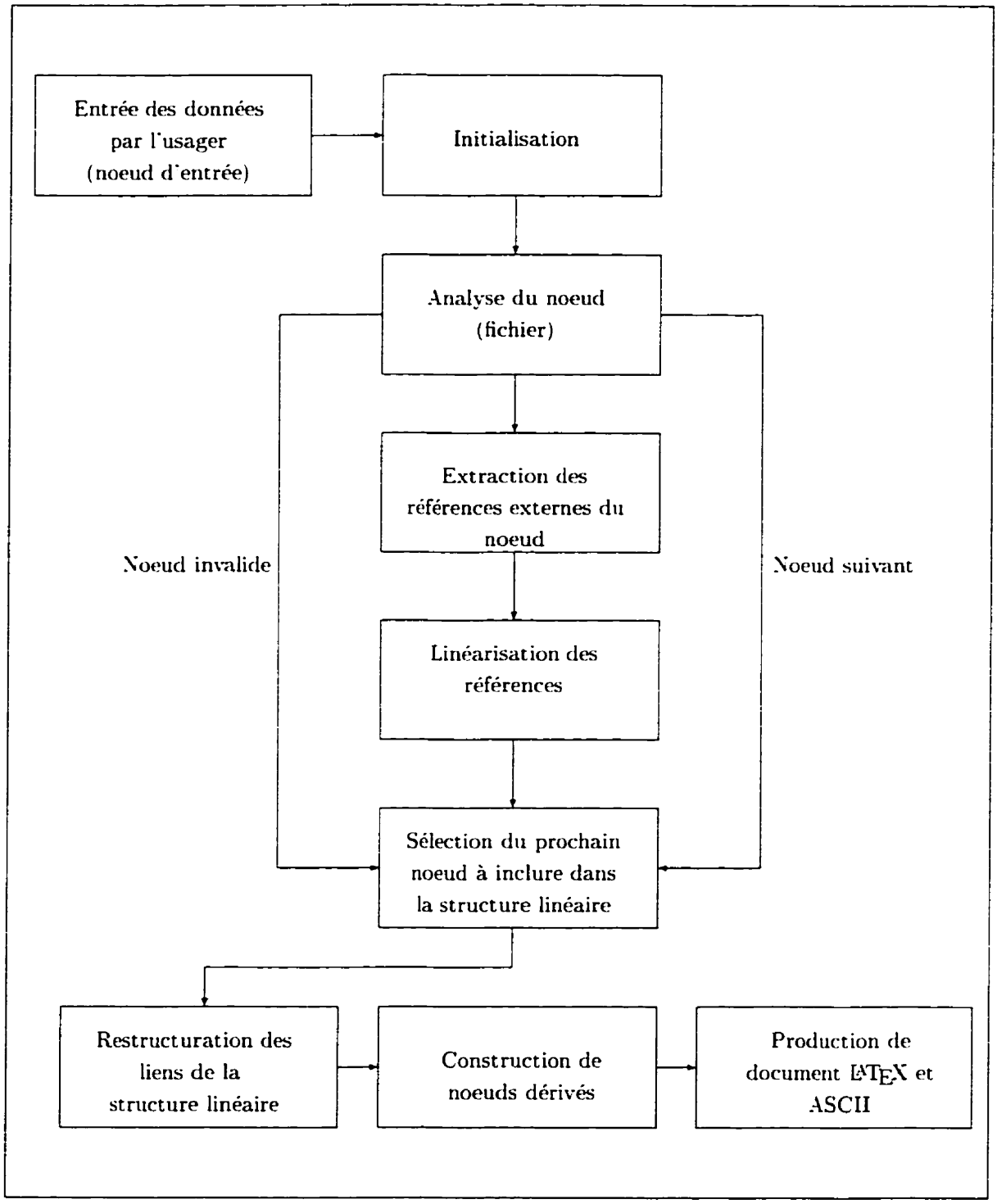


Figure 4.1: Processus général de linéarisation.

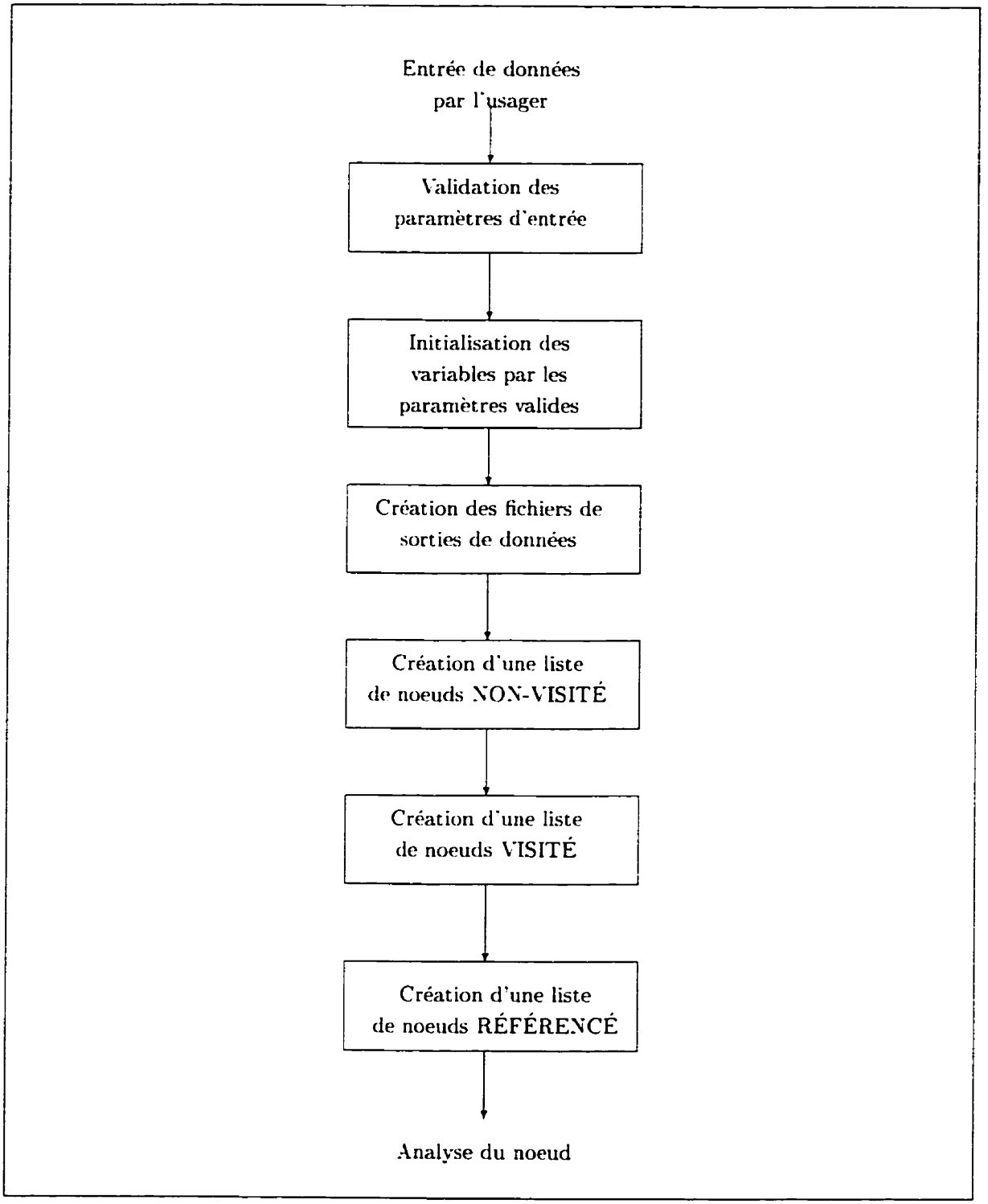


Figure 4.2: Initialisation.

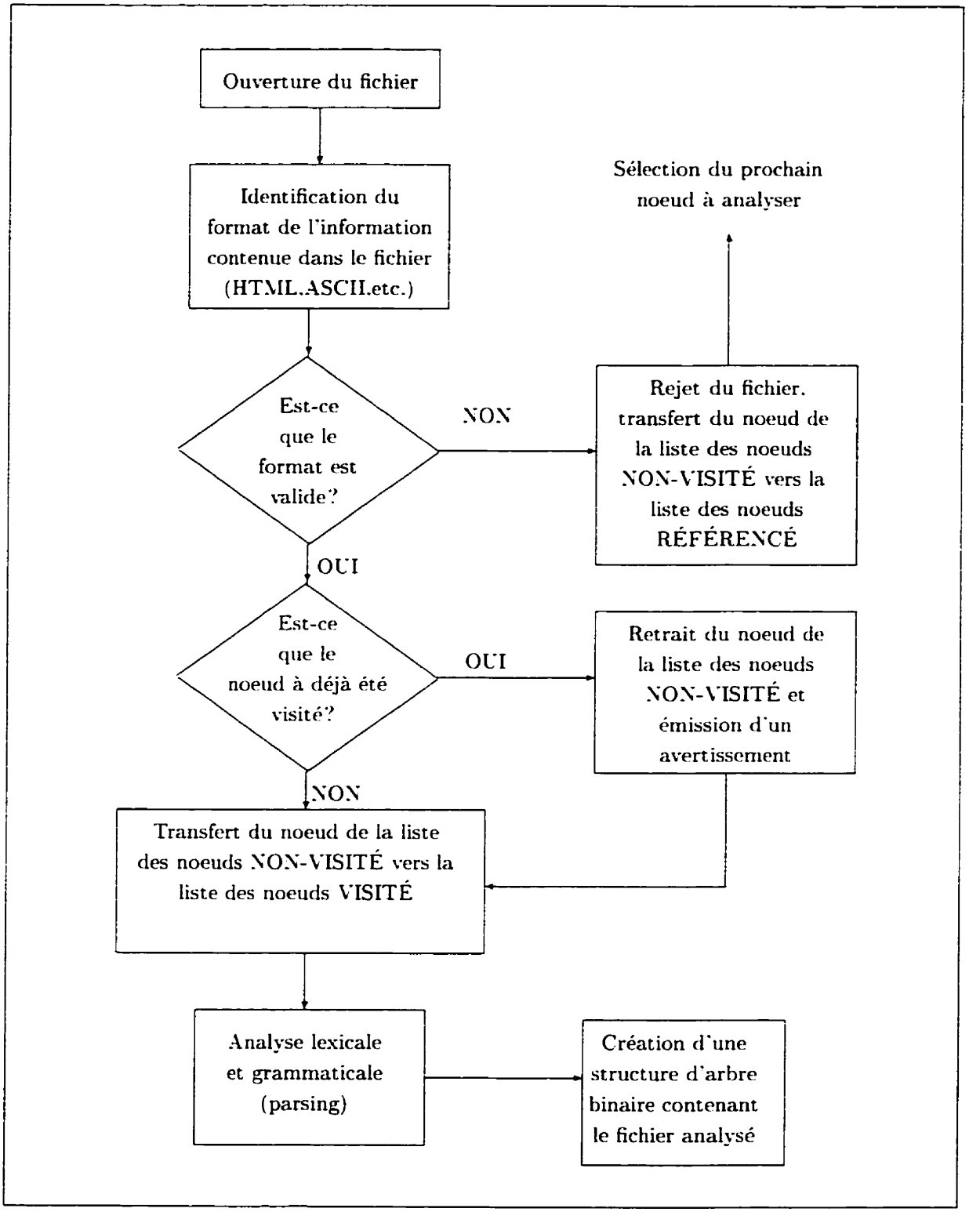


Figure 4.3: Analyse du noeud.

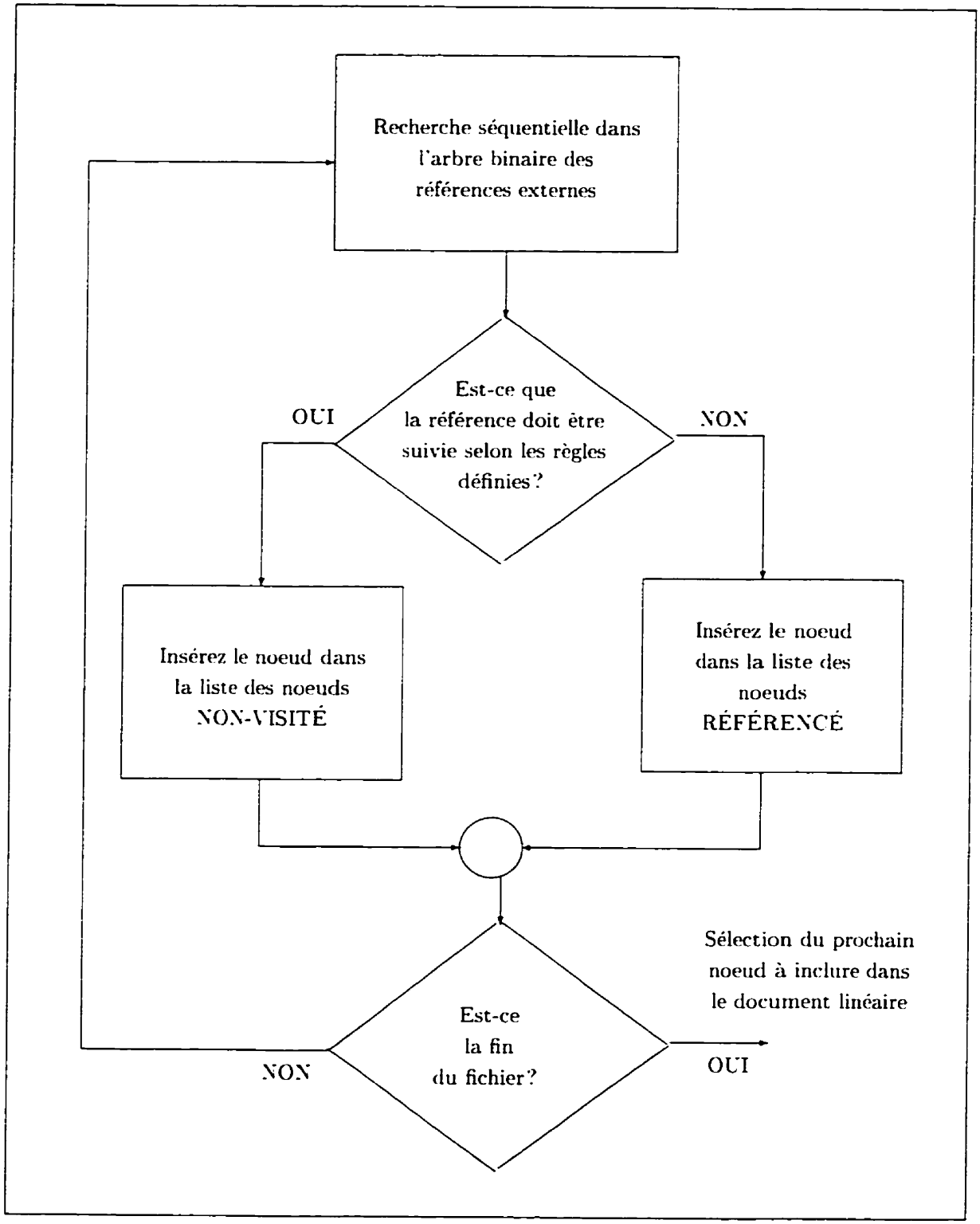


Figure 4.4: Recherche des références externes.

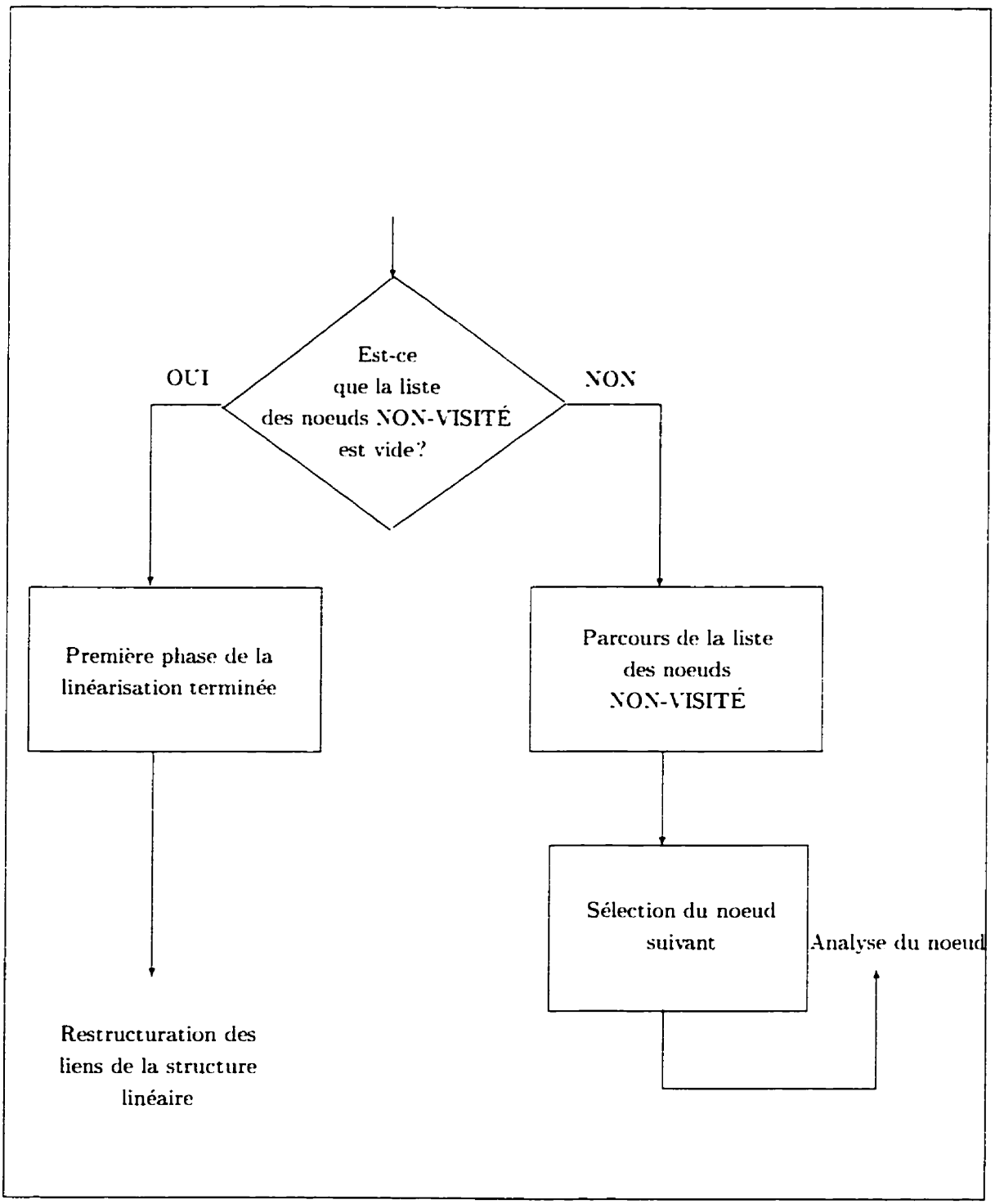


Figure 4.5: Linéarisation.

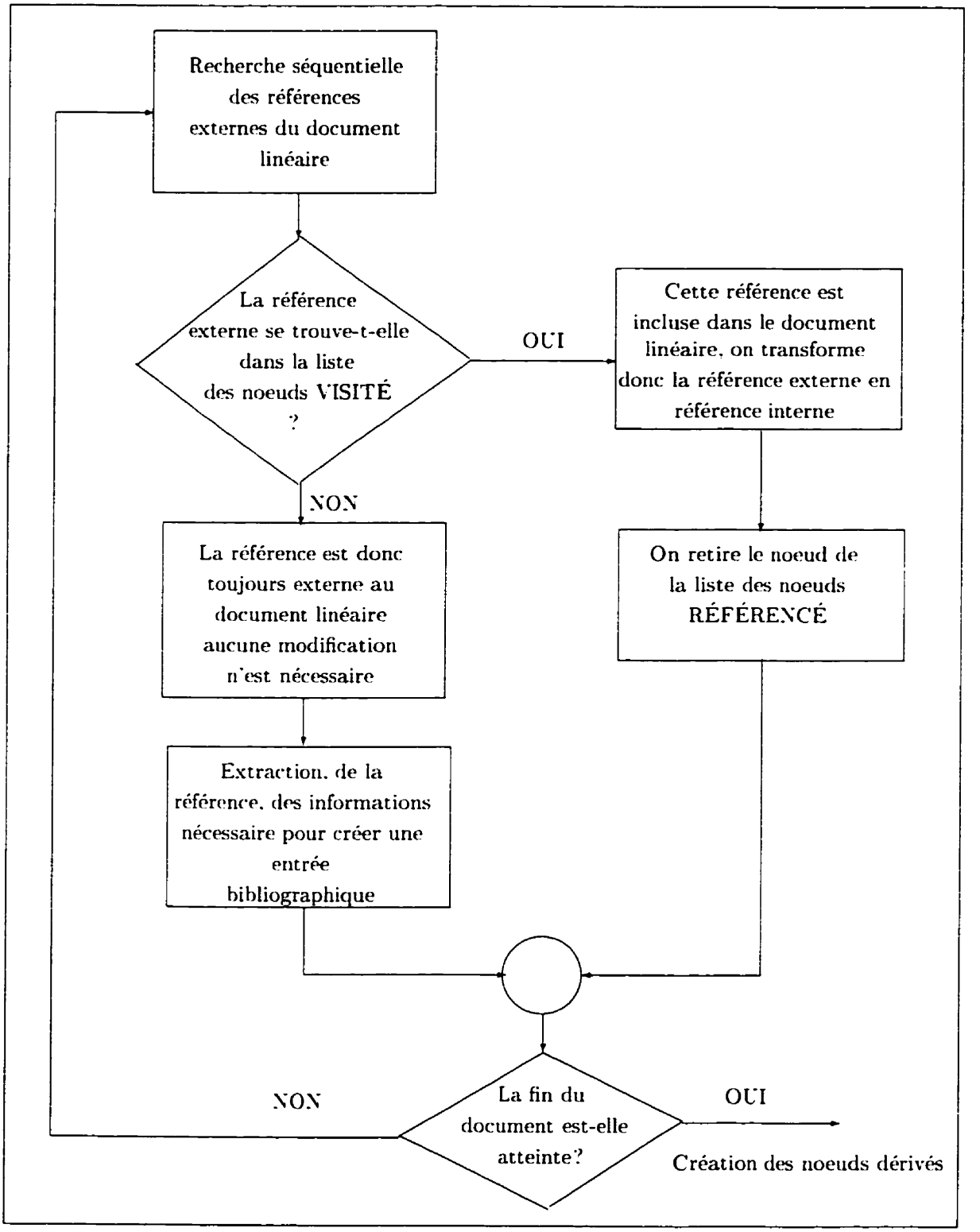


Figure 4.6: Redéfinition des liens.

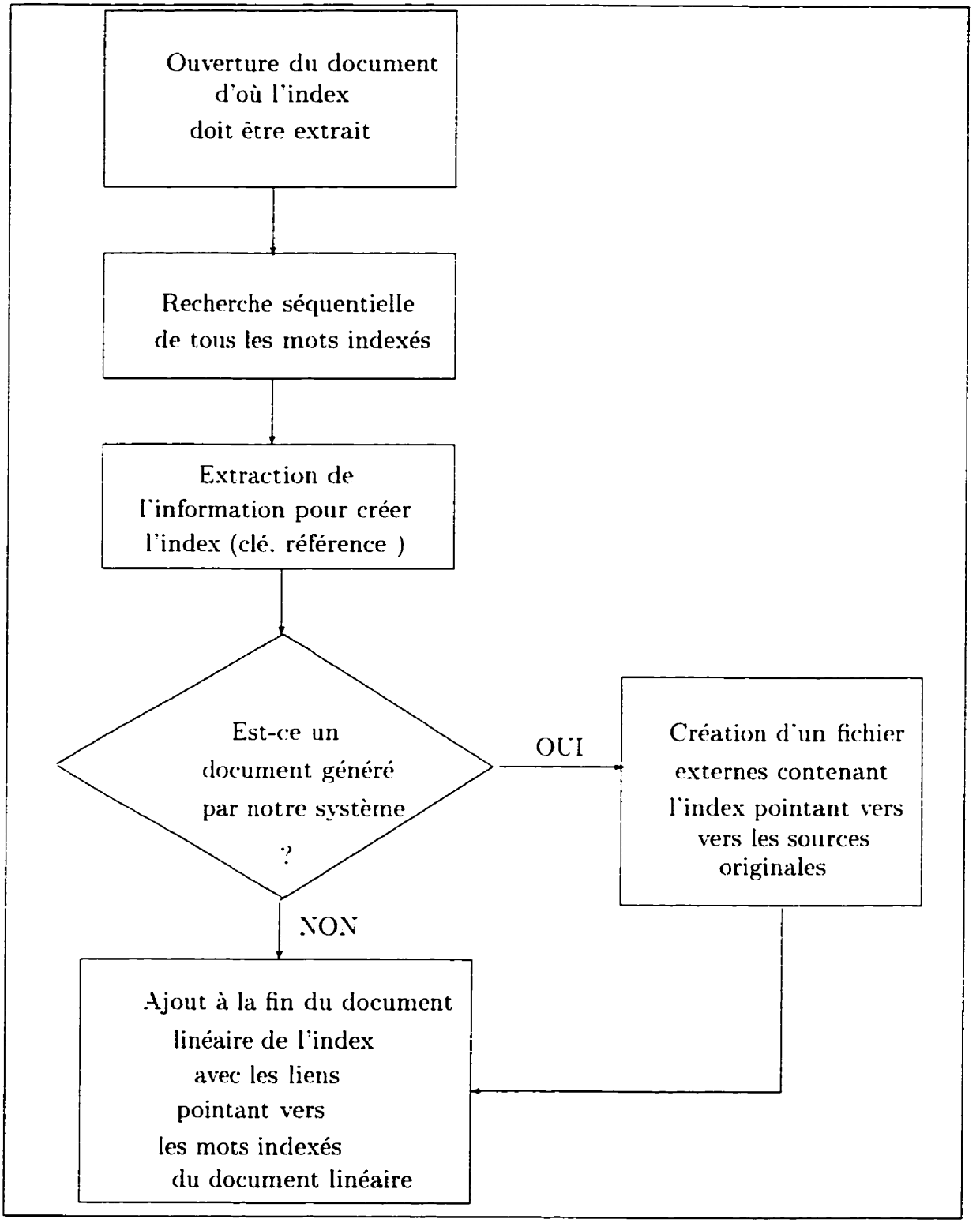


Figure 4.7: Création de l'index.

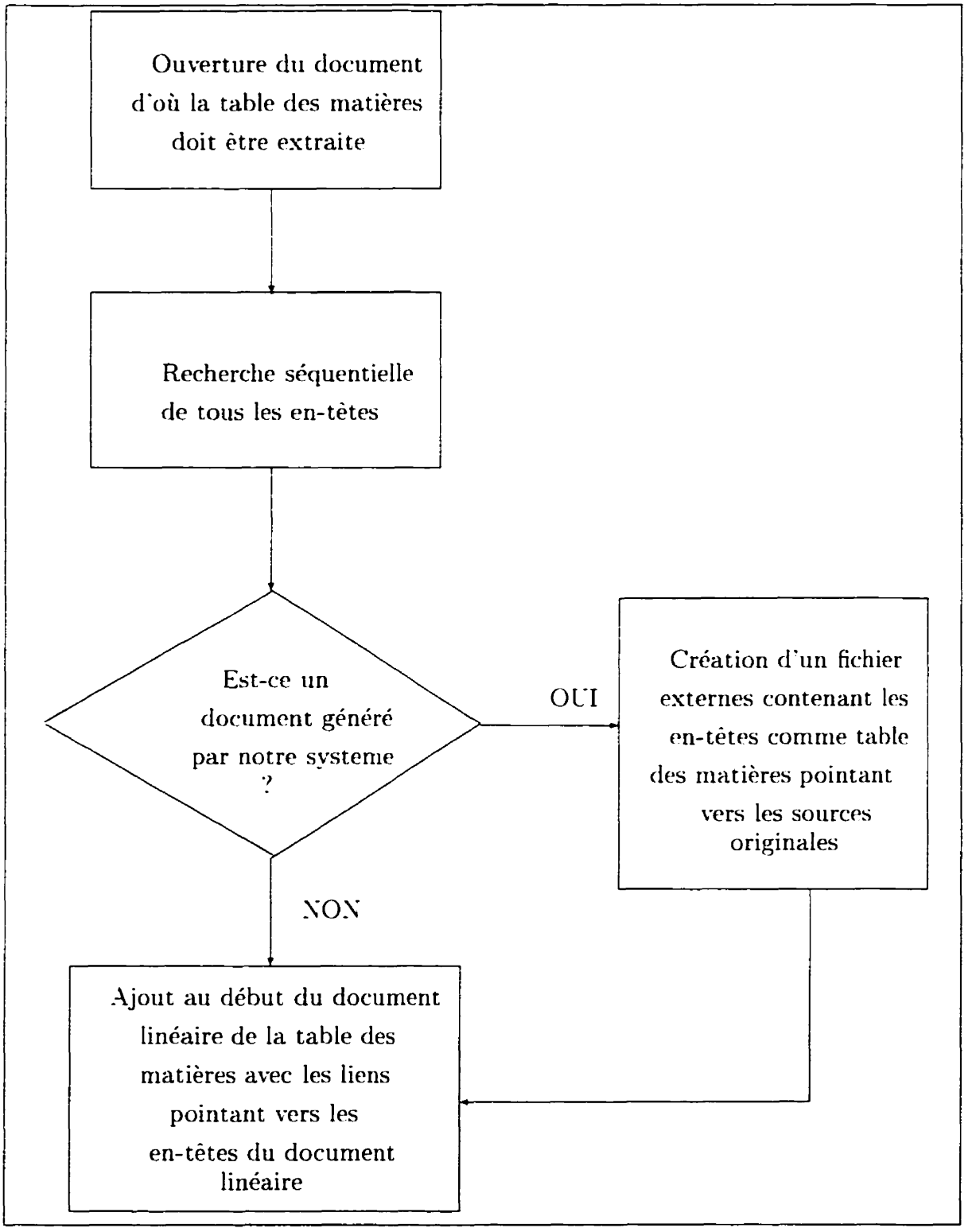


Figure 4.8: Création de la table des matières.

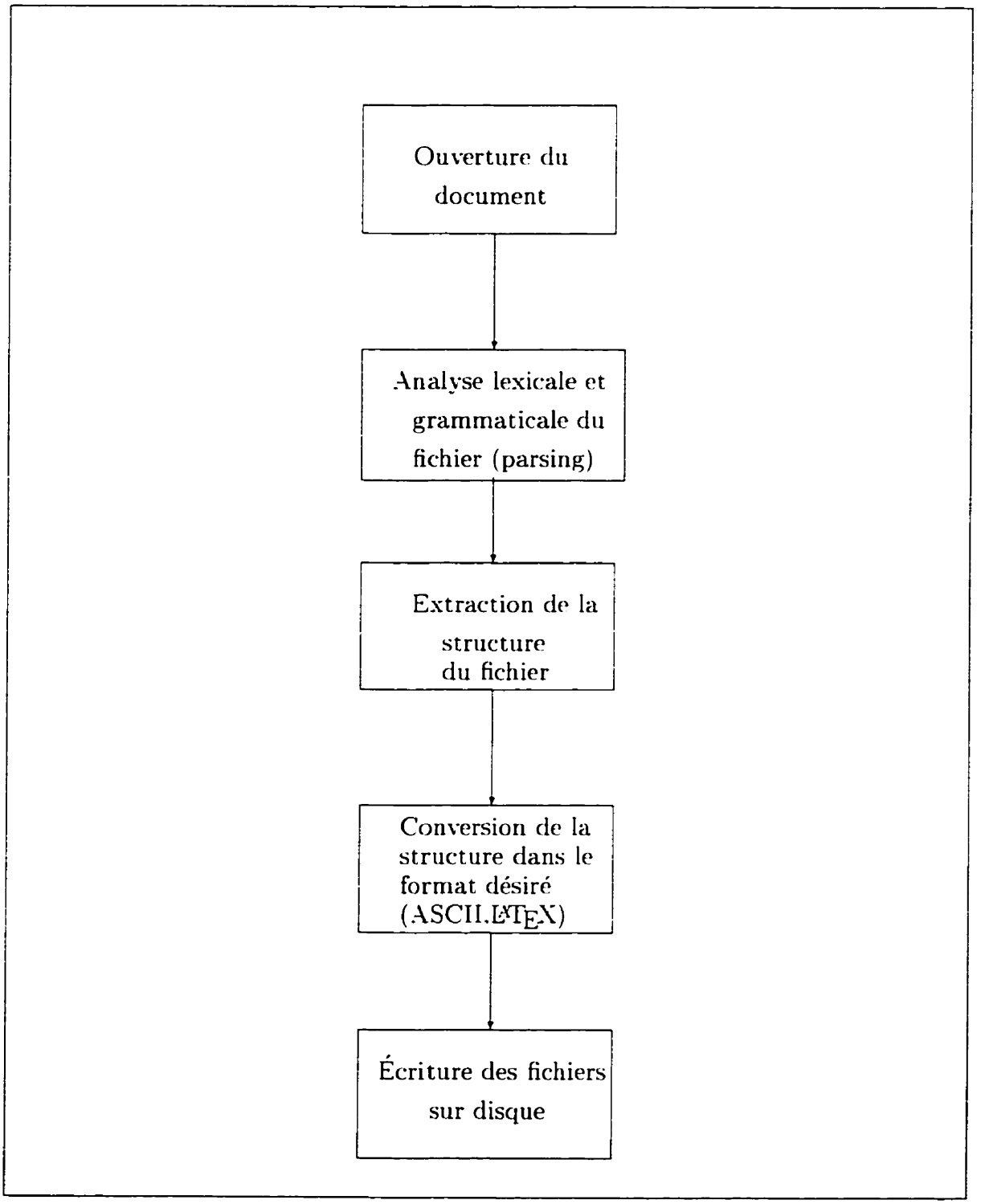


Figure 4.9: Conversion du document vers d'autres formats.

Les libraries sont présentées dans l'énumération suivante; une courte description suit chacun des items:

- Bas niveau (voir page 72)

  Communication Ces modules fournissent les fonctions permettant de communiquer avec un serveur selon 3 types de communication: FTP, Gopher et HTTP.

  Entrées/Sorties Ces modules permettent de gérer les entrées et sorties en relation avec les écritures sur disques ou les ports de communication.

  Fichiers Ce module permet de gérer les ouvertures, fermetures et créations de fichiers.

  Texte Ce module fournit certaines fonctionnalités permettant de travailler sur des chaînes de caractères.

  Reconnaissance de format Ce module permet de reconnaître le format de l'information reçue.

  Outils oeuvrant sur des structures de listes et d'arbres Ces modules fournissent les outils permettant de gérer les structures de données de type linéaire (liste) ou binaire (arbre).

  Parser Ce module permet de lire un document de type HTML.

- Intermédiaire (voir page 92)

  Outils oeuvrant sur des fichiers HTML linéaires ou hiérarchiques Ces modules fournissent les outils permettant de manipuler des documents HTML préalablement mis sous forme de liste ou d'arbre binaire.

  Algorithme de tri (générique) Ce module permet de trier n'importe quels éléments nécessitant un ordre particulier.

Reconnaissance de patron Ce module permet de reconnaître et d'associer des expressions régulières à des chaînes de caractères.

Redéfinition de liens Ce module permet de redéfinir les liens d'un document hypertexte qui vient d'être linéarisé par le processus de linéarisation du système.

- Haut niveau (voir page 97)

  Linéarisateur Ce module permet de fournir toutes les fonctions permettant de linéariser un document hypertexte de type HTML.

  Générateur de fichier ASCII Ce module fournit un filtre permettant de convertir un fichier HTML en fichier ASCII.

  Générateur de fichier LaTeX Ce module fournit un filtre permettant de convertir un fichier HTML en fichier LaTeX.

  Outil pour construire la table des matières Ce module offre la fonctionnalité permettant de construire une table des matières à partir d'un document HTML.

  Outil pour construire l'index Ce module offre la fonctionnalité permettant de construire un index à partir d'un document HTML.

  Outil pour construire la liste de références Ce module offre la fonctionnalité permettant de construire une liste des références à partir d'un document HTML.

Avec cette structure il est donc facile pour un utilisateur qui est peu familier avec le système, d'utiliser les librairies. Par exemple, il peut créer une table des matières d'un fichier sans nécessairement vouloir linéariser un site entier, ou bien utiliser les modules de communication pour une autre application où des transferts de fichiers sont requis. De plus, pour aider l'usager à comprendre la structure des librairies, celles-ci sont fortement documentées et un tutoriel (voir page 69), sur la façon de

les utiliser, les accompagnent. Ce tutoriel se présente sous la forme d'un programme commenté qui permet de numéroter les en-têtes des chapitres, sections et sous-sections qui composent les documents HTML. Le programme est construit sous forme d'étapes qui expliquent quelles librairies sont appelées et leur utilité dans l'application.

Développé à l'aide de ces librairies, le prototype est opérationnel sur la majorité des plate-formes supportés par l'environnement Modula-3 (SPARC, LINUX , WIN32, AIX, ...). Le prototype peut être opéré selon deux modes (voir page 68). Le premier mode est la ligne de commande, qui n'exige pas d'environnement graphique et offre toutes les fonctionnalités décrites précédemment. Le deuxième mode d'opération est par l'interface graphique. Cette interface a été créée à l'aide de l'outil de développement FormsVBT et permet donc de rendre la linéarisation plus conviviale pour l'usager.

4.4 Résultats

Après avoir implanté le prototype, nous avons testé ses performances à l'aide d'un profileur, sur une station Sun SPARC 5 disposant de 32 Megs de RAM, avec comme exécution typique une linéarisation comprenant la construction d'une table des matières, un index, une liste de références et un fichier LaTeX. Les premiers résulats obtenus furent le temps d'exécution (correspondant au temps écoulé) et l'espace mémoire requis en fonction du nombre de noeuds dans le graphe et de la dimension totale des fichiers linéarisés. On peut observer dans la table 1 que, pour un site dont la taille se situerait dans la moyenne (c-à-d $\approx$ 100 noeuds dont la dimension totale est de $\approx$ 100 Ko), le temps d'attente (temps usager) serait de $\approx$ 40 secondes. Le temps requis s'explique par le fait que la modularité fait chuter les performances car certaines tâches peuvent être exécutées plusieurs fois inutilement si elles sont présentes dans plusieurs modules distincts appelés consécutivement. Par exemple,

l'arbre représentant un fichier HTML est traversé à plusieurs reprises pour différentes fonctions (table des matières, index, ...). Une autre raison est le temps perdu par les modules de communication qui font les requêtes aux différents serveurs de fichiers HTML. Ce temps est directement proportionnel à l'achalandage du serveur, il peut donc varier d'une exécution à l'autre.

Le temps d'exécution (figure 4.10) augmente linéairement en fonction du nombre de noeuds linéarisés, cette observation est directement reliée aux performances de l'algorithme de linéarisation d'ordre linéaire aussi.

Une autre observation importante est l'espace mémoire occupé en fonction du nombre de noeuds: on peut remarquer que l'espace mémoire requis par le prototype varie peu ($\approx$ 13 %) (table 1 et figure 4.11) comparativement à celui du nombre de noeuds dans le graphe. Ce résultat nous assure donc que la mémoire est bien gérée et qu'aucune fuite de mémoire ne risque de se produire.

La figure 4.12 et la table 4.2 présente l'utilisation de la mémoire pour un réseau de noeuds ayant une dimension constante (94 Ko) en fonction du ratio octets/noeuds. On peut remarquer que l'utilisation des ressources de la mémoire par le prototype est inversement proportionnelle au ratio octets/noeuds du réseau de noeuds. L'effet d'augmenter la dimension d'un noeud dans le graphe réduit sensiblement la profondeur de l'arbre et par conséquent le nombre de noeuds visités. La dimension de la liste des noeuds visités et celles des références rappetisse si le nombre de noeuds dans le graphe est réduit. Par contre, la contribution de ces listes est inférieure à celle d'un noeud de grande dimension qui doit être analysé et structuré. Ceci peut s'expliquer par l'exemple suivant, prenons deux sites contenant la même information, de dimension totale équivalente, mais dont le premier serait constitué de 5 noeuds de 100 Ko chacun et le deuxième de 20 noeuds de 25 Ko. Si l'on tient compte que les deux sites ont les mêmes références externes (puisqu'ils contiennent la même information), seule la dimension des noeuds et la liste des noeuds

visités différeront d'un site à l'autre. Tel qu'expliqué précédemment, l'algorithme de linéarisation utilise la recherche en profondeur, ce qui a pour effet d'écrire directement le noeud sur le disque après l'avoir analysé et structuré. Ainsi, seulement un noeud occupe la mémoire à la fois. Par contre, la liste des noeuds visités et la liste des références doivent être mémorisées car elles permettent de briser toutes les références cycliques lors de la linéarisation. Ces listes sont constituées de structures de données pour chacun des noeuds visités et référencés. Cette structure de données contient toute l'information relative au noeud visité (nom du fichier, identificateur du noeud, chemin relatif, nature du fichier, protocole de communication, etc.). On peut assumer que la structure de données pour chacun des noeuds est d'approximativement 500 octets. Puisque le même nombre de noeuds est référencés par chacun des deux sites, (ils contiennent la même information), et que les noeuds constituant chacun des sites sont de dimensions égales, on peut calculer l'espace mémoire maximal utilisé par la linéarisation de chacun des sites. Pour ce calcul, on peut utiliser la formule suivante:

Définition 4.1 : Pour obtenir l'espace mémoire maximal utilisé par le processus de linéarisation.

$$D_{tot} \approx D_n + (N_v + N_r) \cdot D_{data}$$

où D_{tot} est la dimension totale de l'espace mémoire utilisé par la linéarisation.
D_n est la dimension moyenne d'un noeud du site,
N_v le nombre de noeuds visités,
N_r le nombre de noeuds référencés,
D_{data} la dimension de la structure de données permettant de mémoriser un noeud visité ou référencé.
En utilisant cette définition, on obtient donc pour le cas #1 (en assumant 5 noeuds référencés):

$$105Ko = 100Ko + (5 + 5) \cdot 0.5Ko$$

Et pour le cas #2 (en assumant toujours 5 noeuds référencés):

$$37,5Ko = 25Ko + (20 + 5) \cdot 0.5Ko$$

On voit que le site #2 composé de plusieurs noeuds de faible dimension prend beaucoup moins d'espace mémoire que le site #1, démontrant ainsi les faits observés. Par contre, cette formule ne donne qu'une approximation de l'espace mémoire puisqu'elle se base sur l'hypothèse que tous les noeuds du site sont de dimension égale ce qui n'est évidemment pas le cas. On peut augmenter la précision de la définition en spécifiant que l'équation doit être utilisée avec la dimension du plus gros noeud visité. Puisque ce noeud ne sera pas nécessairement visité à la fin de la linéarisation, lui ajouter la liste complète des noeuds visités et référencés assure d'obtenir le cas critique d'utilisation des ressources mémoires requises par la linéarisation.

Définition 4.2: Pour obtenir l'espace mémoire critique maximal utilisé par le processus de linéarisation.

$$D_{tot} \leq D_{critique} = D_{max} + (N_v + N_r) \cdot D_{data}$$

où D_{tot} est la dimension totale de l'espace mémoire utilisé par la linéarisation,
$D_{critique}$ est la dimension maximale possible de l'espace mémoire utilisé par la linéarisation,
D_{max} est la dimension du plus gros noeud contenu dans le site,
N_v le nombre de noeuds visités,
N_r le nombre de noeuds référencés,
D_{data} la dimension de la structure de donnée permettant de mémoriser un noeud visité ou référencé.

Les autres résultats furent obtenus à l'aide d'un profileur qui nous indique le temps pris par chacunes des procédures pendant l'exécution, ainsi que le nombre d'appels à cette procédure par le programme. Cette exécution linéarisa 94 fichiers en 43.02 secondes, de temps écoulé. Nous avons aussi utilisé dans nos résultats le temps écoulé, puisque c'est le temps durant lequel l'usager devra attendre pour obtenir le document linéaire. Le temps CPU indique les performances du système lors des opérations de linéarisation mais n'indique pas le temps d'attente pour créer les liens de communication, contrairement au temps écoulé. Tel que vu précédemment, ces temps d'attente ne sont pas négligeables lorsque le réseau de communication est chargé. Si on met de côté les procédures coûteuses en temps mais auxquelles nous n'avons pas accès (allocation de mémoire, ramasse-miette, *mapping* de la mémoire), on peut identifier (sur la table 3) que les modules critiques sont surtout ceux qui traitent le texte et qui font les appels sytèmes pour les modules de communication.

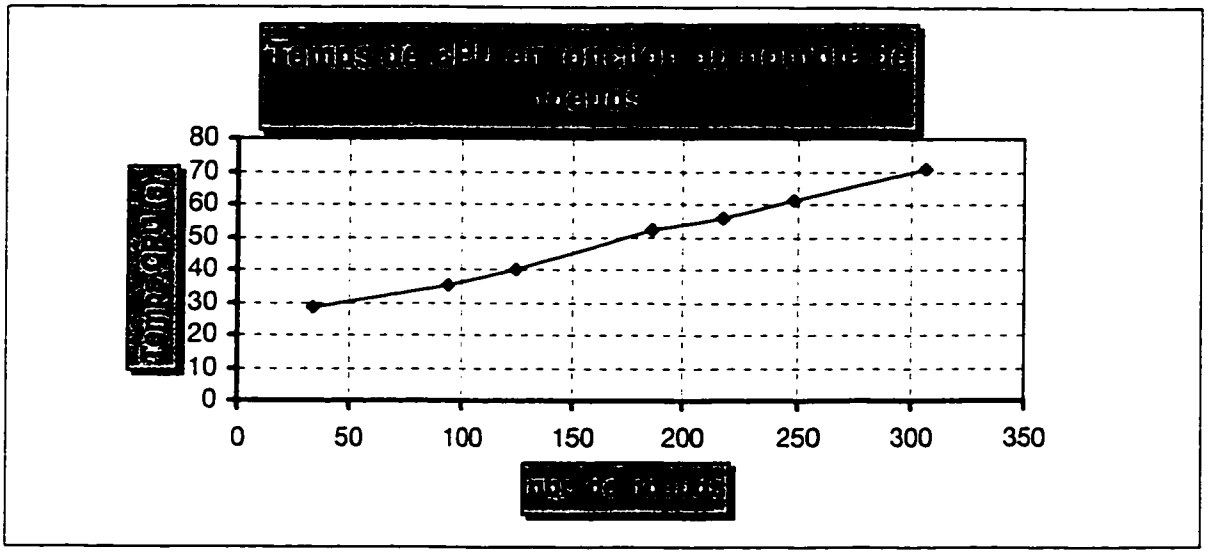


Figure 4.10: Temps de CPU en fonction du nombre de noeuds.

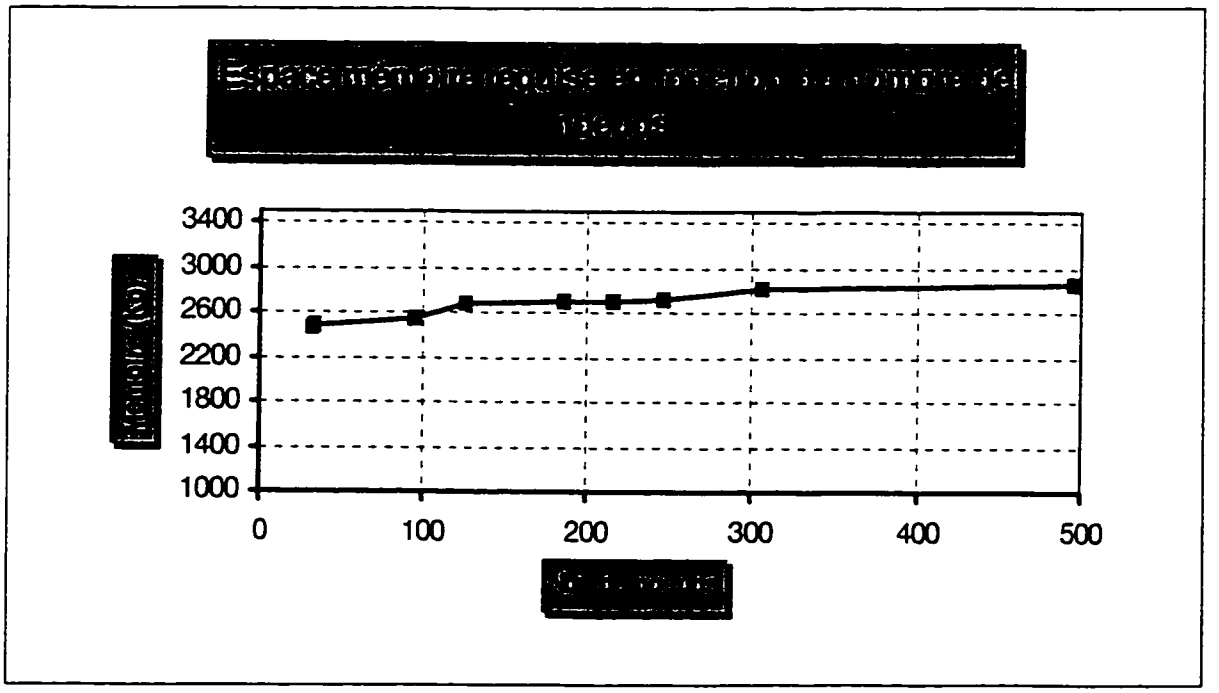


Figure 4.11: Espace mémoire requise en fonction du nombre de noeuds.

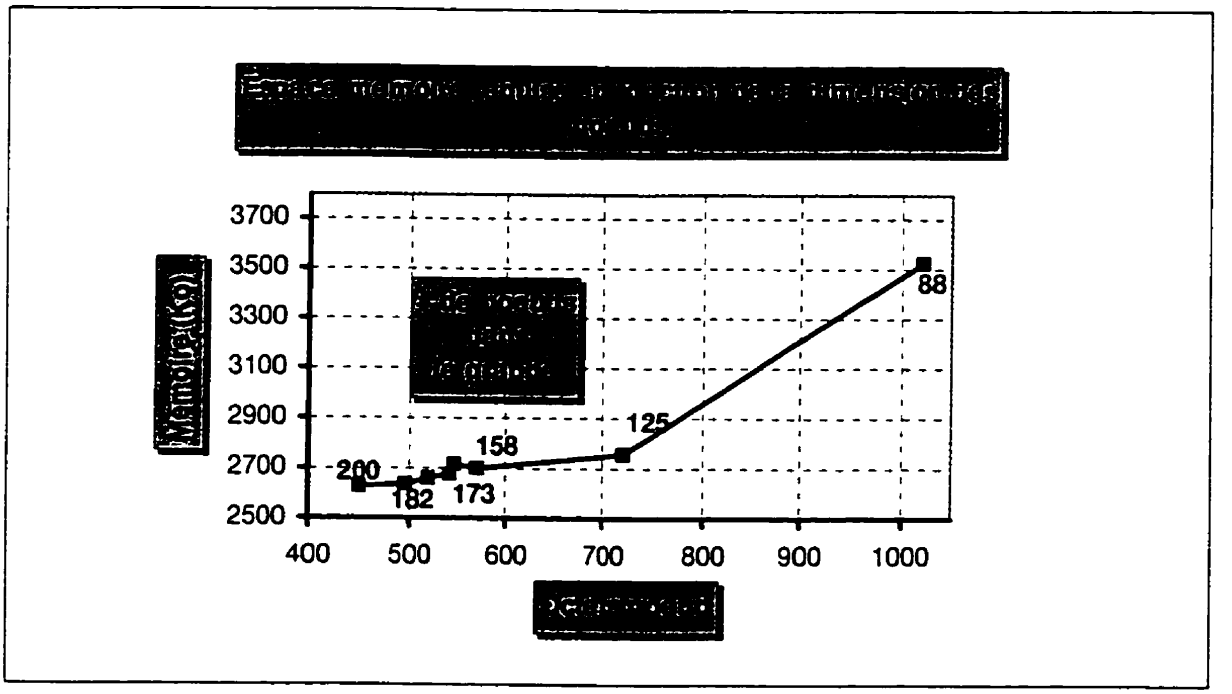


Figure 4.12: Espace mémoire requise en fonction de la dimension des noeuds.

Tableau 4.1: Tests de performance pour linéariser un réseau de noeuds HTML.

Nbr. de noeuds	temps (s) usager	temps (s) CPU	dimension	
			des fichiers (octets)	espace mémoire (koctets)
34	31.84	28.31	34687	2494
94	38.62	35.11	53654	2536
125	45.84	40.81	90132	2685
186	71.89	52.72	101866	2699
217	76.33	56.75	107630	2703
248	82.66	61.17	128856	2709
306	110.84	71.03	166397	2816

Tableau 4.2: Mémoire utilisée en fonction du ratio octets/noeud.

Nbr. de noeuds dans le graphe	ratio octets/noeud	Espace mémoire (koctets)
200	450.0	2632
182	496.0	2635
173	520.0	2661
166	543.4	2679
165	547.6	2714
158	570.8	2697
125	721.5	2758
88	1020.2	3526

Tableau 4.3: "Profilage" du prototype pour un graphe contenant 94 noeuds HTML.

Modules/ Fonctions	temps	temps (sec)	appels
Text/Cat	3.4	1.48	224697
syscall	2.9	1.25	6
HTMLTree/SearchEndTag	2.4	1.03	2019
Text/GetChar	1.3	0.56	21297
Text/Equal	1.2	0.52	22123
HTMLParser/GetField	1.2	0.51	5196
HTMLParser/GetTag	1.1	0.48	6183
Text/Length	0.8	0.36	623285
HTMLList/ResetList	0.8	0.35	634298
Text/FromChar	0.7	0.31	224697

Conclusion

Ce mémoire présente un système comprenant une série de règles permettant de linéariser un document hypertexte de type HTML. Ces règles pourront être améliorées dans le futur pour pouvoir limiter les indications provenant de l'usager au strict nécessaire pour effectuer la linéarisation. Dans ce cas, certaines heuristiques pourront être envisagées pour choisir quels liens suivrent dans le graphe de noeuds. Dans notre cas, une des limites de notre système est la faiblesse de ses heuristiques lorsque le système linéarise un document qui n'a pas auparavant intégré les règles.

Par contre, un de ses avantages est sa flexiblité à être modifiée pour linéariser des documents hypertextes autres que ceux de type HTML.

La structure des librairies sur laquelle est construit notre système permet facilement toute forme de modification visant l'amélioration ou la conversion du sytème vers d'autres formats. Sa conception orientée objet permet facilement la réutilisation de

ses librairies dans le but de développer ou d'améliorer de nouvelles applications relatives à ce domaine.

L'analyse des performances du prototype nous a permis de déterminer ses modules critiques et ses goulots d'étranglement. Ceux-ci pourront recevoir une attention particulière pour rendre le prototype plus performant dans le futur.

La suite du projet pourrait donc être de développer et d'améliorer certains modules pour rendre la linéarisation plus performante quant au temps requis, et développer des heuristiques qui permettront de limiter au strict nécessaire les indications provenant de l'usager. Toutes ces améliorations seraient nécessaires à un logiciel qui se veut un outil convivial et simple d'utilisation pour les usagers d'Internet.

Références

ANKLESARIA, F., MCCAHILL, M., LINDNER, P., JONSHON, D., TORREY, D., ALBERTI, B. (March 1993). The Internet Gopher Protocol: PrA distributed document search and retrieval protocol. RFC 1436, Internet Draft CERN.

BENCH-CAPON, T.J.M., DUNNE, R.J., STANIFORD, G. (1992). Linearising Hypertext through Target Graph Specification. Database and Expert Applications, 3, 173-178.

BENCH-CAPON, T.J.M., DUNNE, R.J., STANIFORD, G. (1993). Linearisation Shemata for Hypertext. Database and Expert Applications, 697-708.

BERNERS-LEE, T. (June 1993). Universal Resource Identifiers in WWW: A unifying syntax for the expression of names and adresses of objects on the network as used in the WWW. RFC 1630, Internet Draft CERN.

BERNERS-LEE, T. (July 1993). Hypertext Transfer Protocol. Internet Draft CERN.

BERNERS-LEE, T., CONNOLLY, D. (July 1993). Hypertext Markup Language. Internet Draft CERN.

BERNERS-LEE, T., CONNOLLY D. (July 1993). Introduction to HTML 3.0. Internet Draft CERN.

BERNERS-LEE, T., MASINTER, L., MCCAHILL, M. (February 1993). Uniform Resource Locator (URL). RFC 1738, Internet Draft CERN.

BERNSTEIN, M. (1988). The bookmarks and the compass: Orientation tools for Hypertext users. ACM SIGOIS Bulletin, 9, 34-45.

BIRELL, A., NELSON, G., OWICKI, S., WOBBER, E. (1994). Network Objects. DEC SRC Report.

BOYLE, C., ENCARNATION, A.O. (1994). Metadoc: An Adaptative Hypertext Reading System. User Modeling and User-Adapated Interaction, 4, 1-19.

BROWN, M.H., MEEHAN, J.R. (1993). vbtkit: A toolkit for Trestle Reference Manual. DEC SRC Report.

BROWN, M.H., MEEHAN, J.R. (1993). The FormsVBT Reference Manual. DEC SRC Report.

CONKLIN, J.J. (September 1987). Hypertext - An introduction and survey. Computer, 17-41.

CONKLIN, J., BEGEMAN, M. L. (1988). gIBIS: a hypertext tool for explanatory policy discussion. ACM Transactions on Office Information Systems, 6, 303-331.

DAVIS, F., KAHLE, B., MORRIS, H., SALEM, J., SHEN, T., WANG, R., SUI, J., GRINBAUM, M. (April 1990). WAIS Interface Protocol Prototype Functionnal Specification. Thinking Machines Corporation.

GEORGE, S. (1992). Approach to linearising hypertext. Ph.D Dissertation, Dept. of Comp. Sci., Univer. of Liverpool.

GOLDFORBS, C. F. (1990). The SGML Handbook. Oxford University Press, New York.

HARBINSON, S. (1992). Modula-3. Prentice Hall.

KANTOR, B., LAPSLEY, P. (February 1986). Network News Transfert Protocol: A proposed standard for the stream-based transmission of news. RFC 977, Internet Draft.

KNUTH, D.E. (1994). The TexBook. Addison-Wesley.

LAMPORT, L. (1994). A Document Preparation System Latex, user's guide and reference manual. Addison-Wesley.

MANASSE, M. S., NELSON, G. (1991). Trestle Reference Manual. DEC SRC Report.

NORDIN, B., BARNARD, D.T., MACLEOD, I.A. (1993). A review of the Standard Generalized Markup Language (SGML). Computer Standards & Interfaces, 15, 5-19.

PARRY, I.(1992). A document modification system. M.Sc Dissertation. Dept. of Comp. Sci., University of Liverpool.

POSTEL, J., REYNOLDS, J. K. (October 1985). File Transfer Protocol (FTP). RFC 959, Internet Draft.

SAMS DEVELOPMENT TEAM (1994). UNIX Unleashed. Sams Publishing.

SCHNASE, J., LEGGET, J., KACMAR, C., BOYLE, C. (1988). A comparison of hypertext systems. TAMU Technical reports 88-017, Dept. of computer science, Texas A&M University, College Station.

SEDGEWICK, R. (1994). Algorithm in Modula-3. Addison-Wesley.

SHARPLES, M., GOODLET, J. (1994). A comparison of algorithms for hypertext notes network linearization. Int. J. Human-Computer Studies, 40, 727-752.

STOTTS, P.D., FURUTA, R. (1988). Adding browsing semantics to the hypertext model. Proc. ACM Conference on Documents Processing Systems, 43-50.

SUBBOTIN, M.M. (1991). Hypertext systems with algorithmic navigation. Proc. of the Int. Colloquium New Information Technology, 8-10.

SUBBOTIN, M., SUBBOTIN, D. (1992). INTELTEXT: Producing Coherent Linear Texts While Navigating in Large Non-Hierarchical Hypertexts. Proceeding of the 2nd East-West Int. Conference on Human-Computer Interaction, 281-289.

Annexe I: Manuel d'opération du linéarisateur

Web Linearizator

By Martin Gagnon

You can perform a linearization directly from the command line (the syntax and availables options are below) or trough the graphical user interface (GUI)

GUI

To call the GUI, type the following comand

linear

With the GUI, a Help Button will provide all the information useful to execute a linearization

Command line

The options available on the command line are :

```
    -TeX : Generate a latex file
    -ASCII : Generate an ASCII file
    -PS : Generate a PostScipt file ( and by the way a Latex file )
    -ToC : Build a table of content
    -LoF : Build a list of figures
    -LoT : Build a list of tables
    -LRef : Build a list of references
    -Index : Build an index
    -DEBUG : Print some information for debugging
    -o a.out : specify a.out as output file
    -patternDiag *.jpg jpgtops : specify a filter for files
 matching the pattern
    -Ind : assume that no REL attribute is set in links
    -prune pattern : specify links not to follow
    -include pattern : specify links to follow
```

Here is an example :

linear -PS -TeX -ToC -Index -o a.out ../TEST/index.html

The path can be absolute or relative

Each unreacheable host will be printed

A message is printed at the beginning of each phase. Building the reference list may take a while.

BUGS

When some references cannot be accessed, a text window may pop up. The linearization proceeds neverthless.

Annexe II: Tutorial

An example of how to use the Modula-3 HTML library

Written by Martin Gagnon

This program renumbers each header of the HTML document. It will show step by step how to use the WWW HTML library, and which package to include in the program. Each time an object from the package is created it will be explained why the application needs it.

THE IMPORT AND EXPORT

```
MODULE Main;
IMPORT HTMLParser,HTMLList,HTMLFile,Params;
IMPORT HTMLTree,HTMLStream,HTMLLinear,Text;
IMPORT Pathname,HTMLCommLine,Wr,Stdio,FileWr;
IMPORT HTMLToolTree,Fmt;
```

TYPE

```
TYPE
   Deepness = ARRAY[1..6] OF CARDINAL;
 (* array of 6 cardinals recording the depth of each header *)
```

GENERAL VARIABLES

```
VAR
    i : CARDINAL := 1; (* increment to get the paramaters *)
    fileIn : Pathname.T; (* the name of the input file *)
    outFile : Pathname.T; (* the name of the output file *)
    file : HTMLFile.T; (* object that hold all the information on the
  input file *)
    input : HTMLStream.In; (* input streamer for the input file *)
    parser: HTMLParser.T;  (* a parser to parse the input file *)
    tokenList :HTMLList.T;  (* a list to hold the tokens *)
    tokenTree : HTMLTree.TokenTree; (* a tokentree to build
the tokens tree *)
    output : HTMLStream.Out; (* an output streamer for the output
file *)
    deepInit := Deepness{0,0,0,0,0,0}; (* initialize
                                          the depth of each
     header at one *)
BEGIN
```

Get the input file name and by default set the name of the output file to the same name.

```
    fileIn := Params.Get(Params.Count -1);
    outFile := fileIn;
```

here we get the option set on the command line

```
    WHILE i < (Params.Count - 1) DO
CASE HTMLCommLine.ParseComm(Params.Get(i)) OF
|HTMLCommLine.CommLine.FOut => INC(i);
       outFile := Params.Get(i);
       (* the ouptut file will have
  automatically the .html
  extension *)
       IF Text.Equal(Pathname.LastExt(
     outFile),"")
       THEN
  outFile := outFile & ".html";
       ELSE
  outFile := Pathname.ReplaceExt(
       outFile,"html");
       END; (* IF *)
ELSE
   Wr.PutText(Stdio.stderr,"Invalid option : "
                              & Params.Get(i) & "\n");
END; (* CASE *)
INC(i);
    END; (* WHILE *)
```

Now that we have all the parameters, we start the application. First we need to initialize the input file

```
    file := NEW(HTMLFile.T).init(fileIn);
```

We create an object HTMLFile.T that extracts all the information from the input file name and stores it. For example:

http: //www.vlsi.polymtl.ca: 80/m3/Bugs.html

will be stored as

```
filenature = http
host = www.vlsi.polymlt.ca
port = 80
name = /m3/Bugs.html
```

Then, an input stream is opened to get the data from the file

```
    input := NEW(HTMLStream.In).init(file);
```

An object is instantiated to provide a reader on the input file. This reader may be built on a TCP connection, if it is a remote file. In any case close() method will close the file and free all related ressources.

If the input file is reached through a TCP connection, the connection may fail. (unknow host, unreachable host , etc ...).

If this happens, the init() method returns NIL , which has to be checked.

```
IF (input # NIL) THEN
```

It succeeded, the file may be parsed.

```
output := NEW(HTMLStream.Out).init(FileWr.Open(outFile));
parser := NEW(HTMLParser.T).init(input.getReader());
```

this initializes a HTMLParser.T object with the reader of the input file. Then, the getTokenList() method produces a list of tokens.

```
tokenList := parser.getTokenList();
```

To renumber the headers, it is more convenient to operate on a hierarchical tree representing the nested sections. To build the sections a HTMLTree.TokenTree object (derived from a HTMLTree.Tree object) is created and initialized.

```
tokenTree := NEW(HTMLTree.TokenTree).init(tokenList);
```

the recursive procedure TraverseTreeAndNumerotation will traverse the tree and preform the header numbering. The final step is to print the new tree.

```
TraverseTreeAndNumerotation(tokenTree.getHead(),deepInit);
```

print the tree on the output stream

```
output.putTree(tokenTree);
```

close the output file

```
output.close();
END; (* IF *)
```

close the input file

```
input.close();
END Main.
```

Annexe III: Description des librairies

Librairies de bas niveau

Communication

HTMLFTP

```
(* Copyright (C) 1995, Martin Gagnon                                    *)
(* All rights reserved.                                                 *)
(* See the file COPYRIGHT for a full description.                       *)
(*                                                                      *)
(* Last modified on Tue Feb 20 15:00:00 EST 1996 by Martin Gagnon*)
(*                                                                      *)

(* This interface implement the procedures for downloading
   a file located on an FTP server.
   *)
INTERFACE HTMLFTP;
IMPORT HTMLFile,HTMLStream,TCP,Thread;
CONST
     Passwd : TEXT = "WWWUser@";   (* password use for
                                      anonymous user *)
     TimeOut : LONGREAL = 5.00D0; (* timeout in sec.*)
EXCEPTION
     Abort;
PROCEDURE Standard(name:TEXT):HTMLFile.FileGen;
(* Build a file structure FileGen , when a file with
   ftp:// as a prefix
   is called *)

PROCEDURE ContactControl(fileGen:HTMLFile.FileGen):HTMLStream.Stream;
(* Make the contol connection of a FTP connection,
   the return stream will be
   used to send and receive the control information *)
PROCEDURE ContactData():TCP.Connector;
(* Create a valid data connector that will be used to
   transfer the data
   when the data connection is established by the
   control connection  *)

PROCEDURE LoadTheFile(fileGen:HTMLFile.FileGen;
      streamCtr:HTMLStream.Stream):HTMLStream.Stream
      RAISES{Abort};
(* Load the file, in the structure FileGen, on the streamer Stream so
   that the file can be read on stream.reader. If the
   connection was not
   established the Abort exception is raised
   *)
END HTMLFTP.
```

HTMLHTTP

```
(* Copyright (C) 1995, Martin Gagnon                                    *)
(* All rights reserved.                                                 *)
(* See the file COPYRIGHT for a full description.                       *)
(*                                                                      *)
(* Last modified on Tue Feb 20 15:00:00 EST 1995 by Martin Gagnon*)
(*                                                                      *)
(* This interface implements a fonction that connects a
   client to an HTTP server. It uses the TCP protocol.
   If the connection cannot be established the Abort exception
   is raised.
   *)
INTERFACE HTMLHTTP;
IMPORT HTMLFile,HTMLStream,TCP;
CONST
     TimeOut : LONGREAL = 5.00D0; (* Timeout for the loggin in sec.*)
TYPE
     Content = RECORD
 stream : HTMLStream.Stream;
 connection : TCP.T;
       END; (* RECORD *)
EXCEPTION
     Abort;

PROCEDURE Standard(name:TEXT):HTMLFile.FileGen;
(* Build a file structure FileGen , when a file with
   http:// as a prefix is called *)
PROCEDURE Contact(fileGen:HTMLFile.FileGen): Content
  RAISES{Abort};
(* Build the connection to an HTTP server. The stream returned is
   used to send and receive the data information *)
PROCEDURE LoadTheFile(fileGen:HTMLFile.FileGen;
                      stream:HTMLStream.Stream);
(* Load the file contained in the structure FileGen.
   The file can then be read on the streamer.reader *)

END HTMLHTTP.
```

HTMLGopher

```
(* Copyright (C) 1995, Martin Gagnon                                    *)
(* All rights reserved.                                                 *)
(* See the file COPYRIGHT for a full description.                       *)
(*                                                                      *)
(* Last modified on Tue Feb 20 15:00:00 EST 1995 by Martin Gagnon*)
(*                                                                      *)
(* This interface implements a fonction that connects a client
   to a Gopher server. It uses the TCP protocol.
   If the connection cannot be established the Abort exception
   is raised.
   *)
INTERFACE HTMLGopher;
IMPORT HTMLFile,HTMLStream;
CONST
    TimeOut : LONGREAL = 5.00D0; (* Timeout for the loggin in sec.*)
EXCEPTION
     Abort;

PROCEDURE Standard(name:TEXT):HTMLFile.FileGen;
(* Build a file structure FileGen , when a file with
   gopher:// as a prefix is call *)
PROCEDURE Contact(fileGen:HTMLFile.FileGen):HTMLStream.Stream
  RAISES{Abort};
(* Build the Gopher connection. The stream returned is
   used to send and receive the data information *)
PROCEDURE LoadTheFile(fileGen:HTMLFile.FileGen;
                      stream:HTMLStream.Stream);
(* Load the file, in the structure FileGen, on the
   streamer Stream so that
   the file can be read on stream.reader           *)

END HTMLGopher.
```

HTMLStream

```
(* Copyright (C) 1995, Martin Gagnon                                    *)
(* All rights reserved.                                                 *)
(* See the file COPYRIGHT for a full description.                       *)
(*                                                                      *)
(* Last modified on Tue Feb 20 15:00:00 EST 1996 by Martin Gagnon*)
(*                                                                      *)
(* This module implements two objects that control
   the input and output stream.
   the input stream : HTMLStream.In
   the output stream : HTMLStream.Out
   *)

INTERFACE HTMLStream;
IMPORT Wr,Rd,HTMLTree,HTMLList,HTMLFile;
TYPE
   Stream = RECORD
    writer : Wr.T;
    reader : Rd.T;
    END;
   Out <: PublicOut;
   PublicOut = OBJECT
   METHODS
       init(wr:Wr.T:= NIL):Out;
(* This procedure initializes the output stream to a writer "wr".
   If the writer is NIL, the writer will be connected on
   the HTMLFile.DefaultHTML default file. *)

       getWriter():Wr.T;
(* This procedure returns the writer of the output streamer *)
       close();
(* This procedure closes the writer of the output streamer *)
       putTree(tree:HTMLTree.TokenTree);
(* This procedure prints the whole parsing tree "tree" on
   the output stream
   of the HTMLStream.Out object
   *)
   END;
   In <: PublicIn;
   PublicIn = OBJECT
   METHODS
       init(rd:REFANY):In;
(* This procedure initializes the input stream (HTMLStream.In)
   with "rd".
   The initialization can be on a HTMLFile.T object,
   so that the reader will
   be connected to this file, or on a valid reader (Rd.T).
   Otherwise, the initialization connects to the standard input.
   *)
```

```
        close();
(* This procedure closes the reader of an HTMLStream.In object *)
        closeConnection();
(* This procedure closes the TCP connection of the reader of
   an HTMLSTream.In
   object if his flagConnection is set to TRUE *)
        getReader():Rd.T;
(* This procedure return the reader of an input streamer
   (HTMLStream.In) *)
   END;

   PROCEDURE PrintTree(item:REFANY;stream:Wr.T);
(* This procedure typecast the token and put it on the output stream
   "stream"
   *)

   PROCEDURE TraverseTree(ptr:HTMLTree.Node;stream:Wr.T);
(* This procedure iterates over a tree prints each node to
   the output stream "stream".
   "ptr" must reference a token tree. *)

END HTMLStream.
```

Fichiers

HTMLFile

```
(* Copyright (C) 1995, Martin Gagnon                                    *)
(* All rights reserved.                                                 *)
(* See the file COPYRIGHT for a full description.                       *)
(*                                                                      *)
(* Last modified on Tue Feb 20 15:00:00 EST 1996 by Martin Gagnon*)
(*                                                                      *)
(* This interface implements an object HTMLFile.T. This object can be
   initialized by a local file name or an Uniform Resource Locator
   ( URL ). The initialization will create a FileGen
   structure that will
   store all the information extracted from the URL. The pathname is
   converted to a cannonical form to detect identical path.
   *)
INTERFACE HTMLFile;
IMPORT Pathname,IP;
CONST
     (* Theses are the default output file names *)
     DefaultHTML : Pathname.T = "out.html";
     outToC : Pathname.T = "ToC.html";
     outIndex : Pathname.T = "Index.html";
     DefaultLaTeX : Pathname.T = "out.tex";
     DefaultASCII : Pathname.T = "out.ascii";
TYPE
     (* nature of the HTML file
Local : file located on local disk
HTTP : file reach by http protocol
FTP : file reach by ftp protocol
GOPHER : file reah by gopher protocol
File : file reach by one of the above protocol
       ( not implemented yet ) *)
     FileNature = { Local,HTTP,FTP,Gopher,File };
     (* record that store all the information extract from a URL *)
     FileGen = RECORD
       name : Pathname.T;  (* name of the file *)
       host : TEXT;   (* host of the file location *)
       port : IP.Port;  (* port of the host *)
       flag : BOOLEAN := FALSE;  (* flag set to true, if the file
    is "path dependant" of his
    ancestor file's pathname *)
       END; (* RECORD *)

     T <: Public;
     Public = OBJECT
     METHODS
       init(names:TEXT;fileAnc:T:=NIL):T;
       (* Initialize an HTMLFile.T  object by creating
          a FileGen structure
  with the URL "name". FileAnc is the current file name and is
  useful for relative paths *)
```

```
        getFileGen():FileGen;
        (* Return the FileGen structure *)
        getName():Pathname.T;
        (* Return the name of the file *)
        getType():FileNature;
        (* Return the nature of the file *)
      END;
PROCEDURE IdentType(name:TEXT):FileNature;
(* This procedure identify the nature of the URL "name" *)
PROCEDURE Standard(name:TEXT;pathAbs:Pathname.T):Pathname.T;
(* This procedure create an absolute file path with the absolute path
   "pathAbs" and the file name "name" *)
PROCEDURE FormatPath(path:Pathname.T):Pathname.T;
(* This procedure removes all the /../ present in the
   pathname "path" *)
PROCEDURE StandardFile(name:TEXT): FileGen;
(* Build a type structure FileGen from a standard pathname "name",
   WARNING : use only on a local file *)
PROCEDURE GetTmpFile(name:TEXT;ext:TEXT;Delete:BOOLEAN:=TRUE)
                    : Pathname.T;
(* This procedure return an unique temp file and automatically
   deletes it if the flag Delete is TRUE *)
END HTMLFile.
```

Texte

HTMLText

```
(* Copyright (C) 1995, Martin Gagnon                                  *)
(* All rights reserved.                                               *)
(* See the file COPYRIGHT for a full description.                     *)
(*                                                                    *)
(* Last modified on Tue Feb 20 15:00:00 EST 1995 by Martin Gagnon*)
(*                                                                    *)
(* This interface implements some tools used by other modules
   and define some constant.          *)
INTERFACE HTMLText;
IMPORT HTMLList;
TYPE
     Alphabet = ARRAY [1..26] OF CHAR;
CONST
     slash : TEXT = "/";
     spa : CHAR = ' ';
     tab : CHAR = '\t';
     CR : CHAR = '\r';
     NL : CHAR = '\n';
     FF : CHAR = '\f';
     equal : CHAR = '=';
     greThen:TEXT="<";
     space:TEXT=" ";
     lesThen:TEXT=">";
     amper:TEXT="&";
     semicolon:TEXT=";";
     quot:TEXT="\"";
     alpha = Alphabet{'A','B','C','D','E','F','G','H','I','J','K'
,'L','M','N','O','P','Q','R','S','T','U','V','W','X','Y','Z'};
PROCEDURE CompareText(string1,string2:TEXT):BOOLEAN;
(* This procedure compares 2 strings of text and returns
   true if they are case
   sensitive identical *)
PROCEDURE TextEmpty(text:TEXT):BOOLEAN;
(* This procedure returns true if the text in the string
   is composed of
   one or more of these charracters ( <space>, <tab> ) *)
PROCEDURE Equal(string1,string2:TEXT):BOOLEAN;
(* This procedure compares 2 strings of text and returns true
   if they are
   non-case sensitive identical *)
PROCEDURE RemoveQuote(string:TEXT):TEXT;
(* This procedure removes the bounded quotes from a string *)
PROCEDURE UPCase(char:CHAR):CHAR;
(* This procedure puts the character char in uppercase *)
PROCEDURE Parse(string:TEXT;char:CHAR):HTMLList.T;
```

```
(* This procedure parses a string of text with the character char and
   returns a list of strings that were separated char *)
END HTMLText.
```

Reconnaissance de format

HTMLFormat

```
(* Copyright (C) 1995, Martin Gagnon                                    *)
(* All rights reserved.                                                 *)
(* See the file COPYRIGHT for a full description                        *)
(*                                                                      *)
(* Last modified on Tue Feb 20 15:00:00 EST 1996 by Martin Gagnon*)
(*                                                                      *)
(* This inteface gives some information about the HTML 3 format
   parsing *)
INTERFACE HTMLFormat;
TYPE
     (* These are the mnmemonics of the HTML 3 language derived
from the SGML standard *)
    Tag = { TITLE,
    OL,
    UL,
    LI,
    DL,
    DT,
    DD,
    FIG,
    HEAD,
    BODY,
    BOX,
    MATH,
    LEFT,
    RIGTH,
    OVER,
    ATOP,
    ARRAYS,
    ROOTS,
    BELOW,
    ABOVE,
    A,
    P,
    B,
    I,
    S,
    U,
    TT,
    BIG,
    SUB,
    SUP,
    IMG,
    LH,
    SMALL,
    EM,
    HR,
    NOTE,
    BQ,
    CREDIT,
    ADDRES,
    BR,
    PRE,
```

```
      FN,
      XMP,
      CITE,
      H1,
      H2,
      H3,
      H4,
      H5,
      H6,
      META,
      TABLE,
      TR,
      TH,
      TD,
      CAPTION,
      NILE,
      TITLEend,
      OLend,
      ULend,
      DLend,
      FIGend,
      HEADend,
      BODYend,
      BOXend,
      MATHend,
      Aend,
      Bend,
      Iend,
      Send,
      TTend,
      BIGend,
      Uend,
      SUBend,
      SUPend,
      SMALLend,
      EMend,
      LHend,
      NOTEend,
      BQend,
      CREDITend,
      ADDRESend,
      PREend,
      FNend,
      XMPend,
      CITEend,
      H1end,
      H2end,
      H3end,
      H4end,
      H5end,
      H6end,
      TABLEend,
      CAPTIONend,
      UNKNOW
       };

      (* These are the mnemonics of the ISO characters defined in the
SGML standard *)

      TagChar = { QUOTE,
```

AMP,
LT,
GT,
AElig,
Aacute,
Acirc,
Agrave,
Aring,
Atilde,
Auml,
Ccedil,
ETH,
Eacute,
Ecirc,
Egrave,
Euml,
Iacute,
Icirc,
Igrave,
Iuml,
Ntilde,
Oacute,
Ocirc,
Ograve,
Oslash,
Otilde,
Ouml,
THORN,
Uacute,
Ucirc,
Ugrave,
Uuml,
Yacute,
aacute,
acirc,
aelig,
agrave,
aring,
atilde,
auml,
ccedil,
eacute,
ecirc,
egrave,
eth,
euml,
iacute,
icirc,
igrave,
iuml,
ntilde,
oacute,
ocirc,
ograve,
oslash,
otilde,
ouml,
szlig,
thorn,
uacute,

```
 ucirc,
 ugrave,
 uuml,
 yacute,
 yuml,
 UNKNOW };

PROCEDURE TextToTag(tag:TEXT):Tag;
(* This procedure transforms a string of TEXT  into a mnemonics of
   HTML 3, if it does not find one it returns Tag.UNKNOW  *)
PROCEDURE Character(char:TEXT):TagChar;
(* This procedure transforms a string of TEXT ,
   identifyize a character,
   into a mnemonics of SGML ISO CHARACTER, if it does not find one it
   returns TagChar.UNKNOW.
   It handle numerical ( &12; ) and alphanumerical ( &gt; )
   entities *)
END HTMLFormat.
```

Outils oeuvrant sur des structures de données de type liste ou arbre binaire

HTMLList

```
(* Copyright (C) 1995, Martin Gagnon                                  *)
(* All rights reserved.                                               *)
(* See the file COPYRIGHT for a full description.                     *)
(*                                                                    *)
(* Last modified on Tue Feb 20 15:00:00 EST 1995 by Martin Gagnon*)
(*                                                                    *)
(* This interface implements an object HTMLList.T with all the
   methods to work on it.
   The object is a list with two pointers,
   ptrHead : points to the head and its items is always NIL
   ptr : the other pointer     *)

INTERFACE HTMLList;
TYPE
    Node = REF RECORD
   item : REFANY;
   next : Node := NIL;
    END;
    T <: Public;
    Public = OBJECT
    METHODS
       init(items:REFANY):T;
       resetList();
(* This procedure resets the list by assigning the pointer to the
   node next to the head pointer ( the 1st NON-NULL) *)
       resetListInit();
(* This procedure resets the list by assigning the pointer directly
   to the NULL head pointer *)
       countItem():CARDINAL;
(* This procedure counts the number of items in the list *)
       nextItem(count:CARDINAL:=1);
(* This procedure goes to the next item with an increment of 1, by
   default the increment is 1 *)
       insertItem(item:REFANY);
(* This procedure inserts a node with the item "items"
   after the ptr *)
       deleteItem():REFANY;
(* This procedure deletes the node refrenced by ptr and
   returns the item private fonction *)
       deleteNextItem():REFANY;
(* This procedure deletes the next node after the one
   refrenced by ptr and returns the item private fonction *)
       getItem():REFANY;
(* This procedure returns the item *)
       changeItem(items:REFANY);
```

```
(* This procedure changes the item in a list *)
      getHead():Node;
(* This procedure returns the head of the list *)
      getPtr():Node;
(* This procedure returns the active pointer in the list *)
      EOL():BOOLEAN;
(* This procedure returns TRUE if we are at the end of the list,
   FALSE otherwise *)
      goEOL();
(* This procedure goes at the end of the list *)
    END;
PROCEDURE DelNextItem(ptr:Node):REFANY;
(* This procedure deletes the next node after the one referenced
   by ptr and returns the item public fonction *)
PROCEDURE DelItem(ptr,ptrHead:Node):Node;
(* This procedure deletes the node referenced by ptr and returns the
   item *)
END HTMLList.
```

HTMLTree

```
(* Copyright (C) 1995, Martin Gagnon                                *)
(* All rights reserved.                                             *)
(* See the file COPYRIGHT for a full description.                   *)
(*                                                                  *)
(* Last modified on Tue Feb 20 15:00:00 EST 1996 by Martin Gagnon*)
(*                                                                  *)

(* This interface implements two object types : a binary tree and
   a token tree derived from the first one *)
INTERFACE HTMLTree;
IMPORT HTMLList,HTMLParser;
TYPE
   (* Node tree *)
   Node = REF RECORD
  item : REFANY;    (* the item *)
  father : Node :=NIL;  (* its father *)
  brotherAin : Node := NIL;  (* previous brother *)
  brother : Node := NIL;  (* next brother *)
  child : Node := NIL;  (* its child *)
   END;

   Tree <: PublicTree;
   PublicTree = OBJECT
   METHODS
       init(items:REFANY):Tree;
(* This procedure initializes a tree with the items "items". *)
       resetTree();
(* This procedure resets a tree a by setting its "ptr" to head. *)
       insertItem(items:REFANY;child:BOOLEAN:=FALSE);
(* This procedure inserts an item next to the pointer "ptr".
   If child is FALSE then the item will be a brother of ptr.
   If child is TRUE  then the item will be a child of ptr.
   *)
       deleteItem():Node;
(* This procedure deletes the item points by "ptr". *)
       deleteNextItem():Node;
(* This procedure deletes the next item points by "ptr". *)
       deleteChild();
(* This procedure deletes all the children of the ptr "ptr". *)
       nextFather():BOOLEAN;
(* This procedure sets the pointer "ptr" to its father and
   returns TRUE, FALSE otherwise.
   *)
       nextChild():BOOLEAN;
(* This procedure sets the pointer "ptr" to its first child and
   returns TRUE, FALSE otherwise.
   *)
```

```
        nextBrother():BOOLEAN;
(* This procedure sets the pointer "ptr" to the brother
   next to it and returns TRUE, FALSE otherwise.
   *)
        getItem():REFANY;
(* This procedure returns the item referenced by "ptr". *)
        getHead():Node;
(* This procedure returns the "head" . *)
        getPtr():Node;
(* This procedure returns the "ptr" . *)
        setPtr(ptr:Node);
(* This procedure sets the pointer "ptr". *)
    END;
    TokenTree <: PublicTokenTree;
    PublicTokenTree = Tree OBJECT
    METHODS
        init(list:HTMLList.T):TokenTree;
(* This procedure initializes and builds a tokentree
   from a parsing list sent by an HTMLParser.T object.
   *)
        findTag(tag:TEXT): BOOLEAN;
(* This procedure finds the tag "tag" in a token tree.
   If it finds one it returns TRUE and sets the pointer "ptr" on it.
   If it does not find one it returns FALSE.
   It begins the search at the starting pointer "ptr", and
   finds the first matching tag.
   *)
    END;
    PROCEDURE InsertBrother(ptr:Node;items:REFANY);
(* This procedure inserts an item as a brother of the node "ptr". *)
    PROCEDURE InsertNewParent(ptr:Node;items:REFANY);
(* This procedure inserts an item as a parent of the node "ptr". *)
    PROCEDURE DelItem(ptr:Node):Node;
(* This procedure deletes the item referenced by "ptr". *)
END HTMLTree.
```

Parser

HTMLParser

```
(* Copyright (C) 1995, Martin Gagnon                                  *)
(* All rights reserved.                                               *)
(* See the file COPYRIGHT for a full description.                     *)
(*                                                                    *)
(* Last modified on Tue Feb 20 15:00:00 EST 1996 by Martin  Gagnon*)
(*                                                                    *)

(* This interface implements an HTMLParser.T object .
   This object returns a
   list of tokens generated from the parsing of an
   SGML derived language.
   The tokens are separated in two categories :
  1-Tag
    the tag delimitaters are < and >
    the main tag and his attributes are stored in an atoms list
    the < and > are NOT in the list.
    the comments have the specific tag "!" and the rest of the
    text is its only attribute
  2-TEXT
    everythings else that is not in the < > delimiters.
   *)

INTERFACE HTMLParser;
IMPORT HTMLList,Rd;
TYPE
   (* A list of Atom.T identifying each attribute in the tag *)
   Tag = RECORD
   attributeList : HTMLList.T;
 END; (* RECORD *)

   T<:Public;
   Public=OBJECT
     METHODS
init(file:Rd.T):T;

(* This procedure initializes a file with a reader
   for a future parsing *)

getTokenList():HTMLList.T;

(* This method returns a list of tokens *)

     END;

PROCEDURE Traduce(text:TEXT):TEXT;
(* This procedure translates the < and > characters to the rigth
   delimiter Tag. This procedure is useful to convert an encoded
   tag used to generate an index.
   Example :
   If text is "&lt;I&gt;" it will return "<I>"
```

```
   *)

PROCEDURE GetToken(source:Rd.T): REFANY;
(* This procedure returns the next valid token from a reader.
   The token can be a Text field or a Tag.
   NIL if no valid token was found.
   *)

PROCEDURE GetTag(source:Rd.T):REF Tag;
(* This procedure returns the Tag, if its not a valid tag it returns
   NIL *)

PROCEDURE GetField(source:Rd.T):TEXT;
(* This procedure returns the Text field, if it is not a
   valid text field it returns NIL *)
END HTMLParser.
```

HTMLCommLine

```
(* Copyright (C) 1995, Martin Gagnon                                    *)
(* All rights reserved.                                                 *)
(* See the file COPYRIGHT for a full description.                       *)
(*                                                                      *)
(* Last modified on Tue Feb 20 15:00:00 EST 1996 by Martin Gagnon*)
(*                                                                      *)
(* This interface sets the options that
   can be found on the command line
   their meaning is detailed in the main module *)
INTERFACE HTMLCommLine;
TYPE
   CommLine = { FTeX,
FASCII,
FToC,
FLoF,
FLoT,
FLRef,
FDEBUG,
FIndex,
FPS,
FOut,
FInd,
FPatternDiag,
FPrune,
FInclude,
UNKNOW };
(* these are the possibles options that
   a user can set  *)

PROCEDURE ParseComm(commText:TEXT) : CommLine;
(* This procedure receives each parameter sent by Params.Get,
   and returns a type identifying the options
   set by the user                              *)
END HTMLCommLine.
```

Librairies de niveau Intermediaire

Outils oeuvrant sur des fichiers HTML linéaire ou hiéararchique

HTMLToolList

```
(* Copyright (C) 1995, Martin Gagnon                                    *)
(* All rights reserved.                                                 *)
(* See the file COPYRIGHT for a full description.                       *)
(*                                                                      *)
(* Last modified on Tue Feb 20 15:00:00 EST 1996 by Martin Gagnon*)
(*                                                                      *)
(* This interface implements somes procedures to work
   on a list of tokens produced by the parsing.
   Two kind of structured form the list of tokens.
   A HTMLParser.Tag used to hold the tag and its attributes
   and TEXT used to hold the data field.
   Each pointer used by the procedures is of type HTMLList.Node.
   *)
INTERFACE HTMLToolList;
IMPORT HTMLList;

PROCEDURE GetTag(ptr:HTMLList.Node):TEXT;
(* This procedure returns a string of text that identifies the tag
   referenced by ptr. If ptr does not point on a HTMLParser.
   Tag structure it returns an empty string.
   *)

PROCEDURE InsertAttribut(ptr:HTMLList.Node;attribu:TEXT);
(* This procedure adds an attribute to a tag referenced by "ptr".
   The attribute is inserted directly after the main tag. The
   attribute may written as :
   CLASS , CLASS= , CLASS=INDEX .
   *)

PROCEDURE GetTheAttribute(ptr:HTMLList.Node;attribu:TEXT):REFANY;
(* This procedure gets the value of an attribute.
   The attribute is a part of a
   tag referenced by "ptr". If there is no
   attribute mathing the "attribu"
   string it returns the NIL pointer. If the attribute is there but
   has no value, it returns an empty string.
   *)

PROCEDURE SetTheAttribute(ptr:HTMLList.Node;attribu:TEXT;new:TEXT);
(* This procedure sets the value "new" to an attribute "attribu" of a
   tag refernced by "ptr". If there is no attribute "attribu",
   a new one
   is created and set to the desired value.
   *)

PROCEDURE DelTheAttribute(ptr:HTMLList.Node;attribu:TEXT);
(* This procedure deletes an attribute "attribu" from a tag
```

```
   referenced by "ptr". *)
END HTMLToolList.
```

HTMLToolTree

```
(* Copyright (C) 1995, Martin Gagnon                                    *)
(* All rights reserved.                                                 *)
(* See the file COPYRIGHT for a full description.                       *)
(*                                                                      *)
(* Last modified on Tue Feb 20 15:00:00 EST 1996 by Martin Gagnon*)
(*                                                                      *)

(* This interface implements some procedures to
   work on a tree of tokens.
   Two kind of structures form the tree of tokens.
   A HTMLParser.Tag is used to hold the tag and its attribute.
   and TEXT holds the data field.
   Each pointer use by the procedures is of type HTMLTree.Node.
   *)

INTERFACE HTMLToolTree;
IMPORT HTMLTree,HTMLParser;

PROCEDURE IsThisTag(ptr:HTMLTree.Node;tag:TEXT):BOOLEAN;
(* This procedure returns TRUE if the name of the
   tag referenced by "ptr"
   matches exactly the string "tag".
   This means that if you specify tag
   as H1 and the tag is /H1, it will return false.
   *)

PROCEDURE IsThatTag(ptr:HTMLTree.Node;tag:TEXT):BOOLEAN;
(* This procedure returns TRUE if the string "tag"
   matches partially or
   completely the name of the tag referenced by "ptr".
   This means that
   if you specify tag as H1 and the tag is /H1, it will return TRUE.
   *)

PROCEDURE GetTheTag(ptr:HTMLTree.Node):TEXT;
(* This procedure returns a string of text that identifies the tag
   referenced by "ptr". If "ptr" does not point on a HTMLParser.Tag
   structure, it returns an empty string.
   *)

PROCEDURE GetTheAttribute(ptr:HTMLTree.Node;attribu:TEXT):REFANY;
(* This procedure gets the value of an attribute.
   The attribute is part of a
   tag referenced by "ptr". If there is no attribute
   mathing the "attribu"
   string, it returns the NIL pointer. If the attribute is there but
   has no value, it returns an empty string.
   *)

PROCEDURE SetTheAttribute(ptr:HTMLTree.Node;attribu:TEXT;new:TEXT);
(* This procedure sets the value "new" to an attribute "attribu" of a
   tag referenced by "ptr". If there is
   no attribute "attribu", a new one
   is created and set to the desired value.
```

```
   *)
PROCEDURE InsertAttribut(ptr:HTMLTree.Node;attribu:TEXT);
(* This procedure adds an attribute to a tag referenced by "ptr".
   The attribute is insert as directly after the main tag. The
   attribute may be written as :
   CLASS , CLASS= , CLASS=INDEX .
   *)

PROCEDURE ChangeTheTag(ptr:HTMLTree.Node;newTag:TEXT);
(* This procedure changes the name of the tag
   referenced by "ptr" to a
   new name "newTag". All the attribute remain unchanged.
   *)

PROCEDURE MakeATag(v1:TEXT:=NIL;
   v2:TEXT:=NIL;
   v3:TEXT:=NIL;
   v4:TEXT:=NIL;
   v5:TEXT:=NIL;
   v6:TEXT:=NIL;
   v7:TEXT:=NIL):REF HTMLParser.Tag;
(* This procedure creates a new tag and returns it. The name of
   the tag will be v1 and v2 .. v7 will be its attributes.
   A maximum of 6 attribute is possible.
   *)
PROCEDURE FindTextHeader(ptr : HTMLTree.Node;
 flag : REF BOOLEAN;
 VAR Holdptr : HTMLTree.Node);
(* This procedure finds the first field of text of a header and
   holds a ptr on it.
   ptr : the header ptr.
   flag : flag set if the text is found.
   VAR Holdptr : the holder of the TEXT ptr.
*)

PROCEDURE ReturnAttribut(string:TEXT):TEXT;
(* This procedure returns the sub-string after
   the character '=' in the
   string "string". If no char '=' is found it returns
   an empty string.
   This is useful to get the value of an attribute.
   *)
END HTMLToolTree.
```

Redéfinition des liens

HTMLAdjustLinks

```
(* Copyright (C) 1995, Martin Gagnon                                    *)
(* All rights reserved.                                                 *)
(* See the file COPYRIGHT for a full description.                       *)
(*                                                                      *)
(* Last modified on Tue Feb 20 15:00:00 EST 1996 by Martin Gagnon*)
(*                                                                      *)
(* This interface adjusts the links of the linear document after the
   linearization *)
INTERFACE HTMLAdjustLink;
IMPORT Pathname,HTMLList,HTMLFile,Wr,TableNode;
TYPE
     (* this record contains a structure FileGen which holds all the
information about a node that has been reached.
A unique ID is assigned to it. *)

     Content = RECORD
       file : HTMLFile.FileGen;
       id : TEXT := NIL;
       END; (* RECORD *)
PROCEDURE AdjustLink(file:Pathname.T := HTMLFile.DefaultHTML;
     tableNode:TableNode.Default;
     wrMsg : Wr.T := NIL);
(* Adjusts the link in the file "file". If one of
   its links is in the list of
   the nodes visited "listNode" , it will be
   converted to an internal link.
   This means that if a link is :
   HREF="http://www.vlsi.polymtl.ca/index.html#start" (external link)
   and www.vlsi.polymtl.ca/index.html is in the list
   of nodes visited, it will be changed to :
   HREF="#start" (internal link)
   Any message will be written on the wrMsg writer (by default is the
   Stdio.stdout) *)
END HTMLAdjustLink.
```

Librairies de haut niveau

Linéarisateur

HTMLLinear

```
(* Copyright (C) 1995, Martin Gagnon                                    *)
(* All rights reserved.                                                 *)
(* See the file COPYRIGHT for a full description.                       *)
(*                                                                      *)
(* Last modified on Tue Feb 20 15:00:00 EST 1996 by Martin Gagnon*)
(*                                                                      *)
(* This interface implements somes procedures to work on a HTML file.
   All theses operate on a HTMLTree.TokenTree structure.
   The main goal of this interface is to linearize a web graph.
   *)

INTERFACE HTMLLinear;
IMPORT HTMLTree,HTMLList,HTMLStream,HTMLFile,TableNode;
CONST
     MaxDep = 20; (* Maximum depth to explore in a web graph *)
TYPE
   (* This array stores the value of the the last
      header where the file inclusion was made *)
   DepArr = ARRAY[0..MaxDep] OF CARDINAL;
   (* type of the link *)
   Relation = { SUBDOCUMENT,INCLUDES };
   (* a link to follow *)
   Link = RECORD
 ptr : HTMLTree.Node; (* where in the tree the inclusion has to
 be made *)
 relation : Relation; (* type of the link *)
 depth : CARDINAL := 0; (* the depth of the file relative
                                    to the file *)
 END; (* RECORD *)


PROCEDURE SearchAllSubReference(tree:HTMLTree.TokenTree;
file:HTMLFile.T;
depArr : DepArr;
doc : CARDINAL;
depth:REF CARDINAL;
patPrune : TEXT;
patInclude : TEXT):HTMLList.T;
(* This procedure searches all the "valid"
   sub references to perform a
   linearization. It returns a list of valid "Link" to follow.
   VARIABLE :  tree : the target file in a tokentree form.
       file : the info on the target file.
       depArr : this array provides information
                        on how deep the target
```

```
file is in the linearization, and what was the
value of the that last header before this file
was included in the linearization.
       doc : the id of the document.
       depth : how deep the file is in the linearization.
       patPrune : the prune pattern.
       patInclude : the include pattern.
   NOTE : if a string matches the two patterns, the prune pattern has
  priority over the include pattern.
*)

PROCEDURE Linear(fileName : TEXT;
 tableNode : TableNode.Default;
 output : HTMLStream.Out;
 file : HTMLFile.T;
 flag : BOOLEAN;
 depth : CARDINAL;
 patPrune : TEXT;
 patInclude : TEXT );

(* This procedure linearizes a web graph.
   This procedure is called recursively to perform
   the linearization. To
   begin a linearization, give an empty tableNode,
   a file ancestor set
   to NIL, and the depth set to 0.
   VARIABLE :  fileName : the entry file .
       tableNode : a table that stores the nodes visited.
       output : the output stream for the linear document.
       file : the ancestor file of the entry file.
       flag : set to TRUE if the file is a SUBDOCUMENT,
      FALSE, if it is an INCLUDE file.
       depth : the depth of the file in the linearization.
       patPrune : the prune pattern.
       patInclude : the include pattern.
   NOTE : if a string matches the two patterns the prune pattern has
  priority over the include pattern.
*)

PROCEDURE Hierarchie(ptr:HTMLList.T;
     tableNode:TableNode.Default;
     tokenTree:HTMLTree.Node;
     output:HTMLStream.Out;
     file:HTMLFile.T;
     patPrune : TEXT;
     patInclude : TEXT);

(* This procedure will modify a token tree by inserting
   other trees into it. The other trees come from the list
   of links to follow.
   This procedure calls itself recursively until the
   list of subreference is empty. Each time a new node is reached a
   new list is created.
   VARIABLES
       ptr : a list of links to follow
       tableNode : the table of nodes visited
       tokenTree : the head of the tree to modify
       output : the output stream for the linear document.
```

```
        file : the ancestor file of the tree.
        patPrune : the prune pattern.
        patInclude : the include pattern.

    *)

PROCEDURE ChangeHeader(tree:HTMLTree.TokenTree;deep:CARDINAL);
(* This procedure changes the value of all the headers
   of token tree "
   tree". The value is set by the "deep".
   *)

PROCEDURE InsertSpotID(tree:HTMLTree.TokenTree;id:CARDINAL);
(* This procedure inserts a spot tag <SPOT ID="id"> at the top
   of the tree.
   *)

END HTMLLinear.
```

Générateur de fichier ASCII

HTMLToASCII

```
(* Copyright (C) 1995, Martin Gagnon                                  *)
(* All rights reserved.                                               *)
(* See the file COPYRIGHT for a full description.                     *)
(*                                                                    *)
(* Last modified on Tue Feb 20 15:00:00 EST 1995 by Martin Gagnon*)
(*                                                                    *)

(* This interface uses the command lynx -dump  to provide a procedure
   to convert HTML to ASCII. If the command is not present, the
   conversion is aborted and a message is displayed.
   *)

INTERFACE HTMLToASCII;
IMPORT Pathname,HTMLFile,Wr;
PROCEDURE StartGenASCII(nameIn:Pathname.T;
nameOut:Pathname.T:=HTMLFile.DefaultASCII;
msgWr:Wr.T:=NIL);
(* This procedure starts the conversion of the
   input file "nameIn" and
    produces the output file "nameOut" ( out.ascii by default ).
    The message will be sent on the msgWr writer.
    *)
END HTMLToASCII.
```

Générateur de fichier LaTeX

HTMLToTex

```
(* Copyright (C) 1995, Martin Gagnon                                  *)
(* All rights reserved.                                               *)
(* See the file COPYRIGHT for a full description.                     *)
(*                                                                    *)
(* Last modified on Tue Feb 20 15:00:00 EST 1995 by Martin Gagnon*)
(*                                                                    *)
(* This interface implements all the procedures useful for *)
(* the conversion of HTML to LateX language                 *)
INTERFACE HTMLToTex;
IMPORT Pathname,HTMLList,HTMLFile,Wr;
PROCEDURE StartGenTex(nameIn:Pathname.T;
      nameOut : Pathname.T;
      ToC : BOOLEAN := FALSE;
      ListOfFigures : BOOLEAN := FALSE;
      ListOfTable : BOOLEAN := FALSE;
      FPS : BOOLEAN := FALSE;
      listRef : HTMLList.T := NIL;
      patDiag : TEXT := NIL;
      commDiag : TEXT := NIL;
      msgWr:Wr.T:=NIL);
(* Produce a Latex file from the HTML file specified by nameIn.
   The output defaults to out.tex if it is not specified.
   nameIn : input file (HTML)
   nameOut : output file ( with ext. .tex .log .dvi .aux .ps )
   ToC : if true produce a table of content
   ListOfFigures : if true produce a list of figures
   ListOfTable : if true produce a list of tables
   FPS : if true produce a postscript document
   listRef : if NON-NIL produce a list of references
 *)
END HTMLToTex.
```

Outil pour construire la table des matières

HTMLToC

```
(* Copyright (C) 1995, Martin Gagnon                                  *)
(* All rights reserved.                                               *)
(* See the file COPYRIGHT for a full description.                     *)
(*                                                                    *)
(* Last modified on Tue Feb 20 15:00:00 EST 1996 by Martin Gagnon*)
(*                                                                    *)
(* This interface builds a table of contents
   for a target file "fileIn".
   The table of contents consists in a nested list of all the
   headers in the target file.
   Two ToC are created, one is added at the beginning of the
   target file and its links point to the headers in the file.
   The other is in the ToC.html file and its link point to
   the headers in the original files ( if a linearization was done ).
   If no linearization was done before, the two ToC will point to
   the same target. To keep track of the original files,
   the table of nodes is required, and an attribute "CLASS" is added
   to each header to specify in wich file
   this header first occurred.
  *)

INTERFACE HTMLToC;
IMPORT Pathname,HTMLList,HTMLFile,Wr,TableNode;
TYPE
    (* content of the ToC *)
    Content = RECORD
      title : TEXT;     (* the text of the header *)
      idFile : HTMLFile.FileGen;   (* in which file it occurs *)
      idLin : TEXT := "";     (* its ID in the linear doc *)
      idHie : TEXT := "";     (* its ID in the original doc *)
      deep : CARDINAL;     (* its depth *)
      END; (* RECORD *)
PROCEDURE BuildToC(fileIn:Pathname.T;
   tableNode:TableNode.Default;
   msgWr:Wr.T:=NIL);
(*  This procedure builds the table of contents .
    fileIn : the target file where the index is extracted from.
    tableNode : a table of the nodes visited , useful to keep
                track of the
original file for a linear document. If it is not a linear
document, pass an empty table.
    msgWr : the output for the messages.       *)

END HTMLToC.
```

Outil pour construire l'index

HTMLIndex

```
(* Copyright (C) 1995, Martin Gagnon                                    *)
(* All rights reserved.                                                 *)
(* See the file COPYRIGHT for a full description.                       *)
(*                                                                      *)
(* Last modified on Tue Feb 20 15:00:00 EST 1996 by Martin Gagnon*)
(*                                                                      *)
(* This interface produces an index by extracting
   the information from the
   tag SPOT . The extracted information is encoded this way
   <SPOT CLASS=INDEX ID="key_XXXXX_text">
   where :
   CLASS=INDEX specifies that this tag is used to make
   an index of the file.
   ID="key_XXXXX_text" is the encoded information,
 key : Key to use for the sorting
 XXXXXX : String to use in the index ( with all its attributes )
 text : ID of the index ( to avoid multiple definition of the
  same word )
    This module modifies the target file ("fileIn") by appending the
    index to it, and creates a new file ( Index.html ).
    The links in the index appended to the target file point
    in the same
    file. The links of the Index.html file point to the
    original files
    where the indexed word first occurred. The table of nodes
    visited is
    used to keep track of the original files. *)
INTERFACE HTMLIndex;
IMPORT Pathname,HTMLList,HTMLFile,Wr,TableNode;
CONST
     Max = 20;
TYPE
     (* content of an index *)
     Index = RECORD
     wordSort : TEXT;     (* the key for the sorting *)
     stringIndex : TEXT;  (* the string that appears in the index *)
     attribu : TEXT;     (* the attribu of the index if we got some
   duplicate word in the index *)
     ID : TEXT;          (* his ID *)
     idFile : HTMLFile.FileGen;  (* the file in which it occurs *)
     END; (* RECORD *)
     Ind = REF ARRAY OF Index;
PROCEDURE BuildIndex(fileIn:Pathname.T;
     tableNode:TableNode.Default;
     msgWr:Wr.T:=NIL);
(*  This procedure builds the index .
    file : the target file where the index is extracted from.
    tableNode : a table of the nodes visited , useful to keep
        track of the
```

```
original files for a linearized document. If it
is not a linearized document, pass an empty table.
    msgWr : the output for the messages.       *)
END HTMLIndex.
```

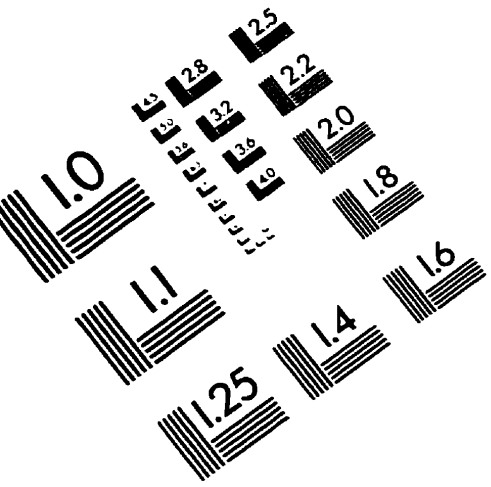
Outil pour construire la liste des références

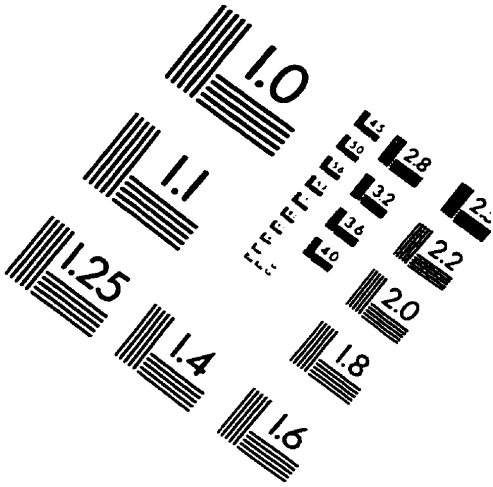
HTMLReference

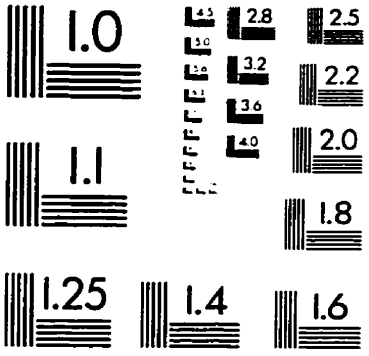
```
(* Copyright (C) 1995, Martin Gagnon                                    *)
(* All rights reserved.                                                 *)
(* See the file COPYRIGHT for a full description.                       *)
(*                                                                      *)
(* Last modified on Tue Feb 20 15:00:00 EST 1996 by Martin Gagnon*)
(*                                                                      *)
(* This interface builds the list of references for an HTML file.
   The list of
   references will include  all the HREF that have not the relation
   REL set to INCLUDE or SUBDOCUMENT. This includes all these
   types of references :
   HREF="http://www.vlsi.polymtl.ca/Features.html"
   HREF="http://www.vlsi.polymtl.ca/Bugs.html#toto"
   From each reference found, a bibliography entry is extracted.
   Two classes of bibliography entries can be created :
   CLASS.Bibitem : this one is created from files that
   have a specific
   section for a bibliographic entry.
   <DIV ID="here" CLASS=BIBITEM>
   Author,<I>book title</I>,O'Connel, 1995
   </DIV>
   CLASS.Ordinary : this one is created by default if a BIBITEM
    section is not found in the document.
    In this case a bibliography item can be deduced from these
    tags :
    <TITLE> book title </TITLE>
    ...
    <P>
    <DIV CLASS=AUTHOR> Author </DIV>
    <P>
    <DIV CLASS=BIBITEM.TAIL> O'Connel,1995
    </DIV>

   This module raises an error if the syntax of the bibitem is
   incorrect.
*)
INTERFACE HTMLReference;
IMPORT Pathname,HTMLList,Wr;
TYPE
     (* possible classes of a bibliography items *)
     CLASS = { Ordinary,Bibitem };

     (* content of a bibliography item *)
     Reference = RECORD
 class : CLASS;  (* the class of the item *)
 author : TEXT :="";  (* the author of the reference *)
 title : TEXT:="";     (* the title of the reference *)
 edition : TEXT:="";    (* edition *)
 URL : TEXT;             (* URL *)
 END; (* RECORD *)
```

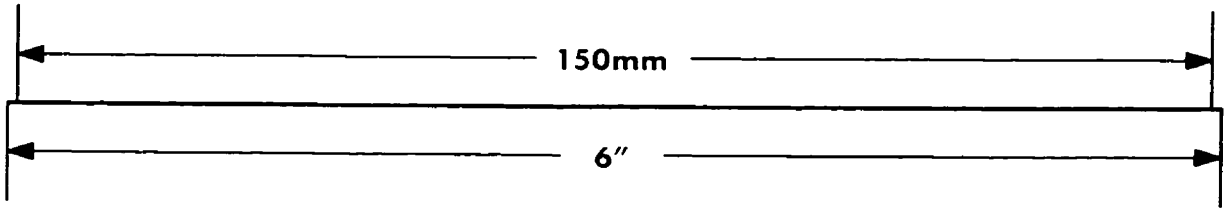
```
PROCEDURE BuildReference(nameFile:Pathname.T;
 msgWr:Wr.T:=NIL) : HTMLList.T;

(* This procedure builds the list of references from
the file "nameFile"
   and returns the list. The list is formed of References records.
   All the output messages are written to msgWr.
*)

END HTMLReference.
```

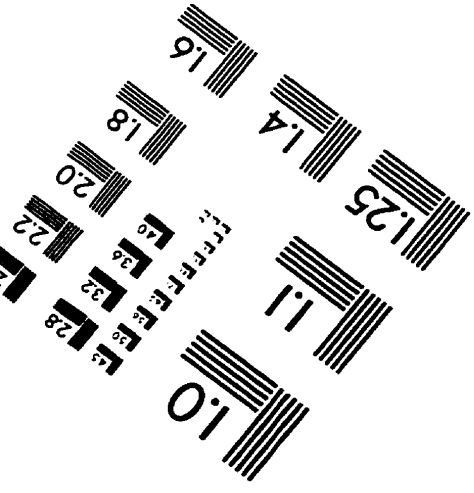
IMAGE EVALUATION TEST TARGET (QA-3)

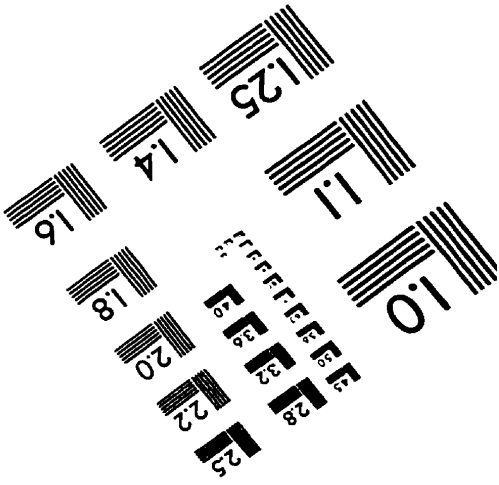












APPLIED IMAGE, Inc
1653 East Main Street
Rochester, NY 14609 USA
Phone: 716/482-0300
Fax: 716/288-5989