Table of contents

Definition of terms 3
1 Introduction 9
2 Motivation 11
2.1 Background 11
2.1.1 Challenges of HCI 11
2.1.2 Designing intelligent user interfaces 12
2.1.3 Design problems on the Web 13
2.2 Goals 15
2.2.1 Analysing, designing and bridging 15
2.2.2 Methodology 16
3 Related research 17
4 Adaptivity 19
4.1 A model for IUI 19
4.1.1 Multimedia analysis 20
4.1.2 Multimedia presentation 20
4.1.3 Automated graphic design 20
4.1.4 Modelling and plan detection 20
4.1.5 Model based approaches 21
4.1.6 Agents 21
4.2 Adaptive systems 22
4.2.1 Classification of adaptive systems 22
4.2.2 Factors and roles 23
4.3 Models add power to adaptive systems 24
4.3.1 The importance of external models 24
4.3.2 Representation 25
4.3.3 Acquisition, maintenance and reasoning 25
4.3.4 ITS, an illustrating example 27
4.4 Adaptive hypermedia 27
4.4.1 A brief history of adaptivity 28
4.4.2 Recommenders - an example of adaptive hypermedia challenges 28
4.4.3 Other approaches to individualised presentations 29
4.4.4 Future challenges 29
4.5 Planning an adaptive hypertext system 29
4.5.1 Building a domain model 30
4.5.2 User modelling 30
4.5.3 Generating adaptive presentations 31
5 It's all about models 33
5.1 Extracting information 33
5.1.1 Hypothesis of this thesis 33
5.1.2 Some possible approaches 34
5.1.3 What is HTML? 35
5.1.4 Using tags in the conceptualization process 36
5.1.5 Importance of the elements 37
5.1.6 Using IR-techniques within important elements 41
5.1.7 Domain specific list 42
5.1.8 Other ways to extract information 43

5.2 A domain model in the horizon 43
5.2.1 The sets of candidate concepts 43
5.2.2 Values separate the candidates 45
5.2.3 Their fate is in the hands of Heuristics 46
5.2.4 Ensuring flexibility 51
5.2.5 Relationship types 52
5.2.6 Completing the domain model 59
5.3 Generating adaptive presentations 60
5.3.1 Structure of the user model 60
5.3.2 Adaptation Model 62
5.3.3 Various forms of adaptations 63
6 Implementation and evaluation 65
6.1 Realising the domain model 65
6.1.1 A walk through the conceptualization processes 65
6.1.2 Seaming up the domain model 68
6.2 Possibilities for the AHS 75
6.2.1 Picturing a simple user model 75
6.2.2 Adaptation rules 77
6.3 Evaluation 80
6.3.1 Optimizing and justifying the performance 80
6.3.2 An ocean of relations 83
6.3.3 Document concepts vs. element concepts 83
6.3.4 Clean up the mess before the guests pay you a visit 85
7 Conclusion 87
References 89
Appendix A - Source documents 93
Appendix B - Results from conceptualization 97
Appendix C - Results from PROLOG rules 103
Appendix D - PROLOG implementation 105
Appendix E - Implementation of conceptualizer 109

Definition of terms

Adaptation model

A model of how an *adaptive system* should behave.

Adaptive Hypertext System

A system which offer to tailor hypertext *documents* dynamically according to the knowledge, needs and preferences of each *user*.

Adaptive System

A system that adapts to the user. It is more general than an adaptive hypertext system.

AE

Adaptation engine responsible for performing the adaptation according to rules in the *adaptation model*.

AI

The field of Artificial Intelligence.

AHS

See *Adaptive Hypertext System*

AM

See *adaptation model*.

Author

A person that have written and structured a collection of hypertext *documents*.

Browser

The environment software in which the *user* is exploring *hypertext documents*.

C

Programming language.

Candidate concept

In the process of *conceptualization*, all *terms* surviving *lexical analysis* are candidates to be chosen as the *concept*.

Concept

A concept is a descriptor of the content of some particular chunk of information, or *knowledge source*. In this thesis, the concepts are abstracted from their respective knowledge sources. See also *document concept* and *element concept*.

Conceptualization

The process of abstracting *concepts* from the *documents* and *elements* of the domain.

DFA

A Deterministic Finite Automaton object, or DFA, represents words in a network of interconnected characters. Whole words can be found by following paths of linked characters.

DM

See *Domain model*.

Document

By a document, we think of a collection of formatted information, e.g. including *text*, graphics, tables and the like. For this thesis, we assume the documents are published on the *Web*, if not otherwise noted.

Document concept

Concept abstracted from a *document*.

Domain

A domain is a collection of *documents*, where the documents are somehow related around one or more abstract, higher level *concepts*.

Domain model

A domain model is a description of how the domain looks like regarding content and structure. The domain model is built from *concepts* and *relations*.

DSL

Domain specific list used by the *system* in the process of *conceptualization*, that is a list of *terms* that the *author* has decided to be important for the *domain*.

Element

A *document* consists of elements of different types. HTML provides easy integration of elements like graphics, *text*, tables, lists and the like. In HTML, the form of the elements are determined by the browser by use of *tags*.

Element concept

Concept abstracted from an *element*.

Emphasizer element

Common notion for *elements* surrounded by the <B>, <I>, <EM>, <STRONG>, <U> and <CITE> tags. Such elements get emphasised by the browser, so they are likely to attract the attention of the *user*.

HCI

The field of Human Computer Interaction aims at achieving user friendly *systems*. HCI concerns both the design of user *interfaces* and tools that support the developement and implementation of interfaces.

Heuristic

Rules based on observations from a small set of data. We use two types of Heuristics, both for finding *concepts* and for finding *relations*. Each Heuristics employ a *value* that, when the Heuristic decides to fire on a *term*, is added to the total *score* of the term.

High quality

When we speak of a *domain model* of high quality, we think of a resulting model that would please the *author* when compared to his view of the *domain*.

HTML

The Hyper Text Markup Language (HTML) is the language in which most *documents* on the *Web* is written today. HTML uses mark up *tags* to specify the formatting of documents.

Hyper reference

A link between two *documents* on the *Web* provides the *user* the option to jump from the first to the other when selecting the link.

Hypertext

Documents written in the *HTML* language.

Information source

IR-techniques produce different sets of terms. These techniques are referred to as information sources. *DSL* and *lexical analysis* are examples of information sources.

Interface

The interface is what is between the *user* and the *system*.

Internet

A network of networks, providing several services like e-mail, news, the *Web*, etc.

IR-techniques

Well known methods from the field of Information Retrieval that basically process *text*.

IUI

The field of Intelligent User Interfaces (IUI) constitute the intersection of *AI* and *HCI*, and concerns how to enhance the usability, simulating intelligent behaviour in the dialogue with the user.

Knowledge source

Elements or *documents* filled with content in *HTML*, we call knowledge sources.

Lexical analysis (LA)

The process of transforming a stream of characters into a list of lower case *terms* or “tokens”. When combined with stopword removal, the idea is that any text will be transformed into a list of terms where the most frequent words in the english language, are removed.

Models

Models describe something so that the system understands it. In this thesis, models for *domain* and *users* are represented in terms of *concepts* and *relations*.

PHP

Programming language suitable for server side programming on the Web.

PROLOG

The declarative programming language PROLOG (programming in logic) is often used when *AI* is involved.

Relation

Relates *concepts*. There are different relationship types, we use parental, prerequisites, deeper explanations and synonyms.

Score

When *conceptualising* some text, the *terms* are given *values* from different *Heuristical* rules. The score is the sum of all these values.

Sets on the form C_i

The union of all sets produced by the *information sources*. The result constitute the *candidates* that compete for presidency as the *concept*.

Sets on the form K_{DOC}

Holds all the *element concepts* and the *document concept* for a specific *document*.

Sets on the form $S_{P\ LA}$

$S_{P\ LA}$, $S_{P\ DSL}$, $S_{P\ EMP}$ and $S_{P\ TF}$ are sets of *terms* that are produced when a paragraph *element* is exposed to the different *information sources*.

Sets on the form R_{type}

All relations in the domain of the specified type

Stemming

The automatic fusion of *term* variants into one common stem.

Stopword removal

See *lexical analysis*.

System

With system, we mean the software running on a computer.

Tag

The *author* can use tags to mark up and format *documents*. In order to make a sentence appear in bold, the <B> tag is placed in front of the sentence, and it is ended with an </B> tag.

Term

A term is a word that has gone through the process of *lexical analysis*.

Term frequency

Indicates how many times a *term* occurs in a *text*.

Text

A collection of words.

TF

See *term frequency.*

UM

See *user model*.

User

The user is a person that is using a computer system. The user is communicating with the system through the *interface*. Users have different skills and can be broadly classified as novices, intermediates or experts. Novices are assumed to have no knowledge at all in the domain at the time of starting the learning process, intermediates have the basic knowledge, while experts are assumed to have deeper knowledge.

User model

A model of preferences and the knowledge held by a user.

Value

Attached to the *Heuristics* are values, which are numbers to be given to each *candidate concept* that leads the corresponding Heuristic to fire.

Web

The biggest service on the *Internet* is the World Wide Web. People often mistake the Web for the Internet, but the two are not the same.

1 INTRODUCTION

The merging of many existing networks and the use of the standard HTTP protocol for exchanging data on the Internet has opened for new ways of presenting information to its users. Distinct from e.g. database systems, the information system called the World Wide Web (or “Web” for short) are not structured and users often feel overwhelmed with information when navigating the virtual space at hand. Moreover, most online documents are static in nature, written once and for the average reader. Adaptive systems promise a new interactive user experience by accounting for the knowledge and preferences of each individual user or groups of users and tailor presentations accordingly. The process of turning existing documents into dynamic presentations is an expensive affair since it requires the system to learn the knowledge of both the users and the domain. In this thesis the goal is to identify methods for how the computer can assist a domain expert in the construction of a domain model of *high quality* for an adaptive hypertext system, requiring as little effort as possible from the expert. A domain model can be constructed in many ways. Due to different methods and various approaches, some models might represent the domain knowledge better than others. With “a domain model of high quality” we therefore mean that the model not only correctly captures the domain knowledge with respect to concepts and relations, but also describes it very well if judged by an human expert.

The thesis is organised as follows: Motivated by an increased need for building intelligent systems that can tailor presentations on the Web, we define the main goals of this research in chapter 2. Thereafter, related research exemplify some existing systems and important work from the field in chapter 3. Of the systems built by today, a characteristic is that all operate in closed, restricted domains. Chapter 4 explains why it is difficult to do otherwise, and furthermore places adaptive hypertext systems within the frames of intelligent user interfaces in order to prepare the reader of important terminology and knowledge from the field necessary to understand before moving on. Next, the identification of Heuristics is central in chapter 5 and provides the basis for ensuring quality to the adaptation phase. Finally, in chapter 6, we develop a prototype implementing the ideas, and evaluate the work. We also sketch how possible adaptation variants can be realised fairly easily when a model of the domain is completed.

2 MOTIVATION

Throughout the last decade, several studies in the field have shown attempts to make systems that fit individual users better. In an Internet context, adaptive hypertext systems are named as promising.

2.1 Background

With his MEMEX, Vannevar Bush tried to beat the information overload he was facing in 1945 [*Bush45*]. His ideas of storing and linking microfilm documents are the origin of the hypertext idea, which have developed to the collection of documents linked together in a non-linear manner today known as the World Wide Web [*Berners-Lee96*]. The information overload problem has become much larger than the generation of Bush ever imagined, and despite a dynamic conduct in terms of a continually changing document collection, the Web is in many ways a very static medium regarding the content presented to each user.

2.1.1 Challenges of HCI

From being a tool for the scientist or expert user only, the computer turned into common property a couple of decades ago. As the number of people that used a computer increased, so did the need for more user friendly systems. The society today is flooded with information and is commonly referred to as “the information society”. From a user point of view, computers are tools that provide services like money transactions, entertainment, information development and exchange. In order to fulfil their services, it is often required that the users have certain skills in how to use the systems. Unless being trained before interaction, the fellow man might therefore face troubles when interacting with computer systems. From the user point of view, one should, ideally, not need to have any technical knowledge on computers in order to survive in the digitized world.

An user interface constitute the intersection between the user and the software. In the area of Human Computer Interaction (HCI), the user is in focus. The field concerns the design of user interfaces, mental models to help the programmer in the design phase, and tools to support the development and implementation of interfaces. Schneiderman points out that due to the possibly broad spectrum of usage situations, the developers are forced to go for designs that benefit all users or the average user. The final product is not only constrained to user demands, but also to the technology available. Thus, for the user, the individual satisfaction of a piece of software may vary a lot [*Schneiderman00*]. Due to the great challenge when designing for a broad audience, several techniques have evolved during the past two decades in order to support developers to meet the goal of *usability.*

Usability concerns how well a system can be used. If a system can be easily learned, is efficient in use, its functionality easy to remember, its appearance pleasant to the user and the offered services are regarded trustable, the user is more likely to feel satisfied. E.g. command based systems that have no graphical user interface might please some users, but the vast majority will probably feel more comfortable with a graphical user interface, or GUI, using windows, icons, menus and pointing manoeuvres in order to navigate and act. Even though the graphical environments of today are somewhat more user friendly than their text-based predecessors, users still feel it challenging to learn new systems and programs. In order to write a document, a novice user must first learn how to navigate in the operating system and how start and use a text editor and its functionalities - which may indeed not seem too obvious. Thereafter, the typing begins and the user eventually discovers that it is not easy to instruct the computer on what to do. All details on which procedures to follow in order to initiate an action are likely to interrupt or confuse the user, hence removing focus from the task at hand, i.e. writing the content of the document.

According to Nielsen, the notion of usability is difficult, yet important, to realise [*Nielsen92*]. Moreover, the design and implementation of user interfaces is a complex matter. There is no silver bullet to make the design and implementation easier [*Myers94*], and many problems face designers, only but a few are mentioned here:

- designers have difficulties thinking like users do
- the tasks and domains in which to operate can be complex
- iterative design is difficult, but necessary
- the design process is creative rather than mechanised due to lack of theory and methodology
- there is a huge span in user knowledge, and the cost of making many differing versions of the software often is too expensive.

Nielsen further proposes an usability engineering *lifecycle* of design. In order to obtain successful user interfaces, the process of usability engineering should not only consider a larger context and analyse the user needs and characteristics *before* the design stage, but also collect user feedback *after* the release date. In other words, the result should be regarded as a prototype for the next version of the software.

2.1.2 Designing intelligent user interfaces

The difficulties involved in creating an interface that is easy to understand, yet flexible for the user, have inspired the growth of another field, namely one that adds intelligence to the system. The field of intelligent user interfaces (IUI) is an effort to solve the problems of usability, thereby increasing the user satisfaction of the interactive experience. Combining techniques from artificial intelligence (AI) with HCI yields interesting possibilities, including the ability for the system to reason on what the user really wants, adapt its behaviour to each individual user, provide context sensitive help, understand inaccurate user input and generate presentations on the fly according to individual preferences or the knowledge level of each user.

As an example, an intelligent interface to an image processing program could monitor the user's actions and then try to infer the user's goals. Thereafter, it would offer to complete the work for the user. Say, if the user resized a picture, saved it, and then repeated the action for another picture, the intelligent system would expect the user to repeat the same procedure on other pictures as well, and hence it should offer the user to automatically complete the task of resizing and saving a series of pictures. Identification of patterns from user behaviour is a simple example of intelligent behaviour. A wide range of potential application areas exist, however. Fine, if IUI is such a good thing with its main goal to meet the user in new, more intelligent ways - then why don't existing software apply such systems?

In order to improve efficiency and naturalness of the interaction, an IUI must represent, reason and act on models of users, domain, task, discourse and media [*Maybury+98*]. In short, the extended model needed for IUI development along with the difficulties involved in applying artificial intelligence techniques, makes the process of designing an intelligent user interface a lot more complex than that of traditional user interfaces. Chapter 4 contains a survey of some subordinate fields of IUI.

2.1.3 Design problems on the Web

Hypertext provides information in a non-linear manner (e.g. as opposed to the predefined curriculum of textbooks) which means the user can decide what to visit next and reach related information through visiting linked documents. The links is a structure of paths, and the paths available offer flexibility to the user, as illustrated in Figure 2–1. For a domain on a specific topic, notice that since readers might jump in on a document from somewhere else, it is not even certain that the user has followed one of the paths as outlined by the author [*Berners-Lee95*]. In his style guide for online hypertext, Berners-Lee emphasises that what is natural as links for one visitor may seem redundant for others [*Berners-Lee95*]. Due to the navigational freedom of hypertext environments and the vast amount of information available, users often feel a sense of getting lost in hyperspace, that is they are confused about where they are, or where the information they seek is located. In general, publishing information online does not automatically make that sort of information more accessible.

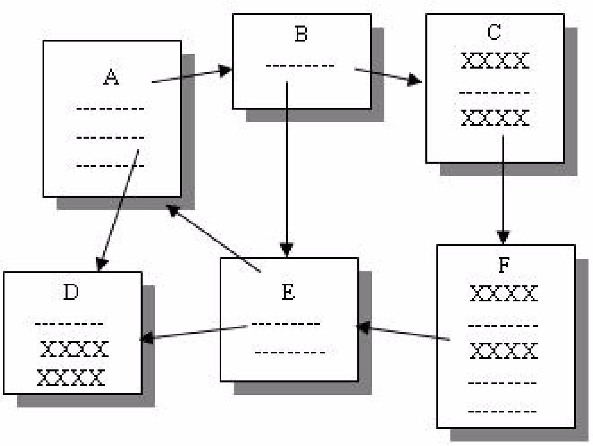


Figure 2–1: A typical hypertext graph structure. Some users might choose to jump directly from A to D, whereas others prefer to follow the path A to B to E to D or even the entire path A to B to C to F to E to D.

The implicit ordering of a document concerns the organisation of information within the document. The author of one domain does not have to know about the content of another, nor what the user knows in advance of interaction, and therefore a document essentially consists of the information the author finds most natural to group.

With the growth of the Internet, the usability and design problems have become particularly challenging. The huge span in user interest, level of knowledge, motivation and experience, makes it very hard to publish online material that is suitable for each particular user using conventional methods. The field of Adaptive Hypermedia, and in particular Adaptive Hypertext, promises to overcome such problems by offering hypertext presentations that are automatically tailored to each particular user. For instance, a domain on the Web offering articles on “Astronomy” may expect very different visitors ranging from the open-minded child, the novice student, the retired woman pursuing the hobby of star watching, to the Doctor of Astronomy. Obviously, what is comprehensible for the expert may be very difficult for the child, and what is made fit the novice is annoyingly trivial to a more skilled person. When accessing a document that contains difficult information, the novice should therefore be presented the concepts necessary to understand first. Likewise, an adaptive hypertext system should not present all the information with which the expert is already familiar, but trim it to a suitable level. Moreover, the system could deliver a wide range of presentational forms based on individual preferences, i.e. graphics, text, tabular information, animated sequences and the like.

As with IUI, models are needed in order to make adaptive hypertext systems, in particular models of the domain knowledge and the user knowledge. The number of adaptive (hypertext) systems is increasing, but it is, however, considered time consuming to integrate existing web-pages into an adaptive environment since adaptivity requires that the system has knowledge of the content of each document in the domain, and this process is often plotted into the system's knowledge base by hand.

The challenging task of integrating adaptive behaviour into an Internet setting is of ever more interest. We therefore feel the need to decrease the time required to model the knowledge of the domain. With the basis that it is possible to, at least semi-automatically, *process* existing hypertext documents and extract the information within in such a way that a system can apply the resulting model so as to offer adaptivity to the end user, this thesis explores how.

2.2 Goals

According to Kobsa, there are two main problems with traditional hypertext [*Kobsa+94*]. First, the user often finds it difficult to orientate in large virtual spaces and therefore easily gets lost during the interaction. Second, it can be hard for novices to understand difficult concepts, since regular documents are written for the average audience.

2.2.1 Analysing, designing and bridging

The idea that the user should decide what to visit next through browsing has both positive and negative sides. In a traditional hypertextual context this facility provides freedom to the user, whereas in the adaptive world, it acts as a limitation. In advance of the interaction the user does not know exactly where in the hyperspace each particular chunk of knowledge exists, and in order to get to the information sought, the user has to rely solely on the descriptive names of the links, menus or sitemaps as organised by the author[1]. Furthermore, in situations where some concepts are assumed to be known prior to visiting a document, the information presented could be difficult to understand if the user lacks knowledge on these concepts.

If a system could both learn the content and the informational properties of a domain, and infer what each user knows at a particular time, it would be able to redefine the predefined (implicit or explicit) curriculum of the domain for each user at run-time. In order to prepare a domain for such an adaptive system, a model of the domain must be constructed and understood by the system. Moreover, the better and more automated the construction of the domain model, the less the threshold for an author to go for an adaptive solution while trusting that accurate adaptations will result. Based on the discussion so far, we define the main goal of this thesis as to design an adaptive hypertext system that can work behind the scenes and tailor the documents of a domain according to the knowledge of each

[1] A sitemap shows the structure of the domain so that the user can more easily get an overview of how it is organized and jump directly to the area of interest.

individual user, doing so through an extensive analysis of existing HTML-documents without requiring the author to rewrite and reorganize new documents. A system would perform at best if the domain model is as complete as possible [*Davis+93*]. We therefore mainly concentrate on ensuring quality to the representation of the domain knowledge.

Table 2–1: The main goal of this thesis

The main goal of this thesis is to design an adaptive hypertext system in order to bridge the gap between the domain knowledge and user knowledge, and building the models needed in the system through an analysis of existing hypertext documents of the domain.

As will come clear throughout this thesis, an adaptive hypertext system has many target application areas. E.g. Schank stresses the need for 1-1 learning situations [*Schank95*]. As with the astronomy-example, electronic education often suffers from a lack of individual considerations. In an educational context, a teacher would benefit if offered means to turn an existing static course into dynamic presentations tailored to each student with very little effort required. In addition to the adaptive generations of content, the system could help students in planning how to browse the domain, or give advice along the road.

2.2.2 Methodology

Computer Science have historical bonds to mathematics, science and engineering. According to Denning these roots are the basis for the three paradigms of Computer Science, namely those of *theory*, *experimentation* and *design* [*Denning99*]. Furthermore, Denning divides Computer Science as a discipline into several subareas, each of which has activities in each of the three paradigms. The areas named range from Algorithms, Programming, Architecture, Operating Systems, Databases/Information Retrieval and Software Engineering, to Artificial Intelligence, Graphics, HCI, Computational Science and Bioinformatics.

Whereas the paradigm of theory concerns the construction of frameworks, the paradigm of experimentation explores and tests models of systems. The paradigm of design focuses on how to build computer systems that work in given application domains. Theoreticians often aim at deep, comprehensive analyses around formal models. Experimenters often use prototyping to quickly construct models of systems. Designers build systems according to accurate specifications which satisfy their customers.

This study belongs within the intersection of the paradigms of experimentation and design. Aiming at semi-automatically building a domain model from a collection of hypertext documents, we first experiment on how the use of certain techniques can help in the process, then validate the experiments through empirical prototyping, followed by a discussion of the results which is the basis for designing an adaptive hypertext system.

3 RELATED RESEARCH

Hypertext documents tend to overwhelm the reader with too much information, or an inappropriate level of detail is presented. Several studies and systems propose solutions to these problems.

In his survey, McTear mentions two systems. METADOC uses a technique called “stretchtext” where classifications of users and concepts are used to vary the amount of detail presented. The HYPERFLEX system can guide the user to information judged to be relevant by recommending topics based on preferences, goals and needs of different users. Matrices are used to link topics in a document and to link topics with nodes representing particular user goals. The system learns through user feedback, by adjusting the weights of the links [*McTear93*].

The system KN-AHS achieves adaptivity by using the shell system BGP-MS. The user may ask about more information related to hotwords, and the data presented reflects what the system believes the user knows. The self-explanatory user interface stimulates user navigation. Knowledge about the user is both based on an initial interview with the user and by deducing what the user knows based on navigational behaviour. [*Kobsa+94*].

Three systems are exemplified by Kules: AVANTI is a system that customizes web pages about a metropolitan area for different users (tourists, handicapped, elderly, residents). INTERBOOK is an advanced WWW application and supports incremental learning. In a web-environment the HTML model and the HTTP protocols limit the details about user actions. Therefore in order to infer what the user know, the system keeps track of what the user has seen. Finally, ORIMUHS is a context-sensitive help system and employ sophisticated user models [*Kules00*].

Even though simple user models are able to represent all the necessary knowledge to achieve curriculum sequencing and adaptive guidance, the knowledge state of a user in www-based learning systems is complicated to maintain. Weber et. al present a solution using a combination of an overlay model and an episodic user model [*Weber+97*]. The episodic learner model (ELM) stores knowledge about the user in terms of a collection of episodes. In our work, we take a quite similar approach to the structure of the knowledge based tutoring system ELM-ART II, whose steps are:

1. Translating text to small sections of units/HTML-code associated with concepts to be learned.
2. Building a conceptual network with links among related concepts. When a page is visited, the corresponding node in the network is updated. Dynamic slots are stored with the learner model for each user and make it possible for the system to guide the user optimally through the domain. By marking concepts of a unit as known an inference process (possible recursive) that marks all prerequisites to

this unit as inferred, is started. This corresponds to the curriculum sequencing and adaptive guidance noted above.

3. Recording all interactions of the learner (the student) in an individual learner model
4. Using traffic lights visible to the user during surfing as a metaphor for annotating links, should reflect the information in the user model.
5. Dealing with inconsistent knowledge by means of tests.
6. Incorporating means for the user to re-use the code of previously analysed examples and to easily navigate an optimal learning path by clicking a *next* button. This feature also helps the user from getting lost in the hyperspace.

This approach has several advantages: it provides a selection of examples that best puzzles the present learning situation, it is suited for diagnosing solutions to problems, and it gives individualized help.

ELM-ART II differs from our work in that our focus is on developing means to semi-automatically associate concepts of high quality with each HTML unit and building a conceptual network representing the domain. Like us, Ashish et. al use formatting information in Web pages to "hypothesize the underlying structure" in order to provide integrated access to multiple Web sources in a particular domain. The sources are parsed for sections and subsections relying on heuristics for font size and indentation spaces. [*Ashish+97*]. The AHM system uses a model where the documents explain the concepts they are linked to and the links are assigned values that indicate the level of difficulty [*da Silva+98*].

We believe that once the domain model is built, adaptive variations can be made fairly easily. For example, in their WHURLE system [*Moore+01*], Moore et. al follow the track of Ted Nelson's vision of transclusion by including smaller pieces into a document led by a lesson plan and the user profile.

4 ADAPTIVITY

Developing an adaptive user interface (AUI) requires an interface that can be adapted, a user model and a strategy for how the adaptation should take place. An overview of the field intelligent user interfaces (IUI) is briefly introduced in section 4.1. Within this context, adaptive systems are more deeply examined in section 4.2. For adaptive systems, the importance of the underlying models can not be underestimated, and are discussed in section 4.3. With this, a foundation for understanding adaptive hypermedia, whose strategies are brought to light in section 4.4, is made. Finally, section 4.5 outlines an architecture for an adaptive hypertext system (AHS) to help illustrate the main points of this thesis.

4.1 A model for IUI

Traditionally, interface models accounted for presentation, dialogue and application. With intelligent user interfaces an extended model is needed, including input analysis, management of the interaction and generation of the output [*Maybury+98*]. An interesting observation is that the intelligence in the input, output and interaction processes gets provided through the use of explicit models of user, task, media, domain and discourse. Figure 4–1 gives an overview of a generic model underlying most intelligent user interfaces. Important sub-fields of IUI are exemplified below.

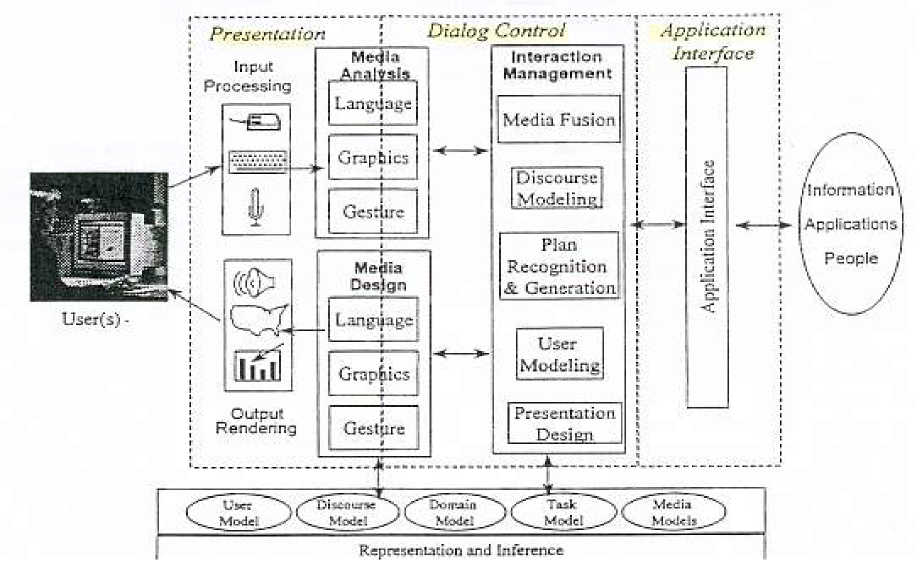


Figure 4–1: Overview of the field of IUI

4.1.1 Multimedia analysis

The field of multimedia analysis supports intelligent processing of multimodal input. The information from all the input sources are interpreted and merged together into one integrated meaning, as shown in the “interaction management” and “media analysis” areas of Figure 4–1. The goal is to let the user *communicate* with the machine instead of only using it, while accepting possibly ambiguous input. Koons describes a prototype operating in the blocks world which integrates and interprets simultaneous input from speech, gaze and gestures using a frame-based method [*Koons+93*].

4.1.2 Multimedia presentation

Multimodal presentation systems use many media in parallel in addition to exploiting the strong sides of each medium. The generic problem when using multimedia in an intelligent setting concerns how the computer can analyse and construct multimedia presentations on the fly. The process of generating output is related to the context, task and user expertise. Selecting the content, allocating and realizing the media and performing layout are interdependent processes. Their underlying knowledge sources are of great importance [*Arens+93*], as also depicted in Figure 4–1. The knowledge-based presentation system WIP generates multimodal presentations by means of an incremental planning process while reasoning about the task and context [*Wahlster+93*].

4.1.3 Automated graphic design

Designing every possible data and presentation situation is an ineffective, comprehensive task that often requires the developer to be an expert. An illustration usually has a communicative intent and gets interpreted in some way by the receiver. Hence, the goals of automated graphic design include letting the system decide how to generate the graphical presentations and from the user’s point of view to remove the possible ambiguity between intended and interpreted presentations. Since the design process needs to be tailored to the context, task and user, it relies upon the use of models. The IBIS system makes use of a generate-and-test approach with a goal-driven search process. If a solution is not satisfactory, the system backtracks so that the illustration can be regenerated. Formalizing the intention of a communication reduces the ambiguity of presentations [*Seligman+91*].

4.1.4 Modelling and plan detection

Typical tasks of intelligent systems like planning explanations, answering questions based on prior discourse and supporting interruption, rely upon the use of underlying models. A user model contains information about users. A discourse model has descriptions of the history, syntax, semantics and pragmatics of the dialogue between the user and the system. UCEgo is a consultation system that corrects the misconceptions of a user or provides needed information that is not explicitly asked for. Gaps in knowledge are discovered through reasoning about the user model e.g. whenever changes in environment or internal state occur. The UCEgo agent needs to be both autonomous and do rational planning to make intelligent initiatives. Goals depend on context, and the central problem for UCEgo concerns how to detect new goals when they are not stated by a human planner [*Chin91*].

4.1.5 Model based approaches

Model based user interfaces constitute a different approach to intelligent user interfaces by allowing the designer to describe a model consisting of facts rather than using large procedural programs. The goals comprise the reduction of the time and expertise required to create user interfaces, the identification of reusable components, and the construction of extensible models in an easy, comprehensible way while maintaining as much of the expressiveness as possible. Model based development has advantages over traditional user interface toolkits and UIMS systems. Dialog control is separated from the application code and the designer is blessed with more powerful design tools. Modifying the behaviour of an interface only requires to change the model instead of reprogramming a certain section. Merging the best aspects from the UIDE and HUMANOID systems, the Mastermind project constitutes a step towards a complete model supporting the entire design-cycle [*Nechest+93*]. The overall goal of Mastermind is to generate automated and animated help facilities, as in UIDE [*Foley+88*], [*Sukaviriya+93*], and use the design models to “map low-level user gestures onto high-level semantics” [*Möller*]. The prototyping stage is easier because conceptualization is regarded as a search in a space of alternative designs and explicit models ensure consistency between task and design.

4.1.6 Agents

Among the arguments in favour of an agent-based approach to intelligent user interfaces, are the need for distributed computing and the limitations of direct manipulation [*Schneiderman+97*]. Direct manipulation and software agents are complementary rather than mutually exclusive. Agents are more convenient in settings with complex environments, difficult tasks, a dynamic network of people and information, or relatively naïve users.

Tveit observes that the most common classification scheme distinguish weak from strong agencies [*Tveit01*]. In a weak notion, agent attributes are:

- Autonomy - agents decide for themselves what to do and when to do it
- Interactivity - they are willing to work in concert with other agents when requested to
- Reactivity - they are able to sense and act on the environment
- Proactivity - they take initiative

In a strong notion, the following attributes add to the previous list:

- Veracity - their actions can be trusted by a person and they do what they are told

- Mobility - they move around from one place to another whenever necessary
- Rationality - they perform their actions in an optimal manner

According to Bradshaw [*Bradshaw97*], one may either look at the agent as an ascription made by a person in terms of what they are, or as a description of its attributes (i.e. a list of what they do). He points out that communication with humans should take place in a non-symbolic, natural language-like way and that the agent should be using models to infer new knowledge. In short, its overall task is to adapt to its environment and learn from experience over time.

Agents fit complex software systems well, since they can be viewed as organised sub-systems that facilitate decomposition and interaction [*Jennings99*]. Furthermore, they may play various roles for different people and situations, i.e. as personal assistants, intelligent user interface managers, agents behind the scenes, performing agent-to-agent communication etc. Acting as personal assistants, agents become more effective as they learn the preferences, habits and interests of a user. How do they acquire sufficient competence of their users, and do users actually trust the help offered? A knowledge-based approach to the problem makes use of domain specific background knowledge about both application and the user [*Maes94*]. Maes argues that an alternative approach that relies on machine learning techniques may be more convenient if different users use different strategies and habits in the interaction with the application. Given only a minimum of background knowledge, the agent has to search actively for information about the user and his tasks. This is done either by monitoring the user i.e. by searching for repetitive actions, through user feedback, training examples provided explicitly from the user or through the interaction with other agents.

4.2 Adaptive systems

Presenting easy, efficient and effective interfaces is the main goal of adaptation. [*Malinowski+92*]. It is difficult to write software that will fit millions of users perfectly. Nobody learns a system completely but uses different parts of it and shares some common basic knowledge about its functionality. Adaptive systems change this paradigm by turning use time into a different kind of design time by adjusting the interface according to the user’s skills, knowledge and preferences. In order to achieve adaptivity, underlying models of both user and task are essential, as well as the separating the user interface from the application [*Fischer00*].

4.2.1 Classification of adaptive systems

When working with computers, users need to adjust the interface according to their own needs and preferences, goals, tasks and contexts. For the system to say the right thing at the right time in the right way implies reducing the information overload and adapting the presentation to the relevant task, knowledge and experience of the user. An AUI supports this process in more or less sophisticated ways, while a static interface doesn’t. Malinowski et. al describe a taxonomy that places adaptive systems in the context of intelligent user interfaces: Roles of an intelligent interface are fulfilled by the integration of an AUI, an intelligent help system and an intelligent tutoring system. These roles comprise adapting to the needs of the user, provide context sensitive help, and supporting the user of the system [*Malinowski+92*]. The taxonomy used is based on four stages of the adaptation process, namely initiation of the adaptation, proposal of possible changes, decision of actions to be taken and execution of the selections. The degree of adaptation depends on whether the user or the system performs each of the stages. As an example, a system is called self-adaptive if it performs all of the above stages itself.

Furthermore two groups of adaptation are distinguished: adaptation of communication and adaptation of functionality. The first group includes systems that provide context sensitive help, like UIDE [*Sukaviriya+93*]. The second covers the automation of tasks and generation of new complex functions, offering a solution to an important goal of intelligent systems, namely to let the computer carry out the routine tasks and allow the user to perform the creative ones. At the syntactic level, adaptation may yield counting the number of interaction steps, while a higher-level adaptation accounts for goals and tasks of the user as a basis for achieving functional adaptation.

4.2.2 Factors and roles

It is important to decide when interaction should occur, what information to use and how to present it on the screen. The needs of users or groups of users must be considered before the system is built. At use time, adaptation can happen continuously by comparing the situational changes to the user's needs, but also on junctures (predefined critical situations), on special occasions or on user requests. Adaptation implies a certain risk, e.g. situations where the user and the system are trying to adapt to each other and thus never reach upon an agreed interface [*Malinowski+92*]. The adaptation process might also confuse the user if not done carefully. Fischer stresses that in an adaptive setting little or no effort is required from the user, possibly resulting in loss of control [*Fischer00*]. In adaptable systems, however, the user is regarded to know its tasks best and should therefore make changes to the functionality by setting preferences. This requires the user to learn about the existence of, and how to use the adaptation component.

The environment is an important factor when designing intelligent user interfaces. Most systems behave intelligently only in their original surroundings - with changes, the performance degrades. An adaptive system, however, gains its power by reacting to a changing environment. One way to defer the design is by incorporating different variants into the system and let triggers activate the set of design choices. Hence measurements for evaluating the benefits of the design are important for the adaptation process, whose requirements are:

- a theory which relate user behaviour to user interface needs
- access to what the user does
- models of e.g. task and user
- a flexibility in the user interface to accommodate new design variants
- an agent to make this design choice

Rautenbach uses a simple game for classifying adaptive systems and proposes a two level architecture for adaptation.

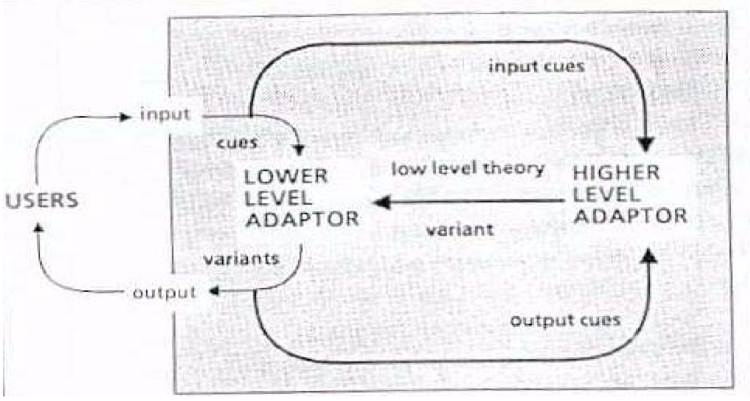


Figure 4–2: Two-level architecture for adaptation

In the model in Figure 4–2 a higher level adaptor identifies major changes and chooses the best design-variant, while at the lower level, the focus is on adapting the interface according to the user's needs. This separation of modeller and introspector is convenient [*Rautenbach+90*].

4.3 Models add power to adaptive systems

An AUI is generated at run-time, meeting the demand that interfaces to complex systems should be able to adapt to different users. In particular, user modelling is a key term for the provision of adapted services, and covers the process of gathering relevant information about each user. This knowledge source is essential for the dialogue behaviour of the system and for reasoning about the user. In this section models, and especially user models, are explored in the context of adaptive systems.

4.3.1 The importance of external models

Making models explicit to the system extends flexibility, but these sources may in some systems be implicitly contained in the code, and furthermore distributed or centralised [*Malinowski+92*]. Modelling components should maintain the models, e.g. by building the model incrementally, maintaining its content, providing consistency, and supplementing other components of the system with information about the user or the dialogue. The XTRA system uses external components in order to assist the user in filling out tax-forms [*Wahlster91*].

4.3.2 Representation

According to Davis et. al, a knowledge representation is a fragmentary theory of intelligent reasoning [*Davis+93*]. Despite its incompleteness, adaptable behaviour does require intelligent selection of content, and in order to achieve it, the choice of representation becomes important.

A user model can represent individual users or classes of users. The dimension ranges from individual user models, through stereotypes, to canonical models. Individual models may be cost-expensive with respect to maintenance, but they provide more flexibility to the system. Canonical models do, on the other hand, characterise abstract, typical users. In between, stereotypes are clusters of characteristics and have proven useful for building powerful models of users when no, or only a small amount of information, is available [*McTear93*]. Rich presents the system GRUNDY that acts as a librarian recommending books based on a dialogue with the user [*Rich79*]. Two kinds of information is required for enabling an effective use of stereotypes in GRUNDY. First, a set of *facets* associated with values will in sum characterise the user. Secondly, *triggers* signal when a stereotype is appropriate and should be activated. The information in the stereotypes is probabilistic, and therefore constitute <attribute, value, rating> triples. The user model, or user synopsis (USS), is built from information of actions, stereotypes and the user. Since the system must justify its own information, the USS consists of a set of <attribute, value, rating, justification> quadruples and is used for guiding the rest of the system. Inheritance plays an important role for stereotypes in order to deduce information.

While the stereotypes in GRUNDY rely on linear parameters with simple numeric scales, other techniques for constructing user models exist. If the system should represent the user's knowledge, goals, plans etc., a more expressive scheme, like concept-based representations, is needed. [*McTear93*]. In FRONTMIND, a bayesian network is used [*Kobsa01*]. Other schemes can be based on logic, inference rules, frames, production rules and connectionist (neural) networks. In the SiteIF project the user models are represented by semantic networks where every node and relation have weights in order to represent different levels of interest for the user [*Stefani+99*]. Kules suggests some guidelines that should be considered when constructing user models for adaptive systems, focusing on the importance of embedding the philosophy “know thy user” into the system [*Kules00*]. The user should also be aware of the existence of, and understand the user model, possibly being allowed to adjust its attributes. Fink et. al stresses privacy issues and an open dialogue with the user in their AVANTI system. In additional to technical solutions regarding security, users can choose the sort of modelling to be used [*Fink+97*].

4.3.3 Acquisition, maintenance and reasoning

In GRUNDY, learning happens through the modification of the stereotypes. In general, updating user models is important when interacting with a user over time. The values of the above attributes may be either explicitly captured by prompting the user for information (user driven acquisition), or implicitly during the course of dialogue (system driven acquisition). In the first the system plays a passive role. Stereotypes might be used when information about the user is limited, or as a supplement to other methods. Implicit acquisition, on the other hand, is more dynamic and requires rules of inference and a way of handling conflicting information [*McTear93*]. The analysis engine is essential to the system as a means for deriving new facts about the user, and next potential steps can be suggested.

Even though the information contained in a user model varies according to the application, situation and the sort of modelling used, a typical user model needs maintenance on the following attributes [*Kules00*]:

- User preferences, interests, attitudes and goals
- Skills of the user (concerning both domain and system)
- Interaction history
- Stereotypes, if present

Connectionist networks can more easily handle inexact information and incremental acquisition by assigning each node with energy levels (which are spread to connected nodes in the network) and allow the weights of the nodes to gradually evolve over time.

How to model the user’s knowledge and beliefs is important for an adaptive system. Plan recognition helps inferring new tasks and is a source for further information about the adaptation. In order to provide context-sensitive feedback, plans can be recognised by monitoring or reasoning about the user *[Malinowski+92].* The KNOME-system infers what the user knows about UNIX, providing different answers to users with different levels of expertise. McTear emphasise that models tend to be incomplete and inconsistent. The information provided by the user is more likely to be true than information based on the user's stereotype. The more specific information should therefore override the more generic information if the properties are inherited. Methods for combining or adjusting numerical values are also part of the maintenance process [*McTear93*].

Dynamic models are closely related to adaptive systems, but require methods for resolving conflicts. According to Kobsa, the shift from traditional shell systems towards less demanding domains like user-tailored web sites, made complex user modelling redundant, i.e. other aspects yield significance:

- quick adaptation should be based on short initial interaction
- companies can integrate their own methods or third-party tools
- heavy work should be distributed
- mechanisms to recover in case of system breakdown
- inconsistencies and faults in the models must be avoided

New services like predicting the future actions of a user based on trends among similar users, demand support for privacy policies and explicit representations of the patterns inferred. Systems like PERSONALIZATION SERVER and GROP LENS, provide many benefits: easy access among applications on user information, methods that can be applied for model protection, easy integration of complementary information from different sources, and centralising the user model servers in order to relieve the clients from the user modelling tasks. Serving many applications at the time is solved by allowing the user modelling system to communicate with the application through inter process communication [*Kobsa01*].

4.3.4 ITS, an illustrating example

Hypertext documents tend to overwhelm the reader with information or present an inappropriate level of detail. User modelling helps an adaptive system to avoid presenting information that is already known to the user. Goals of intelligent tutoring systems (ITS) are curriculum sequencing and interactive problem solving support. These goals reflect the need for models in adaptive systems well. The first concerns the order in which new knowledge should be learned. In textbooks the author has predefined the curriculum of the learning path in advance of the interaction. Such a static organisation assumes an average learner and does not take into account individual preferences. Electronic textbooks take user freedom a step further since they allow for a more random-like surfing through the text by choosing links. An adaptive system should, in addition to this freedom, "give hints as to what pages will be most suitable for visiting next" [*Weber+97*]. In other words, the user should not have to work way through new pages that offer knowledge with which he or she is already familiar. Browsers that annotate visited links come short in comparing the content of a document with the user knowledge. Representing the individual user knowledge in corresponding user models is essential for adapting the presentation distinctively for each user. The second goal concerns interactive problem solving support and can be facilitated through model tracing, using techniques where the system e.g. monitors the user during problem solving and gives advice when it discovers that the path followed will lead to an error.

In ITS, individual student models representing the student's knowledge of a domain, are subject to frequent changes and therefore have to be updated continuously. One way to achieve adaptivity in this setting is by inferring which concepts are learned, and compare each student model with an ideal model constructed in advance of the interaction. The result is that new information can be customised according to what is most useful and understandable for each student. Combined with hypertext and a Web environment, this yields new opportunities for net-based teaching, i.e. replacing the traditional learning situation with a non-linear course tailored to each student.

4.4 Adaptive hypermedia

Conventional hypermedia applications offer navigational freedom, but rely on a strong authoring model which assumes that the author "knows best" [*Bodner+00*]. Adaptive hypermedia systems attempt to overcome these problems of orientation and comprehension, particularly necessary in a Web environment. As noted before, an adaptive system uses explicit models to achieve its goals often founded on intelligent technologies for user modelling and adaptation.

4.4.1 A brief history of adaptivity

Brusilovsky presents the state of the art in one of his surveys [*Brusilovsky01*]. The research field of adaptive hypermedia can be traced back to the early 1990s and is a result of the two somewhat older parent fields of Hypertext and User Modelling. The year 1996 is a milestone in the "adaptive world" because of the explosive growth of the World Wide Web and the accumulation of experience in the field. Before 1996, mostly laboratory systems were built to demonstrate ideas, whereas systems born after 1996 demonstrate real world settings. Currently, three major technologies exist, namely adaptive content selection, adaptive navigation support and adaptive presentation:

1. **Adaptive content selection:** Such systems select and prioritise the information resulting from a query according to what is assumed to be most relevant for the user.
2. **Adaptive navigation support:** Studies have shown that manipulating link anchors can increase the navigational speed and prevent the user from feeling lost in hyperspace, but also that only certain strategies work [*Höök+99*]. Approaches include directed guidance, link hiding, link sorting, link removal, link annotation and map adaptation.
3. **Adaptive presentation:** Systems that have means to present the content dynamically or adaptively belong to this category. A widespread strategy is to conditionally show, hide, highlight or dim fragments, whereas other systems tailor new, adaptive documents by comparing a user model to an overlay domain model. Presentations according to the needs of each individual visitor is often referred to as *customization*, whereas *transformation* improves the presentations by identifying interaction patterns from all visitors. Finally, *content based adaptation* organises material based on content and apply for ITS and the like.

Whereas the first generation of adaptive hypermedia concentrated on modelling user knowledge and goals serving adaptive navigation support and adaptive presentation, the second focused on adaptive recommendation systems based on modelling user interests. These systems monitor the browsing activity and try to deduce the goals and interests of the user. If successful they can present a set of relevant links. Nowadays, a third, mobile generation adds to the system models of context like location, time, bandwidth and platform, hence improving functionality so as to adapt to the user situation as well [*Brusilovsky+02*].

4.4.2 Recommenders - an example of adaptive hypermedia challenges

It is important to distinguish recommenders working in a closed information space from those working with the whole Internet. Today search engines make use of techniques from information retrieval research. A similar (yet far more powerful) adaptive hypermedia system needs to learn about the structure and content of nodes by analysing the documents and turn them into a corresponding closed hyperspace. Learning about the structure of documents requires working within a closed space. According to Brusilovsky, there are two ways to close an open hyperspace [*Brusilovsky01*]:

1. Select the most relevant links by analysing a few steps ahead of the present browsing point of a single user.
2. Learn about the documents by collecting browsing data from a community of users.

The goal of the process is for the system to understand hypertext-documents and links without use of a human indexer, and obtain documents indexed corresponding to the user's goals, knowledge and background.

4.4.3 Other approaches to individualised presentations

Next door to adaptive hypermedia systems we find the field of dynamic hypertext which uses natural language techniques in order to fuse information and move away from the strong authoring model offered by traditional hypertext. Dynamic hypertext unifies querying and browsing, thereby avoiding the need of switching between search mode and browse mode. Another benefit is that dynamic links don't get broken. Bodner et. al claim that systems implementing the idea work best for collections with coherent vocabulary and well written text [*Bodner+00*]. Both adaptive hypertext and dynamic hypertext tailor documents to the user. The latter contrasts adaptive approaches since there are no existing hypertext documents before the user requests them [*Milosavljevic+98*]. In other words, dynamic hypertext systems move further than adaptive hypertext systems, thereby overcoming the problem of committing to some pre-written segments.

4.4.4 Future challenges

Adaptive systems provide dynamic adaptation through possibly implicit acquisition and hence little or no effort is required from the user who may not even know about the existence of user models. With the shift from expert users to relatively naïve inexperienced users in the Web environment, complex systems providing adaptivity at satisfactory levels while preserving the need of the user feeling in control are required [*Schneiderman+97*]. Fischer emphasises the need of separating user modelling from task modelling. Privacy should be maintained and misuse of the models avoided. These aspects challenge many commercial strategies found on the World Wide Web today [*Fischer00*].

4.5 Planning an adaptive hypertext system

According to Wu et. al, most adaptive hypertext system architectures depend on a domain model (DM), a user model (UM) and an adaptation model (AM) [*Wu+01*]. In this thesis we focus on ensuring quality to the construction of the domain model of an AHS, whose overall strategy is introduced in this section. Some components are based on ideas from ITS.

In order for adaptation to take place in a sophisticated way, the AHS needs to build a model of the domain, maintain individual user models, and generate adaptive documents by reasoning on what the user should be presented next.

4.5.1 Building a domain model

A domain model must represent important information in order to express the domain knowledge. As shown in Figure 4–3, we plan to first parse each HTML-document in order to extract as much information as possible, aiming at automatically extracting proper concepts and relations of high quality. Rules should assist the selection of concepts that describe the knowledge of the documents and their sections, and help identify relations among the concepts. DM is incrementally built as documents are exposed to analysis, but since it is regarded a difficult task to determine the meaning of a Web page reliably [*Perkowitz+00*], one might assume the results to be far from perfect. Somewhat similar to ITS, whose modelling concern how a domain expert would represent the knowledge to be taught to the learner [*Beck+96*], manual evaluation and adjustment from someone skilled in the domain is probably needed in order to perfect the DM. For our approach the interesting question is whether we can realise a domain model without asking too much of the author.

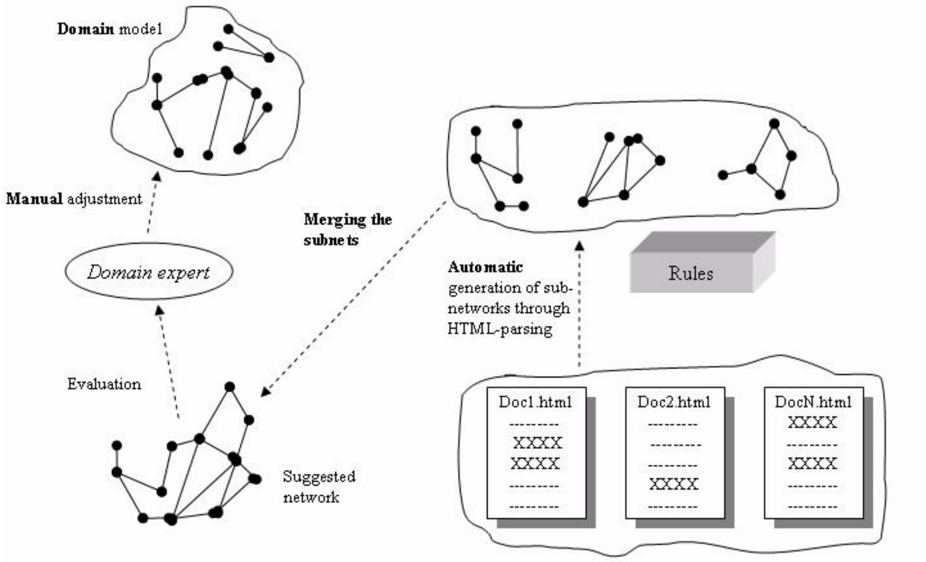


Figure 4–3: The process of making a domain model is semi-automatical, which means an expert should evaluate the proposed model and adjust it manually.

4.5.2 User modelling

By monitoring the user during interaction a conceptual user model that represents the user knowledge in terms of concepts is incrementally built. This information reflects which concepts the system believes the user presently knows. In ITS, one way to represent student models is through an overlay model [*Beck+96*]. The same assumption is made in our system so that the knowledge of the student is regarded a subset of that of the expert, illustrated in Figure 4–4. We note that due to student misconceptions such a model may be imperfect.

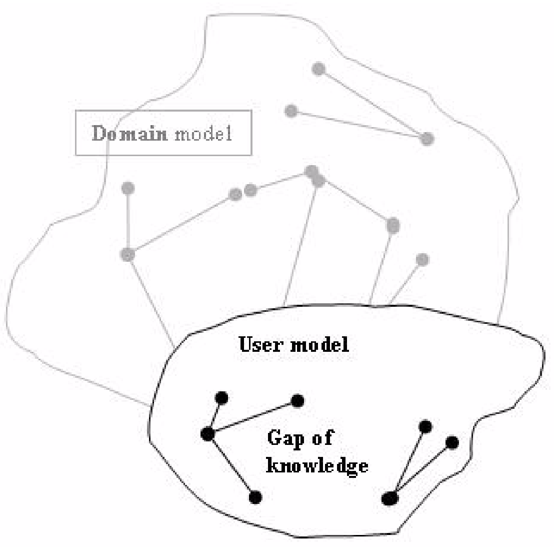


Figure 4–4: The user model is a subset of the domain model.

4.5.3 Generating adaptive presentations

For adaptation to take place, the system needs to modify documents so that they allow for the present knowledge of each user based on information about what the user already knows. The system could perform a real-time analysis of the contents of the document requested for, but there are two problems to overcome. First, going through the processes of parsing, conceptualising and building a network representation again would likely result in an imperfect temporal representation of that document. Secondly, as mentioned before, it is expensive to perform such an analysis in a real-time environment and requiring the user to wait too long would be unacceptable. As a solution, adaptive documents of high quality can be generated simply by comparing the incomplete user model to the perfect domain model. Given that the domain has been analysed in advance, this approach is both inexpensive and flexible. Each presentation would therefore consist of information represented by the concepts selected by the system, accounting for individual preferences of the users. Hence in order to bridge the gap of knowledge, the system would need to reason on which concepts the user has insufficient expertise and tailor adaptive documents with the required information. The steps of this process is visualised in Figure 4–5.

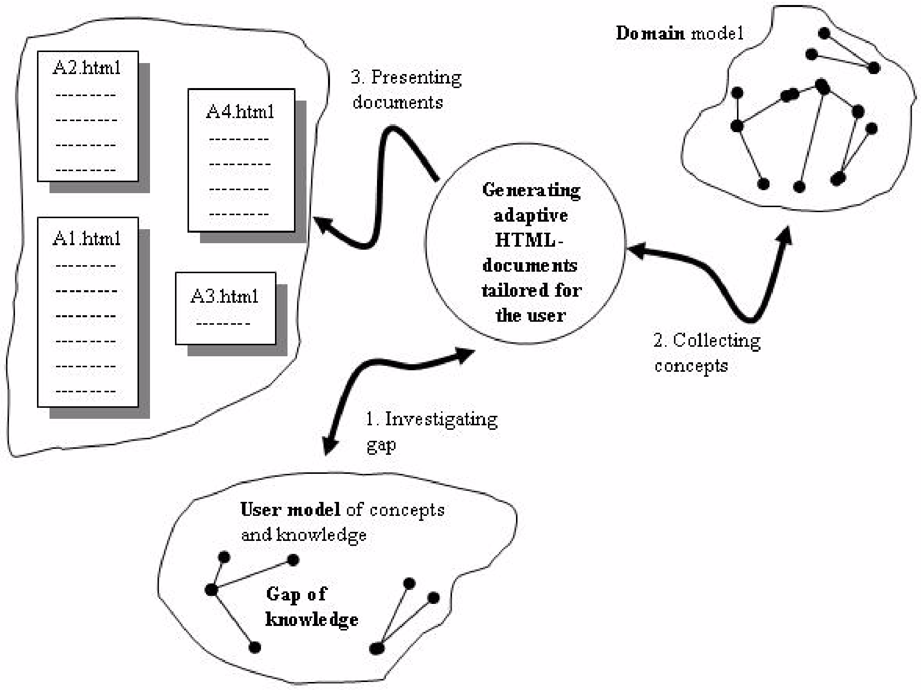


Figure 4–5: Adaptive generations are based on both DM and UM.

5 It's all about models

De Bra et. al outline that an AHS should perform its tasks without requiring any sort of programming by the authors [De Bra+99]. Therefore, the first task towards adaptive behaviour is for the system to learn the content of a domain. Section 5.1 investigates how to develop suitable methods for extracting information from the domain sources in an hypertext context. Next, in section 5.2 we find that Heuristics can ensure quality to the process of abstracting concepts, and we try to identify relations among the concepts found. From this basis, section 5.3 raises issues on how a knowledge representation of the domain can be constructed and how the generation of adaptive documents for each user can be fulfilled.

5.1 Extracting information

It is important to separate the information gathering from the information processing. According to the overall plan outlined in the previous chapter, a crucial step towards a domain model concerns the identification and modelling of the content of each document. After stating the hypothesis of our work, this section discusses the nature of concepts. Also, generic document characteristics and proper techniques for extracting candidate concepts are identified.

5.1.1 Hypothesis of this thesis

As mentioned in the introduction, some models might represent the domain knowledge better than others, and as models play important roles in adaptive systems, the performance of the system depends on the quality of the models. Figure 5–1 illustrates the different roles played in the adaptive system explaining the notion of *high quality*: The author has a mental model of some knowledge, and expresses this knowledge as HTML documents. During analysis of the static documents, the adaptive hypertext system tries to generate a domain model which agrees with the author's view of the domain. The domain model and the user models make up the basis from which the system can generate documents adapted to each user. Since the adaptation depends highly on the content and structure of the domain model, we therefore choose to focus on ensuring high quality to the semi-automatic construction of the domain model, and thereafter use the methods found to build an adaptive hypertext system. Remember from the introductory part that with high quality we mean that the system should capture the domain knowledge well with respect to concepts and relations, so that the domain model is in accordance with the author's view of the domain.

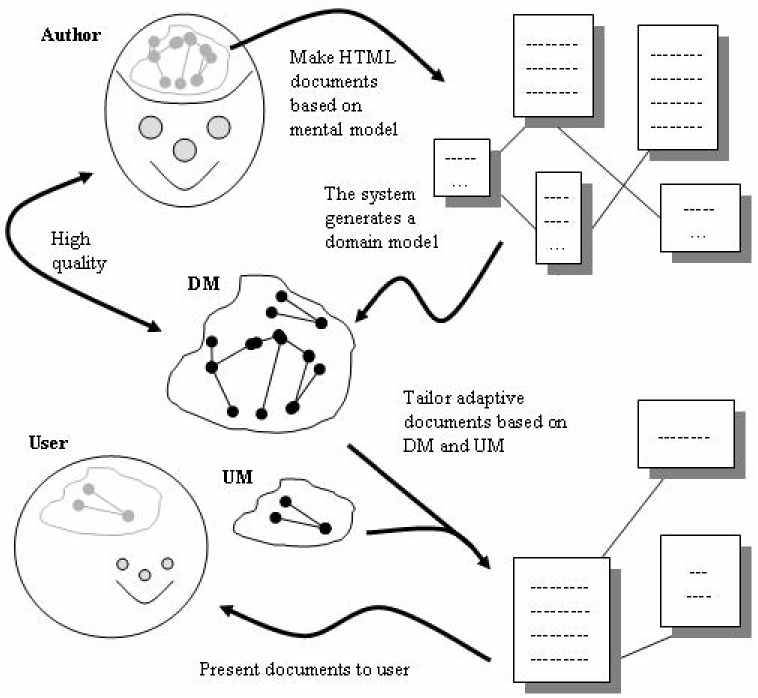

Author
Make HTML documents based on mental model
The system generates a domain model
High quality
DM
Tailor adaptive documents based on DM and UM
User
UM
Present documents to user

Figure 5–1: The concept of high quality explained in terms of the actors.

We hypothesise that the combination of Information Retrieval (IR) techniques and an analysis of document properties and hypertext structure, yields a means to both find and use Heuristical rules for the identification of concepts and relations, and to ensure a resulting domain model of high quality for an adaptive hypertext system.

Table 5–1: Main hypothesis

The combination of *IR-techniques* and an *analysis* of document properties and hypertext structure with respect to content, yields Heuristics that secure the production of an AHS domain model of high quality.

5.1.2 Some possible approaches

How can the content of the documents be abstracted? One way is to require that each document should be structured in a specific manner and tagged with respect to content. Knowing exactly how the documents look like, the adaptive hypertext system can be programmed to perform powerful adaptations. This is clearly a very inflexible approach in that it places a heavy load on the author in the tagging phase. It also conflicts goals of using existing HTML documents as easily as possible, as exemplified in the system KN-AHS, which compose documents based on knowledge about with which hotwords the user is familiar [Kobsa+94]. The hotwords are designed to fit the system in advance of interaction. Systems that manage personalised views of information spaces extend the notion of classical IR-hypermedia systems by not only locating but also organising the information in a way the user wants [Brusilovsky01]. However, running traditional IR-analysis on the documents with vectors representing each section in a document, the adaptive information could be presented by comparing the vector-representation of the document requested for with the user's knowledge represented in a similar vector space. Marinilli et. al propose a case based approach to information filtering that presents HTML documents according to the interests of the users. A filtering component selects relevant documents for each user by means of a vector model. After exposing a text document to stoplist, weighting and separation of words, the resulting array of values (one value for each category) is mapped into stereotypes [Marinilli+99]. Before bringing our approach to light, we refresh on HTML.

5.1.3 What is HTML?

According to the specifications of the World Wide Web Consortium (W3C), HTML (Hyper Text Markup Language) "is the lingua franca for publishing text on the web" [w3]. It is essentially a subset of SGML, but even though a much less complexity in HTML, the language usually offers authors enough flexibility. An HTML-document may consist of elements like text, tables, graphics, sound etc., and these elements are marked up by means of tags. As HTML is fairly easy to learn, anyone can make, structure and format their own Internet pages. The documents may be created either by writing tags and text directly into an HTML-editor or by converting an existing formatted text-document to an HTML-document. In the latter case, a standard text-editor[1] uses advanced techniques like style sheets and the like to extend the expressive power of the HTML-language[2] and make an HTML-document look as similar as possible compared to the original document. A complicating characteristic with such automatic conversion, is that the source of the resulting HTML-document is very difficult to read and edit for a person due to much redundant coding from the automatic process. Programs to "clean up" such auto-generated code, or bad programmed documents, exist, however[3].

An HTML document consists of a header section with meta information and a body section with the text and graphics presented on the screen. The tags mark up the text, so if the author of a document wishes to make a piece of it appear in bold, <B> and </B> tags mark where the bold face-formatting should start and end, respectively. The tags are present in the HTML-file, but only the interpreted result is displayed on the screen by the browser. More advanced features like XML, stylists, Javascript, Java applets and the like can be embedded as necessary to further format and add functionality to the documents.

[1] Text editors range from simple to more complex ones, e.g. Notepad, Microsoft Word, Lotus AmiPro, FrameMaker, etc.

[2] DTDs and style sheets redefines the standard formatting provided by the tags.

[3] The program "Tidy" for Unix is an example of a program that fixes incorrectly tagged documents.

5.1.4 Using tags in the conceptualization process

Brusilovsky emphasises that adaptivity requires knowledge about links and documents, where the documents must be indexed according to the user's goals, knowledge and background [Brusilovsky01]. The process of conceptualization tries to abstract the information of a domain and associate descriptive concepts with this information. For an AHS there is a need to separate the information entities and understand their relative importance as knowledge sources for the user. We therefore question how to identify and conceptualise each chunk of knowledge before constructing the domain model.

There are two interesting observations concerning the nature of online material. First, with the ability to link different documents the author can make a more natural decomposition of large documents into smaller ones, so that the size of an HTML document is on the average relatively small, often limited to knowledge sources dealing with a specific subject. Secondly, writing and formatting a document normally involves emphasising important words, structuring the text through paragraphs, setting up links to related documents, summarising information in tables and bulleted lists, and perhaps of most importance: using descriptive headings. The purpose of formatting is to improve readability and the user’s understanding of the content, and from this we can advantage when aiming for an AHS. Assuming that most concepts, conceptual structures and domain specific terminology appear in the documents of the domain, it should be promising to acquire knowledge from the corresponding nodes [Kietz+00]. Figure 5–2 illustrates how an HTML document is divided into different elements, or sections, in terms of content and presentational form.

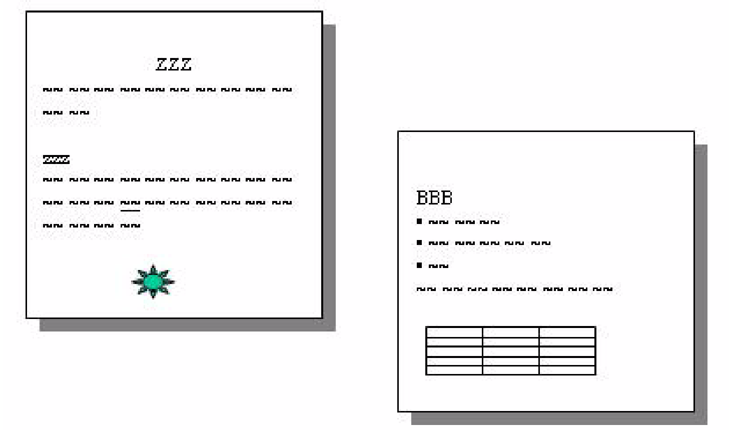


Figure 5–2: Example of HTML documents as they appear for the user on screen, illustrating how the author might organise content into different elements.

In adaptive hypermedia systems, concepts are used both in the domain model and throughout user interaction, and are essential to the system since they are the clues that point to the specific knowledge sources. Davis et. al argues that it is important that a knowledge representation acts well as a medium of communication and expression [Davis+93]. The conceptualization can be fulfilled by picking out the most promising keyword terms from the elements, and concepts can be labelled in a way understandable by a person by using the keywords abstracted. This method therefore agrees with the demand to simplify the manual evaluation and adjustment from a domain expert and facilitates the author's interaction with the domain model. Moreover, users can better control the content of their user models by simply looking at the concept label and sort out which concepts seem familiar or not. In their simplest form, concepts would consist of only one keyword term. By permitting several concurrent terms, or phrases like "Saying the right thing at the right time" when labelling the conceptual states, the problem of how to compare different concepts arises. On the other hand, if several independent terms are allowed for the concept, similarity measures can be applied. Finally, notice the possible ambiguity in that a term can have different meanings in distinct contexts.

De Bra et. al distinguish three kinds of concepts. Atomic concepts are the smallest information units, pages are composed of atomic concepts, and abstract concepts represent larger units of information [De Bra+99]. In a similar fashion, we assume that regarding content, documents are likely to have a context superior to its sections, and likewise, some documents might be superior to other documents. We therefore find it convenient to separate *document concepts* from *element concepts* in the process of building a domain model.

5.1.5 Importance of the elements

Since the formatting tags are included along with the content in the HTML-files, the system can easily search for concepts in elements regarded important. A tag is embraced in brackets and has a name, but not all tags need to be closed, that is, no end-tag is needed to mark the end of the element in order for the browser to interpret the page correctly. Most tags have some optional attributes that can be assigned values specifying the purpose or layout of elements. Table 5–2 presents the most commonly used tags.

Even though different tags are designed for different purposes in HTML, the author is free to use the tags for other purposes, like layout. There is no guarantee that tags are used coherently among different domains or even within a domain.[1] Particularly, in their survey of the history of hypertext, Ashman et. al call attention to studies which have shown that the set of links varies a lot from author to author [Ashman+99].

The diversity of the use of tags complicates the process of selecting which elements are the best indicators of concepts and choosing the correct level of abstraction. In order to extract the meaning from a document based on its elements, we need to find out and analyse how the average author has used tags to mark up documents, or in other words, identify what are the most important tags in terms of conceptualization for a randomly picked document. As expected, this task is complicated. Therefore, and due to the limited scope of this thesis, the following

[1] XML (eXtended Markup Language) is designed with coherence in mind, but since the vast majority of the documents on the web today is HTML-documents, we have chosen to stick to HTML in this research.

discussion is based on a small set of empirical data focusing on the ideal use of tags, the different roles the elements play and potential traps when it comes to extracting concepts based on the elements.

Table 5–2 The most commonly used HTML-tags

Tag name	Short description	Notes	End
<HTML>	Language used is HTML	The document is embraced by this tag	No
<HEAD>	Header information	Not visible to the user, only to the browser	Yes
<TITLE>	A descriptive title of the document	Often used to label the browser window	Yes
<META>	Main purpose is to guide search engines during indexing	The attributes are identified by search engines	No
<BODY>	Indicates the start of the visible portion of a page	Everything contained within the document body is visible to the user	No
<H1>	Heading level 1	Largest font size of the six headings	Yes
<H6>	Heading level 6	Smallest font size of the six headings	Yes
<UL>	Unordered list of LI-elements	Can be nested	Yes
<OL>	Ordered list of LI-elements	Can be nested	Yes
<LI>	List element	Each LI forces a new line	No
<EM>, <I>	Emphasise and Italic, respectively	The <EM> tag usually appears as italic	Yes
<B>, <U>	Bold and Underline, respectively	<U> should not be confused with links	Yes
<FONT>	Provides font formatting	Additional attributes specify size/color	Yes
<P>	Denotes a paragraph	Inserts an empty space before and after	No
<TABLE>	Has rows <TR> and columns <TD>	A table cell can hold any element including another table	Yes
<IMG>	Inserts an image in the document. A description is shown in case the browser has turned images off	<IMG SRC=”name” ALT=”descriptive text goes here”>	No
<!-- -->	Comment	Not interpreted by the browser	Yes
<SCRIPT>	Embeds a script of a specified type	Scripts can be stored in external files	Yes
<A HREF> </A>	Link, i.e. hyper reference	<A HREF =”url”> links to the document with the url specified <A NAME=”location”> creates a target location somewhere *within* a document that can be linked to <A HREF=”url#location”> links to a specific location inside a document	Yes
<BR>	Line break	Inserts a carriage return. No spacing	No
<DIV>	Element used for special purposes	Often contains a class definition	Yes

Meta elements contain information about a document, hidden for the visitor but usable for spiders helping search engines to index Web documents. The three most important meta tags are the description meta tag, the keyword meta tag and the title meta tag. The last one is most commonly used and should identify the content of the document in a fairly wide context. Since the title is a property of the document and not part of the text presented to the user, it can and often is used to label the browser window. Its value is also default as descriptor for both browser history and bookmarked favourite sites. Short and descriptive titles should be used, but there is no guarantee the same title is not used throughout the domain. In general, a drawback in the adaptive setting yields that meta information in the worst case could be equal for every document in a domain. Even worse, this often happens to be true. When creating a bunch of documents an efficient technique is to make a template document with only the structure or form, this acting as the basis for constructing the domain documents in order to ensure the same layout and local structure. Meta information is not visible to the user and the author might therefore more easily forget to change it according to the content of each document.

Headings serve the purpose of describing the content of the sections to follow. Hence headings alone are powerful sources of identifying keywords connected to the distinct elements and the document as a whole. Headings are ordered according to levels ranging from <H1> to <H6>, and by default the font size of <H1> is larger than <H2>, which is larger than <H3> etc., so a natural assumption to make is that the structure of the document agrees with the intended tree-like ordering provided by the HTML specification.

Tags like <B>, <I>, <EM>, <U>, <CITE> and <STRONG> are designed to emphasise text. They often occur inside other elements, and hence serve two purposes: both to outline important keywords or even whole sentences, and to make the text more structured and readable. Even though promising, note that there is no guarantee that emphasizers are used properly or hold important words representative for their parent elements.

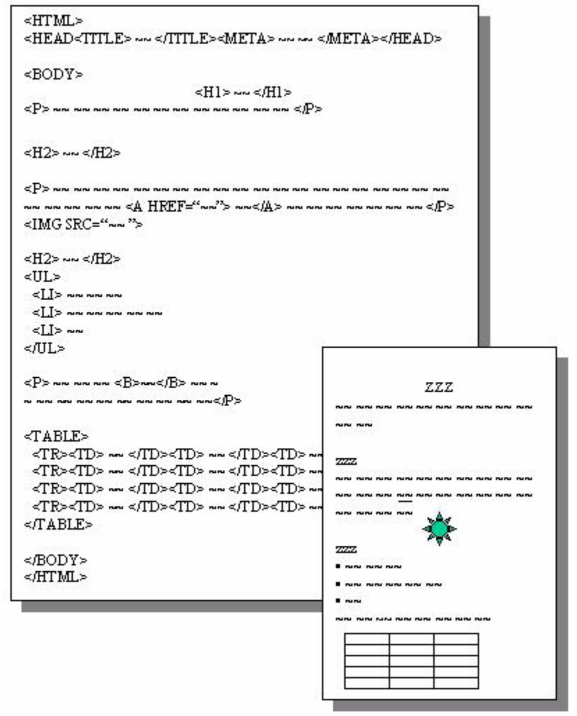
Among indicators of intended groupings are ordered and unordered lists, designated with <OL> and <UL> respectively. Lists consist of several <LI> elements. There is a possibility of nesting lists, that is a <LI> element may actually be replaced by a new list. Another commonly used element for grouping information into paragraphs, is the <P> tag, popular in most documents since it is the easiest way to embody perceived open spaces between textual elements. Other groupings of text are provided by block quotes and tables. The latter is both an interesting and complex matter. <TABLE> can be used to group other HTML elements, summarise information, or even for layout purposes. Many efforts have tried to extract or categorise information into e.g. databases based on analysing table content like in [Cohen00]. It is difficult for the system to find out what purpose a table serves, and hence an analysis of the information exhibited is not too promising.

<A HREF> is the tag indicating a hyper reference, or link, to (somewhere in) a document. Links provide a means to connect related documents, allowing the user to quickly manoeuvre the hyperspace from one place to another. The power of a link is that it can point to anything. Without links, the flexibility of the web disappears. Many styleguides recommend that the text embraced by the link tag should be descriptive of what it is linking to [Berners-Lee95], that is qualified with a clue like “a step-by-step tutorial” rather than “click here”, to allow some people to skip it. What does a link mean to different people? If used properly, even people jumping in on a page from outside the present context would be able to decide whether to explore the link or not. Among common problems are documents

overcrowded with links, and broken links, i.e. links pointing to another destination than intended or to a non-existing destination.

Finally, images should not be forgotten. The phrase “an image says more than a thousand words” is one reason why the web is filled with illustrations. Another reason is that images provide interactive experiences for the user, either as static images or as clickable image maps. The <IMG> tag has an optional ALT attribute that should hold a short textual description of the image. Until recently, this property was needed and widely used since many users instructed their browsers to not show images due to low transfer rates, and authors therefore embedded descriptions in order to allow the users to consider explicitly requesting for the image. Despite increased transfer rates, the ALT-attribute is still widely used today since most browsers show its description whenever the mouse passes over the image. Moreover, images are useful to spice up the content, or for summarising important and difficult subjects in a document. The most commonly used graphical format is jpg- and gif-images.

From the perspective of a the browser software, its task is to interpret the tags in the HTML document in order to present a formatted page to the user. It is our belief that the adaptive hypertext system can make use of the very same tags in order to choose the elements that are important to conceptualise, as illustrated in Figure 5–3. The task to follow concerns how to extract as much information as possible in order to lay the grounds for ensuring quality to the selection of final concepts.



```
<HTML>
<HEAD<TITLE> ~~ </TITLE><META> ~~ ~~ </META></HEAD>

<BODY>
                              <H1> ~~ </H1>
<P> ~~ ~~ ~~ ~~ ~~ ~~ ~~ ~~ ~~ ~~ ~~ ~~ </P>

<H2> ~~ </H2>

<P> ~~ ~~ ~~ ~~ ~~ ~~ ~~ ~~ ~~ ~~ ~~ ~~ ~~ ~~ ~~ ~~ ~~ ~~ ~~ ~~
~~ ~~ ~~ ~~ ~~ ~~ <A HREF="~~"> ~~</A> ~~ ~~ ~~ ~~ ~~ ~~ ~~ </P>
<IMG SRC="~~ ">

<H2> ~~ </H2>
<UL>
  <LI> ~~ ~~ ~~
  <LI> ~~ ~~ ~~ ~~ ~~
  <LI> ~~
</UL>

<P> ~~ ~~ ~~ <B>~~</B> ~~ ~
~ ~~ ~~ ~~ ~~ ~~ ~~ ~~ ~~ ~~</P>

<TABLE>
  <TR><TD> ~~ </TD><TD> ~~ </TD><TD> ~~
  <TR><TD> ~~ </TD><TD> ~~ </TD><TD> ~~
  <TR><TD> ~~ </TD><TD> ~~ </TD><TD> ~~
  <TR><TD> ~~ </TD><TD> ~~ </TD><TD> ~~
</TABLE>

</BODY>
</HTML>
```

Figure 5–3: Some tags indicate which elements are important to conceptualise, while others assist in the process of conceptualization.

5.1.6 Using IR-techniques within important elements

After identifying important features of the elements above, we need methods to further analyse them in order to extract possible concepts. In the field of Information Retrieval (IR), the main goal deals with automated classification and retrieval of unstructured documents [Frakes+92]. IR-techniques such as *lexical analysis*, *stoplists* and *stemming* provide a way to identify the words of each document with high discrimination values, thus forming the basis of building an index. When a query[1] is being executed the most relevant document(s) matching the query can be retrieved.

An IR system seems useless as an adaptive hypertext system since it provides little or no *understanding* of document quality [Berners-Lee96]. The basic IR-techniques, however, may assist in the adaptive hypertext system, since what is needed in the conceptualization phase is a proposal of important keywords. That is the concept candidates are partly a result of a statistical IR-analysis, whereas the selection of final concepts is made by a more intelligently reasoning upon the candidates.

Lexical analysis is the process of transforming a stream of characters into a list of lower case words or tokens, by removing any character that is not a letter nor a digit. This implies that an HTML-document passing through lexical analysis would loose its brackets around the tags, as the brackets are non-letter characters. Clearly the adaptive system would fail if this important information were lost. Performing the IR-analysis on the text embraced with interesting elements only (not the entire HTML-document) solves this problem.

Frequent occurring words[2] from the spoken language have no indexing value. The use of a stoplist may reduce the number of words since stopwords typically account for 20-30 percent of the tokens in an average document. In the stoplist process, every word from the text is checked against the stoplist and eliminated if found there, thus the survivors are more likely to be of importance.

Stemming is the automatic fusion of term variants, so that after stemming the terms “concerned”, “concerning” and “concerns” all get conflated into the stem “concern”. Although stemming may reduce the size of an index by as much as 50%, some information about the terms going through stemming is clearly lost. The Porter-algorithm is a well-known and fairly small implementation of stemming [Frakes+92].

For instance, we performed a test (using the basic IR-techniques) on an element from a randomly picked HTML-document. The results showed that after being

[1] If the user wants to find information on the Internet about a subject, he/she needs to formulate a (boolean) query consisting of key terms.

[2] Van Rijsbergen have developed a stoplist of 250 words that is widely used in IR-analysis [Frakes+92]. Common words occurring in many documents may be for example “time”, “any”, “all”, “into”, “very”, “asking”, “where” etc.

exposed to lexical analysis and stoplist only 133 out of 290 terms remained (i.e. 46% of the total number). Removing duplicates *before* stemming lead to a resulting 96 terms (i.e. 33% of the total number), while removing duplicates *after* stemming reduced the text to 87 terms (i.e. 30% of the total number). Even though about 70% compression was obtained after IR-analysis, the test indicated that these procedures alone were not enough to fully extract a few concepts describing the element. The problem is that one paragraph alone is likely to produce far to many potential competing concepts. A more exhaustive approach is needed.

5.1.7 Domain specific list

A concept of an element could be represented by a vector of all the terms in the element accounting for all the terms in the domain. The conceptual space would then consist of many different vectors so that similar elements would have quite similar vector representations. The alternative approach implemented in this research uses human knowledge to ensure quality with respect to the conceptualization. If the author of a domain wants to prepare a collection of documents for the adaptive system, one might require that the first thing to do is constructing a domain-specific list with keywords regarded as being indicators of concepts or important paragraphs of text. Using such a domain specific list[1] or DSL actively when analysing each document, would help the system to identify possible concepts and important elements and later help in building relations among the concepts proposed. As an example, say that the term “star” is listed in the domain specific list[2] and also found in an element. Then the system could select the word “star” as one strong concept candidate for that element or for the document as a whole. Note that constructing a DSL needs only to be done once for each domain.

The performance of the adaptive hypertext system is related to both the quality and complexity of the DSL, since the idea is for the system to actively be using the list in the process of conceptualising the documents. Obviously the content and size of the DSL would influence the selection of concepts. A thoroughly considered list would therefore increase performance, and, from the view of the end user, make the system act more intelligently. Furthermore, using not only one, but several different lists should add flexibility to the system. Remember the DSL is not only of value before extracting information, but also when monitoring the user and adapting documents on the fly. The influence of several lists is best illustrated in an educational setting where the level of knowledge may vary significant among students due to a variation in individual skills and related knowledge obtained from prior courses. If the students were allowed to modify their own DSL and the system could use lists from related courses, the goal of a presentation tailored to each user seems much closer [3].

[1] The domain specific list can be thought of as an “anti-stop list”, that is a list acting directly opposite to the IR stoplist method.

[2] Allowing for more than one DSL yields a flexible facility to the adaptive hypertext system

[3] This is necessary in the case of documents being analysed on the fly. In our approach, we intend to perform the analysis in advance of the user interaction.

Making models explicit to the system adds both flexibility and system performance with respect to intelligence. The system KN-AHS does not integrate the user-modelling component BGP-MS into the application. This leads to adaptivity in the following senses: many types of information about the user can be represented simultaneously, the user-modelling component can receive and answer questions, and accumulation of knowledge takes place more naturally [Kobsa+94]. Embedding the domain specific list in the domain model should help during the analysis of documents. Using the DSL therefore yields a simple technique that strengthens the strategy of conceptualising a document based on the contents of its elements.

5.1.8 Other ways to extract information

Term frequencies (TF) can be used in order to extract concepts [Kietz+00], based on the assumption that terms that occur often, with the exception of stopwords, are of importance. After counting the TF, the resulting information can be sorted so that the most frequent occurring term (TF_{max}), or all terms with a higher TF than a given threshold ($TF_{treshold}$) can be listed.

So far we have found that a domain consists of different documents linked together, each of which has different elements, and we know some simple techniques that can be used to gain statistical information of the text. Keeping this in mind, the work continues on the development of a strategy to guide the conceptualization.

5.2 A domain model in the horizon

It seems convenient to represent the domain knowledge in terms of concepts and relations among the concepts. Before we explore how to build a domain model in the next section, we formalise the notation of the information as provided by the techniques introduced in the previous, and develop Heuristical rules in order to select the most promising concepts and relations. The rules are the backbone of the construction of the model which in turn is quite critical concerning the system’s ability to perform adaptations. The following discussion is therefore essential to our work.

5.2.1 The sets of candidate concepts

From the discussion in section 5.1.5 “Importance of the elements”, page 37, we see that those elements classified as “important” serve various purposes for both the user and the author. Due to this variety, the system needs to take action according to rules in order to extract concepts. These rules should account for as many aspects as possible, both the global ones concerning the document as a whole, as well as the local ones, like document subsections. Keeping the rules in a rule base yields both power and flexibility to the process of analysing HTML-documents.

New rules may be easily added to the base without affecting other modules of the system. An appealing strategy is to parse each document looking for important elements, extract the information within, and finally, in the search for concepts, analyse it according to rules.

Each method **m** used on an element **i**, produces a set $\mathbf{S}_{im}$ consisting of **n** candidate concept terms $\mathbf{c}_j$ when run on an important element, i.e. $\mathbf{S}_{im} = \{\mathbf{c}_1, \mathbf{c}_{2, \ldots,} \mathbf{c}_n\}$. Therefore we refer to the techniques as information sources from now on. A brief summary of the functionality of each information source is listed in Table 5–3.

Table 5–3: Functionalities of the information sources

Method / Abbreviation	Information source	Functionality
LA	Lexical analysis	Converts the stream of characters in an element to a list of terms, where stopwords are removed
DSL	Domain specific list	Identifies terms from the element that match terms in a domain specific list
TF	Term frequency	Counts number of occurrences for each term in an element
Emp	Emphasizer identification	Lists terms in emphasised elements that occur within an element

Each method produces different sets of terms that all can be more or less suitable as a concept describing the element. In order to visualise, regard the following rather simple element written in HTML:

- `<P>Our solar system is only one of millions of other solar systems. It consists of nine <B>planets</B>, of which the <A HREF=”tellus.html”>earth</A> is the only one with developed life </P>`

Assume a domain specific list (DSL) holding the three terms “Pluto”, “Mars” and “Earth” and a term frequency threshold set to two terms. The list below illustrates the different sets of terms originating from the different information sources. Notice that the terms from the element are stemmed, as are those in the DSL. Finally, since the element is surrounded by a paragraph-tag, we label the sets with a leading “P”. The meaning of the first item in the list is that *the set S of terms from element P when exposed to Lexical Analysis, are “solar”, “system” etc.*

- $S_{P\ LA}$ = {solar, system, million, consist, nine, planet, tellu, earth, develop, life}
- $S_{P\ DSL}$ = {earth}
- $S_{P\ TF}$ = {solar, system}
- $S_{P\ Emp}$ = {planet}

Note that with this notation, it is easy to picture different sets and how their members influence each other.

5.2.2 Values separate the candidates

For an element, the task for the adaptive system is to choose the best concept from the set of all information sources, that is the concept must be chosen from $\mathbf{C}_{\mathrm{i}} = \sum_{\mathrm{m}} \cup \mathbf{S}_{\mathrm{im}}$ Notice that the set notion is also true for the whole document, i.e $\mathbf{C}_{\mathrm{DOC}}$ is a valid set just as $\mathbf{C}_{\mathrm{TABLE}}$, $\mathbf{C}_{\mathrm{IMG}}$ and $\mathbf{C}_{\mathrm{P}}$ are valid sets. Apart from *image* and *link* elements, all elements that go through lexical analysis (and stopword removal, both denoted by LA) have a potentially large set $\mathbf{S}_{\mathrm{i\ LA}}$. The more the number of competing terms, the more difficult the process of ensuring quality to the final selection. Inspired by Sharma, who outlines the power of a strong Heuristics model for developing a user model in the absence of well known methods [Sharma01], we apply a straightforward approach to the conceptualization process of the document and/or each element by *adding values* to each candidate from $\mathbf{C}_{\mathrm{element}}$ based on Heuristical rules. The most important terms can be distinguished from the less important ones simply by judging the final numeric values summed up from all Heuristics, so after all Heuristics have done their job, a set of candidates $\mathbf{c}_{\mathrm{i}}$ ordered by value is the result, where the value of $\mathbf{c}_1 \geq \mathbf{c}_2 \geq \mathbf{c}_3$ etc. To exemplify, say the system produced the candidates “sun”, “rain”, “day” and “wind” for an element, where the candidates had values of 14, 12, 9 and 3, respectively. Then the matter of selecting the concept is simply finding the head of the ordered list, which means that “sun” becomes the concept. The tail of the original list hosts all the candidates that were not selected.

Let us stress the definition of the word *value*. A value is associated with each Heuristic. The terms take on these values as the Heuristics fire, so after the process of analysing an element, all the element terms have various scores. In other words, the *score* of a term is the total of all values added to the term. As an example, say that the following values are associated with three of the Heuristics:

- HeuristicA: positive value of 3
- HeuristicB: positive value of 8
- HeuristicC: negative value of -4

Table 5–4 illustrates the difference between the values of the Heuristics and the total score. Note that the values are fixed, and are used to add up the score of a term. From now on, we refer to the present score of a term as its present value.

Table 5–4: The values as assigned by the Heuristics and the total scores held by the terms

Terms	HeuristicA	HeuristicB	HeuristicC	Total score
agent	Yes	No	No	3
user	No	Yes	Yes	4
model	Yes	Yes	No	11

The following discussion on Heuristics debates how to choose the correct level of abstraction and explores how to guide the process of adding appropriate values to the candidate concepts.

5.2.3 Their fate is in the hands of Heuristics

Three groups of Heuristics were identified. The Heuristics for important elements are listed in Table 5–5 followed by those that focus on aspects for the document as a whole in Table 5–7. An interesting discussion concerning the roles of the elements in total leads to the Heuristics described in Table 5–8.

Consider a fictional piece of text explaining user models with a section debating *user model acquisition*. Within this text the formatted string (written in HTML) `<B>Explicit</B> models gather information by prompting the user` contains the emphasised word “explicit”. The author would prefer the term “acquisition” as the proposed concept, however, so the system needs to account for more than the “emphasizer” Heuristic. Furthermore, when creating the DSL, the author might browse quickly through the pages looking for keywords from headings and emphasizers as a help to create the domain specific list. Higher values should therefore be added to concepts suggested by $\mathbf{S}_{\mathrm{i\ DSL}}$ than those in other sets, as stated in the first group of Heuristics below.

Table 5–5: Some Heuristics used for important elements

HC-1 Terms stated explicitly by some domain expert are very likely to be among the most useful concepts, a fact that suggests assigning high values to members of $\mathbf{S}_{\mathrm{i\ DSL}}$
HC-2 The more a term occurs, the more important it is. The set $\mathbf{S}_{\mathrm{i\ TF}}$ results from counting term frequencies in $\mathbf{S}_{\mathrm{i\ LA}}$. Assigning each **c** with values relative to these frequencies therefore seems promising.
HC-3 Members from $\mathbf{S}_{\mathrm{i\ Emp}}$ can provide quality for the system in the selection process. Even though the author decides selectively which words to emphasise, emphasizers are not always used properly, and therefore the value should not be too dominant.

Interestingly, $\mathbf{S}_{\mathrm{i\ TF}}$ might be quite similar for many documents, if the frequently occurring terms of one document also has a frequent appearance in others. In order to reduce this problem, terms that occur often in many documents could be punished by using a collection frequency and normalising the set [Kietz+00]. Additionally, through the use of stemming the figures are further incremented since terms with the same stem count twice.

Notice the complementary nature of the sets as illustrated in Table 5–6. When their elements coincide, the respective values summed up thus far are incremented or decremented according to Heuristics. The total value, or score of a concept **c** in the set $\mathbf{C}_{\mathrm{element}}$ is therefore the sum of the values calculated from one or more

Heuristical rules, so e.g. if $\mathbf{S}_{\mathrm{i\ LA}} = \{\mathbf{c}_1, \mathbf{c}_2, \mathbf{c}_3, \mathbf{c}_4, \mathbf{c}_5, \mathbf{c}_6, \mathbf{c}_7, \mathbf{c}_8\}$, the score of $\mathbf{c}_5$ is the sum of values defined by Heuristics HC-1 and HC-3.

Table 5–6: The complementary role of the Heuristics

Set name	Set content	Associated Heuristic
$\mathbf{S}_{\mathrm{i\ DSL}}$	$\{\mathbf{c}_2, \mathbf{c}_5\}$	HC-1
$\mathbf{S}_{\mathrm{i\ TF}}$	$\{\mathbf{c}_1, \mathbf{c}_{4,} \mathbf{c}_6, \mathbf{c}_7, \mathbf{c}_8\}$	HC-2
$\mathbf{S}_{\mathrm{i\ Emp}}$	$\{\mathbf{c}_3, \mathbf{c}_5, \mathbf{c}_6\}$	HC-3

The second group of Heuristics focus on aspects for the document as a whole. Since the document can be considered as an element containing children elements (subordinate elements), the information sources (DSL, TF, LA) are valuable and the Heuristics from Table 5–5 can be applied. In addition, the ones listed in Table 5–7 are important contributors to the quality of the Conceptualization of a document.

Table 5–7: Some Heuristics used for the entire document

HC-4 Meta information and the title element ensure quality to the selection since their content most likely are considered thoroughly from the author's side. Relatively high values can therefore be applied in such cases.
HC-5 A value that reflects the characteristics of the element in which a term lives, should be added to the score. The type of element where a candidate concept occurs matters in the selection process, that is some elements are more important than others.
HC-6 If a document **d** is pointed to from another document in the domain, the description of the link element in **d** should be assigned a very high value, based on the assumption that hyper references are of value for the conceptualization of future documents.

Despite the potential quality provided through meta information and the title element, these sources could in the worst case be equal for every document in a domain, and often is, as noted in the discussion in section 5.1.5 "Importance of the elements", page 37, thus misleading the system to choose the same concept for every document. The solution is straightforward and involves comparing meta information in every document with that information found this far. If two documents have the same meta information, then Heuristic HC-4 is cancelled for these documents.

Two very interesting features are implicitly contained in Heuristic HC-5. First, if a candidate concept recurs in several different elements within a document, it is likely to be an important keyword for the entire text. Due to the different roles of the elements, their respective candidates should not count equally in the process of adding up values. In particular, headings at the highest level (normally <H1>) should have higher priority than lower level headings, tables and lists. The second feature e.g. reveals the dual role of external hyper references, i.e. links to other documents. As noted in section 5.1.5, the text embraced by the link tag <A HREF> is often (or at least should be) qualified with a clue indicating what it is pointing to, rather than the often used printing block “click here”. Hence, one of the terms *in the clue* is very likely to be a concept (or at least a very strong candidate concept) of the document *pointed to* by the link. The same term is simultaneously destructive as a candidate for the document at hand[1]. In other words, candidates from hyper reference elements should not be accumulated to the set $\mathbf{C}_{\text{doc}}$ when searching for a document concept, but should be saved for future use, that is only candidates from hyper references *pointing to* the present document should be accumulated to $\mathbf{C}_{\text{doc}}$. Heuristic HC-6 therefore has the implicit assumption that all links in the domain are known in advance of the conceptualization of documents, are represented somewhere (e.g. a matrix or a file) and are used throughout the analysis.

As an example of how Heuristic HC-6 works, assume that the hyper reference `<A HREF=”file3.html”>user</A>` appears in “file1.html”. Then the term `user` is very likely to be the concept of the document “file3.html”, but it should not be the concept of “file1.html”. A high value is added to the score of occurrences of `user` in “file3.html”, and a low value to occurrences of `user` in “file1.html”.

It is interesting to question which rules should fire, that is, are all sets of equal relevance, or should some be omitted under certain conditions? Do the different element types influence the final outcome? Obviously there is a possible conflict between the conceptualization of a document and its elements. By treating the document as one big element enclosed by the <HTML> and </HTML> tags, we see that all sets $\mathbf{C}_{\text{element}}$ from a document is contained in $\mathbf{C}_{\text{doc}}$ but the reverse does not hold, as illustrated in Figure 5–4. The candidate concepts appear as black dots.

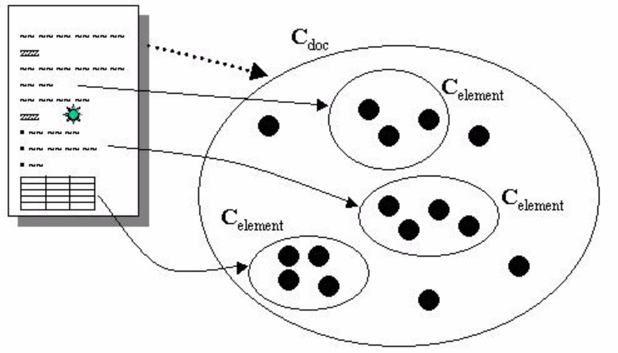

$\mathbf{C}_{\text{doc}}$
$\mathbf{C}_{\text{element}}$
$\mathbf{C}_{\text{element}}$
$\mathbf{C}_{\text{element}}$

Figure 5–4: Candidate concepts of a document belong to different sets

Why is this observation so important? Remember the purpose of the concepts - they should act as descriptors of the content of each particular knowledge source in the domain, providing a basis for building the domain model. Furthermore the

[1] Two documents should not have the same concept according to the requirements of the domain model, but as will be explained in Heuristic HC-7, different *sections* of a document might very well have the same concept due to their complementary roles.

division into document concepts and element concepts was based on the need to acquire knowledge sources at different levels in the adaptation phase. In other words, element concepts should differ from document concepts in order to prevent overlapping information, thus ensuring a better domain model. This conflict is not unique. What happens if the same conceptual description is found for elements of distinct types? Heuristics that solve such problems are listed in Table 5–8 below. Note that most of these Heuristics don't hand out values as did the previous ones (except Heuristic HC-8).

Table 5–8: Some Heuristics used for the ensemble of elements in a document

HC-7 In case the same concept is proposed for two distinct element types, the system can perform more powerful adaptations based on user preferences. Presentational form matters in the adaptive phase, so information on the element types should be preserved.

HC-8 Candidates that equal the document concept found should not be considered when searching for concepts in important elements. In order to prevent their influence, such candidates could be assigned large negative values.

HC-9 The level of abstraction can vary for different documents. Headings indicate the start of new sections and are likely to describe the content that follows.

HC-10 Lists, images and tables should always be considered as single units since they provide variation in the document and often act as complementary sources of information for the surrounding text.

HC-11 Since some elements might be superfluous, an analysis of the document structure should precede that of the elements.

Brusilovsky points out that an adaptive system can choose different types of media with which to present information to the user, according to what is most relevant at the given node (i.e. document) [Brusilovsky01]. Accounting for the fact that different elements act as complementary sources of information for the user provides a means for the system to add variety to the presentation based on context and user preferences, so Heuristic HC-7 suggests that similar conceptual descriptions should be accepted only when the element types differ. Note that this applies for whole documents as well (remember the document-as-HTML-element view), so two documents should not be assigned the same concept. When it comes to the future adaptive generations, the system would be better off if it for elements that equal in terms of both type and proposed conceptual description, tries with another concept. Similarly, candidates that equal the document concept found should not be considered when searching for concepts in the important elements, since they provide no new information to the system. Heuristic HC-8 therefore implies that the search for document concepts should precede that of element concepts, a statement consistent with its fellow Heuristic HC-11.

Heuristic HC-9 claims that the level of abstraction may vary. Moreover, headings should not be regarded as stand-alone elements, as they indicate the start of a new section in the document and are likely to reflect its content well. The scope of a header includes all elements from below it to the next heading at the same level. The problem of choosing the correct granularity concerns which information entities should be regarded as atomic. In the eyes of a natural language expert, the task is to mine text from each sentence. For the adaptive hypertext system, the attempt is to represent the knowledge of a domain in terms of a conceptual domain model: as indicated so far, each document and the different elements are subject to this conceptualization. Some documents are larger than others[1], but if well structured, e.g. assuming a proper use of headings cf. section 5.1.5, the system could try to bring the process of conceptualization to a more fine-grained level. In particular, an examination of which header tags are used should reveal how many levels of the concept hierarchy a single document covers.

Heuristic HC-10 outlines the importance for images, lists and tables to be considered as single units. For instance, in order to conceptualise *list* elements (<OL> or <UL>), it is important to understand how they are used in the document. First, a list can be considered as one entity which implies finding one good candidate concept is desirable. Secondly, an analysis of the content of each <LI> element in a list is somewhat more complex, but provides a means to identify relations between global list concepts and sub concepts within the list. Obviously, if the document only has one element, namely one big list, important information would get lost if the individual <LI> elements were never subject to further analysis. On the other hand, the resulting document representation could get far too complex if all list elements in addition to other elements were analysed. The important and difficult task here is to choose the appropriate level according to the overall document structure. Similar considerations apply for table elements. Finally, due to their value for the user and interactivity, images would probably denote important concepts either for the document or for elements depending of where they occur. The problem of how to conceptualise an image, can be solved by looking in the `ALT` attribute or using the filename of the image.

A summary of the above discussion is exemplified in Figure 5–5. The conceptual descriptions $\mathbf{k}_i$ are document concepts (squares) and element concepts (circles) representing the document and elements of the illustrated types - images, tables, text and lists. Agreeing with Heuristic HC-7, we see that $\mathbf{k}_2$ represents both an image and plain text. The same apply between documents: The system might very

[1] An easy measure for size is file size (remember images are imported by reference in HTML), another more demanding measure is by counting words.

well propose the same conceptual description $\mathbf{k}_4$ for the image element of one document as for the table element of another.

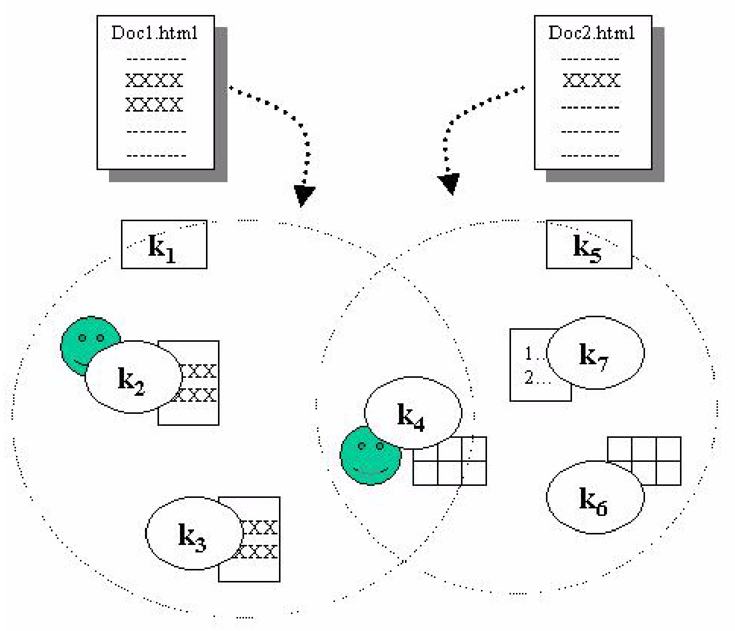


Figure 5–5: Conceptual representation of two documents. Note that an element from the first and one from the second document have been assigned the same concept.

5.2.4 Ensuring flexibility

In order to control the impact from the combination of the different Heuristics, the respective values can be kept in a file and hence adjusted by the domain expert, introducing a means to experiment empirically on the relative importance of the rules. Making a value zero means preventing its rule from influencing the total score of a term. An high negative value would mean that rule to block a candidate from consideration, which would be desirable in cases like Heuristics HC-5 and HC-8. Using ideas from the theory of neural networks (NN), in which values (called weights in NN terminology) are adjusted over time by the system to change performance and knowledge, the adaptive hypertext system could even find means for adjusting the values itself e.g. by tracking the domain expert's rejection of concepts.

Due to the possibly erroneous outcome of the entire conceptualization process and the future manual adjustment, it is important that the system retains all the information found about every element. If a domain expert is dissatisfied with the suggestions, the system can simply propose new promising concepts selected, and it could even explain why it chose particular concepts by logging and displaying the rules fired and their associated values. If the expert keeps on rejecting the suggestions of the system, it could also, for each specific type of element, track which information sources the expert seems to prefer, and further identify patterns in order to modify or develop new heuristical rules, or the system could learn new rules stated explicitly by the expert. This modification is possible only if the rules are kept external to the system and if some sort of rule inference engine exists. In time and throughout interaction, such a system would possibly improve its performance. Hence the adaptation could be taken further to other levels[1], but such issues are outside the scope of this research.

In the above discussion we tried to choose a document concept and some element concepts for each document. The selection tried to capture the best among several candidates by adding values guided by Heuristics. Regardless of the outcome, the result is a set of concepts describing the content or *knowledge* of the original document and its parts. In other words, $\mathbf{K}_{doc} = \{\mathbf{k}_1, \mathbf{k}_2, ..., \mathbf{k}_m\}$ where the selected concepts $\mathbf{k}_i \in \mathbf{C}_{doc} \cup \mathbf{C}_{element}$. The next task is to relate the concepts.

5.2.5 Relationship types

In this section there are two aims. First, we want to find a set **R** of directed relations **r** of various types between the concepts in order to extend the knowledge of the domain model, so $\mathbf{R}_{domain} = \sum_i \cup \mathbf{r}_i$ where $\mathbf{r}_i = (\mathbf{k}_o, \mathbf{k}_p, \textit{relationship\_type})$. This expression should be read “concept $\mathbf{k}_o$ is ‘relationship_type’ to/by concept $\mathbf{k}_p$”. Secondly, we want to discuss the possibilities for automatically extracting these relations, ensuring quality to the process.

Along with all selected concepts, there is a set of candidates **L** (ordered by value) that were not selected, i.e. $\forall\ \mathbf{k}_i\ \exists\ \mathbf{L}_{\mathbf{ki}} = \{\mathbf{c}_a, \mathbf{c}_b, \mathbf{c}_c...\}$, where the value of $\mathbf{c}_a \geq \mathbf{c}_b \geq \mathbf{c}_c$ etc. Repeating one of the examples used in the previous section, say the system produced the candidates “sun”, “rain”, “day” and “wind” for an element, where the candidates had values of 14, 12, 9 and 3, respectively. Then the head of the ordered list is the concept selected, in other words “sun” becomes the concept of the knowledge entity. The tail of the original list hosts all the candidates that were not selected.

For stand-alone individual documents a representation of the knowledge in terms of conceptual names may seem sufficient. However, for a collection of q documents we reveal that in $\mathbf{K}_{domain} = \mathbf{K}_{doc\_1} \cup \mathbf{K}_{doc\_2} \cup \mathbf{...} \cup \mathbf{K}_{doc\_q}$, there obviously are identical items and relations among the elements, reflected through similar concepts and implicit conceptual relations. If we make a peek into the adaptive phase of the interaction, the gap of knowledge in the user model can be bridged by means of selecting appropriate concepts from the domain model and present corresponding knowledge sources to the user. Without representing some sort of connections or relations between the concepts (i.e. the nodes) in the domain model, it is difficult to embody the process of selection in a thoroughly considered plan for a sequence of adaptations.

Many types of relations can be identified in order to extend the domain model and the adaptive performance of the system. In their AHAM system, Wu et. al use the

[1] As noted in chapter 4 “Adaptivity”, page 19, a system that takes initiative to, proposes, selects and executes the adaptation, is called self-adaptive [Malinowski+92].

prerequisite, inhibitor and part-of relations [Wu+01], which illustrate conceptual dependencies in terms of adaptation. Before exploring the characteristics of conceptual hierarchies and prerequisites, we introduce the discussion searching for concepts that act as deeper explanations for others. Again we rely on the use of Heuristics, this time for the *identification* of the different relationship types.

The system should be able to provide the user with many knowledge sources in order to broaden the understanding of a subject. For the user, the most conspicuous relationship type is clickable hyper references (links) provided by `<A HREF>` in HTML, as they relate either documents, elements, or a combination. What does such explicitly stated relations tell the user? Why does the user click on links after all? Is there a reason for the author to embed hyper references in a document?

Heuristic HR-1 states that identifying links provides a means for the system to find deeper explanations of a concept. First, links invite the user to move around in hyperspace. Second, and of importance for deducing this Heuristic, they are designed by the author and likely to point to information of relevance for the present context. That is, the author invites the user to find more information by following the links. Therefore, linked knowledge sources should be represented as relations in the domain model through connecting the corresponding concepts with the *has_deeper_explanation* relationship type, directed from the source to the destination, so the set $\mathbf{R}_{\text{has\_deeper\_explanation}} = \sum_i \cup \mathbf{r}_i$, $\mathbf{r}_i = (\mathbf{k}_o, \mathbf{k}_p, \textit{has\_deeper\_explanation})$. Most likely, such relations would occur between element concepts and document concepts, provided that most links exist within important sections and point to documents. In particular, a link referring to somewhere specific in another document (c.f. `<A HREF=”url#location”>` in Table 5–2), indicates a conceptual relation between the element concept hosting the link and the element it refers to.

Table 5–9: Heuristics for finding deeper explanations

HR-1 Hyper references between two knowledge sources somewhere within the domain are also relations of type *has_deeper_explanation* between the corresponding concepts.
HR-2 External links with destinations outside of the domain are slightly different from internal ones, and the corresponding relations between such knowledge sources should be labelled as *external*.

Let us exemplify HR-1. The relation should be directed from the source of the link to the destination. Assume that the link `<A HREF=”agentsmore.html”>learn more about agents</A>` has a source element whose concept is found to be the term `agent`, and that the document “agentsmore.html” is conceptualised as `reactiv`. Then the relation to be set up is:

- $\mathbf{r} =$ (**agent**, **reactiv**, *has_deeper_explanation*).

Instead of duplicating information that already exists elsewhere (outside the domain), the author may choose to set up external links. Furthermore, as pointed out by Heuristic HR-2, the external links are of less importance than the internal ones discussed above, as their destinations are more likely to go beyond the scope of the domain. Still, such links should broaden the user's understanding, hence the corresponding relations should be labelled *external*, so that $\mathbf{R}_{\text{external}} = \sum_i \cup \mathbf{r}_i$ where $\mathbf{r}_i = (\mathbf{k}_o, \mathbf{k}_p, \textit{external})$. Notice that due to Heuristic HC-6 (page 47), the adaptive hypertext system has a means to identify both relations and concepts in one turn, thus strengthening the participatory role of hyper reference elements in the process of building the domain model. From this we can advantage when searching for a label for the external concept, namely by conceptualising the <A HREF> element and let the winner candidate describe the external concept that the link points to.

So much for explicitly stated references. A document consists of subordinate elements subject to conceptualization, and their contents are likely to be somehow related since they appear in the same document. This kind of implicit hierarchy is not constrained within the limits of single documents, but apply between documents and possibly even between domains, as illustrated in Figure 5–6.

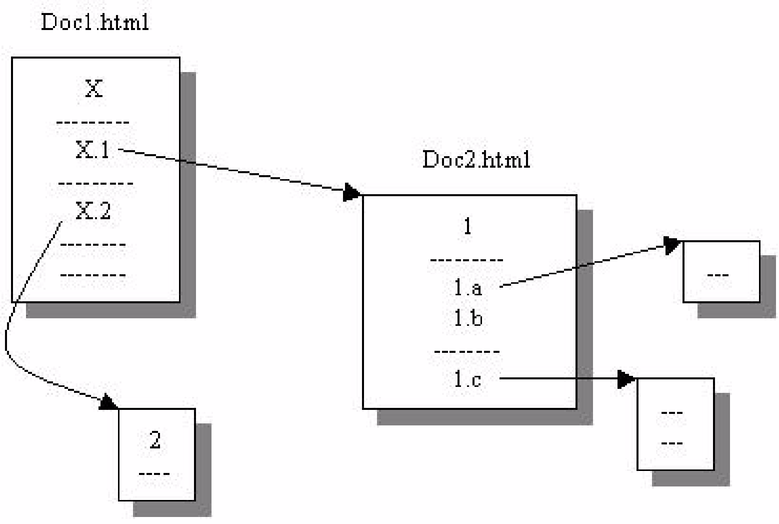


Figure 5–6: The two sections (X.1 and X.2) of the first document are related since they are both in the context of the more general document subject (X). The document hierarchy shows how section X.1 is superior to the entire second document, as illustrated by the directed arrows.

Heuristics for the conceptual hierarchy guide the search for the sets of *parent* and *synonym* relationship types, denoted $\mathbf{R}_{\text{parent}}$ and $\mathbf{R}_{\text{synonym}}$ respectively.

Table 5–10: Heuristics for the conceptual hierarchy

HR-3 All element concepts found within a document are children of the document concept.
HR-4 There is a *parent* relation if a member from a set of candidate concepts equals an already found concept.
HR-5 If two concepts have some joint members from their set of candidates, the concepts are synonymous.

In agreement with the assumption that the concept abstracted from the document has a context superior to the concepts at the element level, Heuristic HR-3 suggests relating the document concept and the corresponding element concepts through the *parent* relationship type, instead of relating all the element concepts found in a document with each other. In other words, such elements are implicitly related through their cohabitation in the same original document, a structure which should be kept intact in the final domain model. Note that due to concept similarity, implicit parental relations between different documents or elements can be caught simply by comparing the conceptual descriptions. For instance, a document conceptualised to `agents` with four elements conceptualised to `reactiv`, `interfac`, `autonom` and `dictionary`, respectively, leads to the relations

- **r** = (**agent**, **reactiv**, *parent*)
- **r** = (**agent**, **interfac**, *parent*)
- **r** = (**agent**, **autonom**, *parent*)
- **r** = (**agent**, **dictionary**, *parent*)

Figure 5–7 illustrates the merging of the conceptual representations of the two documents from Figure 5–6. Note that one element concept of the first document equals the document concept of the second, namely $\mathbf{k}_3$. The *parent* relations correctly capture the hierarchy "$\mathbf{k}_6$ is secondary to $\mathbf{k}_1$" through the relational sequence $\mathbf{r}_2$ to $\mathbf{r}_5$. The system must somehow record that $\mathbf{k}_3$ now both represents a document ("Doc2.html") and an element ("X.2"). Since the document is an element of type `<HTML>`, the same method apply for the situation illustrated in Figure 5–5 (page 51). The implementation specific details are discussed in the next chapter.

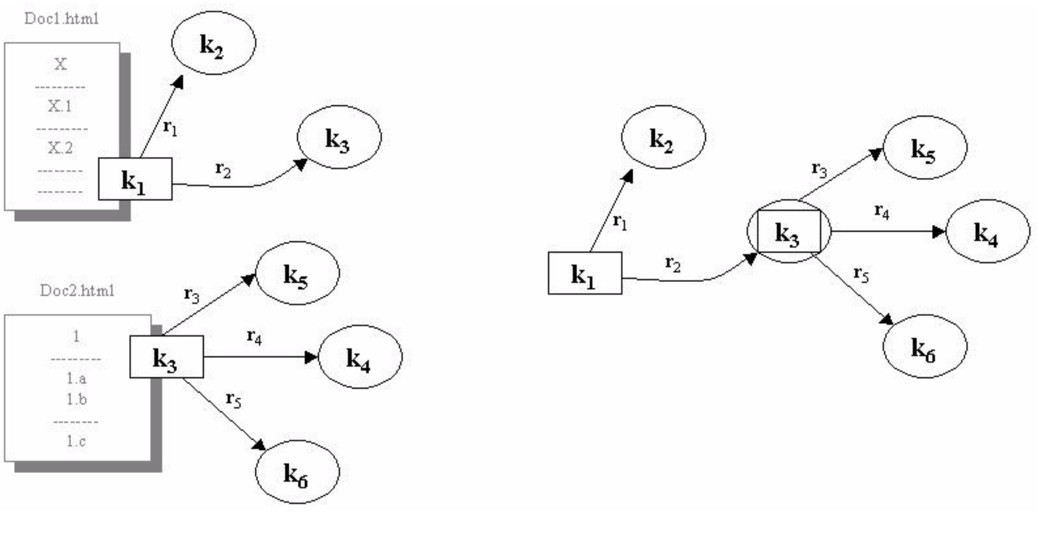


Figure 5–7: The relations are of type parent, correctly capturing the concept hierarchy

Now let's turn to the last two Heuristics from the set of hierarchy. As far as the loser candidate concepts concern, their lives should not be ended despite their loss in the fight for presidency in the selection process. Given new importance by Heuristic HR-4, they can finally rejoice and contribute in the process of identifying parent relations. If there, for two concepts $\mathbf{k}_1$ and $\mathbf{k}_2$, exists a candidate $\mathbf{c}'$ in the loser set $\mathbf{L}_{\mathbf{k}1}$ with the same conceptual description as $\mathbf{k}_2$, then the context of $\mathbf{k}_1$ is likely to be superior to that of $\mathbf{k}_2$. In other words, if a knowledge source has the concept `softwar` and that one of its loser candidates `user` is the concept of another knowledge source, then the `softwar` concept is parent to `user`. More formally, if $\exists\ \mathbf{c}' \in \mathbf{L}_{\mathbf{k}1}$, $\mathbf{c}' = \mathbf{k}_2$, then $\mathbf{r} = (\mathbf{k}_1, \mathbf{k}_2, \textit{parent})$. Additionally, if the reverse also hold at the same time (i.e. $\exists\ \mathbf{c}' \in \mathbf{L}_{\mathbf{k}2}$, $\mathbf{c}' = \mathbf{k}_1$) then the relationship is not of parenthood, but rather of a more synonymous nature. Heuristic HR-5 therefore claims that two concepts are synonymous if their **L** sets have some common denominators. More generally, note that whenever the **L** sets of two concepts share candidates so that $\mathbf{L}_{\mathbf{k}i} \cap \mathbf{L}_{\mathbf{k}j} \neq \{\varnothing\}$, the synonym relation is present between concepts $\mathbf{k}_i$ and $\mathbf{k}_j$. Finally, due to the values associated with each candidate, their joint sum provides a means to determine the strength of the relation. E.g, a threshold might be used so that relations weaker than the threshold can be turned down, hence controlling the size of the sets $\mathbf{R}_{\text{parent}}$ and $\mathbf{R}_{\text{synonym}}$. Figure 5–8 illustrates the role of the candidate sets.

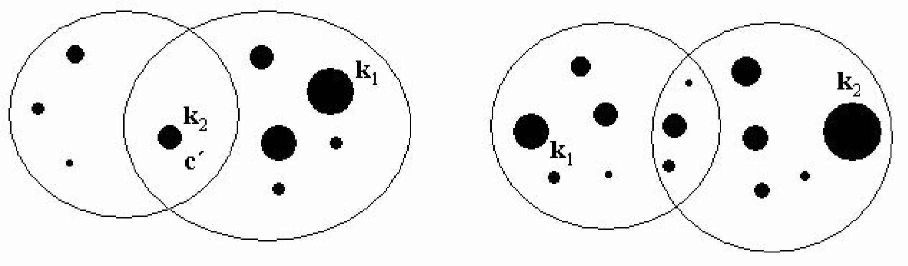


Figure 5–8: Heuristic HR-4 is illustrated to the left and HR-5 to the right. Different values of the candidate concepts are reflected by the various sizes of the black dots.

An interesting question is whether other relationship types can be found or should be embedded in the domain model. Obviously, some concepts are more difficult to understand than others. Does the presentational sequence matter during user interaction? Are some concepts redundant for some users, but highly necessary for others? How can we improve and facilitate intelligent reasoning in the adaptation process? The *prerequisite* relationship type adds knowledge to the conceptual hierarchy as it rely on a deeper understanding of the semantics of the concepts. If concept $\mathbf{k}_1$ is a prerequisite for concept $\mathbf{k}_2$ it means that $\mathbf{k}_1$ should be presented to the user before $\mathbf{k}_2$, whereas if $\mathbf{k}_1$ inhibits $\mathbf{k}_2$ the latter is no longer desirable in a presentation once the first is known. Note that it can be possible to infer the inhibitor relation from a sequence of prerequisites, though not necessarily: if $\mathbf{k}_1$ is a prerequisite for $\mathbf{k}_2$ which in turn is a prerequisite for $\mathbf{k}_3$, $\mathbf{k}_3$ most likely inhibits $\mathbf{k}_1$. The next set of Heuristics therefore focuses on ways to find prerequisites only.

Table 5–11: Heuristics for prerequisite relations

HR-6 A term in a knowledge source $\mathbf{KS}_1$ which is neither selected as concept nor in the upper list of candidates, yet a member of the DSL and chosen as concept for another knowledge source $\mathbf{KS}_2$, is prerequisite to the concept of the first knowledge source $\mathbf{KS}_1$.

HR-7 In the presence of a relation between two concepts, together with an unique, explicitly stated path of documents connecting the two corresponding knowledge sources, there is a set of prerequisite relations between the concepts of each document (but the first one), and the destination of the path.

As seen before, only the most valuable terms of a knowledge source were subject to abstraction. Remember that all candidates are terms but not all terms are candidates. There are two steps used by Heuristic HR-6 in order to infer prerequisites, visualised by Figure 5–9 and explained in the following.

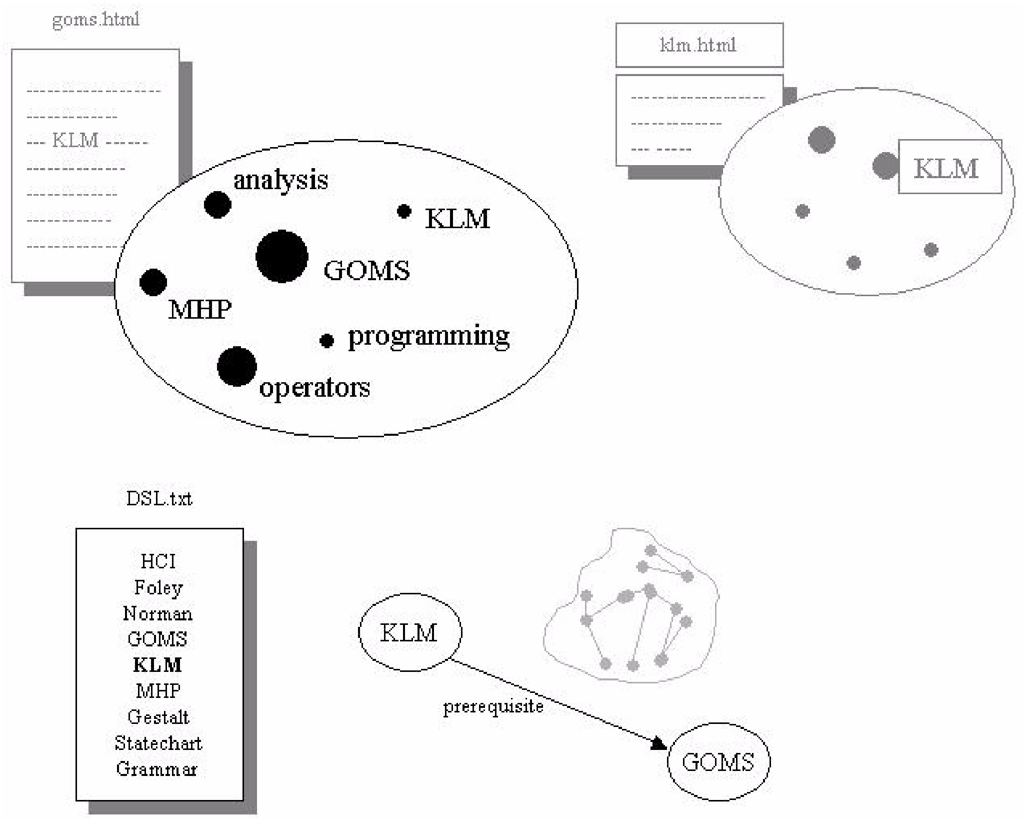


Figure 5–9: In the knowledge source “goms.html”, “GOMS” is selected as concept. One of the low-score terms “KLM” also occurs in the domain specific list. Since the “KLM” concept is already identified as a concept of another knowledge source (namely “klm.html”), the system concludes it to be important for the user’s understanding of the “GOMS” concept. This is indicated in the domain model through the prerequisite link.

For a randomly picked knowledge source **KS**, assume $\mathbf{k}_i$ was selected as the concept. The first step towards the prerequisites is to check if a term **t** from **KS** is assigned a low score during the conceptualization process. Then the system believes it to fall short as a key issue. Therefore, due to the Heuristics of the previous section (page 46 and on) it is unlikely that such a **t** is explained thoroughly in the text of **KS**. Second, if **t**, despite its low value, turns out to be a DSL member, it is for certain that it represents important knowledge of the domain as a whole. Since this knowledge is not debated in **KS** during user exploration, we conclude that if a concept **k´** from another knowledge source **KS´** that equals **t** already exists in the domain model, then it is a prerequisite to $\mathbf{k}_i$. In other words, the system can identify prerequisite relations **r** = (**k´**, $\mathbf{k}_i$, *prerequisite*) by comparing the DSL with the "less important" candidates from $\mathbf{Lk}_i$.

The last Heuristic depends on the results from the previous ones, as it makes use of one of the discovered implicit relations of any type between two concepts $\mathbf{k}_i$ and $\mathbf{k}_j$, i.e. **r** = ($\mathbf{k}_i$, $\mathbf{k}_j$, *relationship_type*). A sequence of linked knowledge sources (stated explicitly by hyper references) make up a path, and therefore the corresponding concepts constitute an explicitly stated path $\mathbf{P}_{\mathbf{k}i,\ \mathbf{k}j}$ = [$\mathbf{k}_i$, ..., $\mathbf{k}_j$], which is at least true at the document level. If both an **r** and some **P** exist for two concepts $\mathbf{k}_i$ and $\mathbf{k}_j$ as exemplified in Figure 5–10 below ($\mathbf{k}_1$ to $\mathbf{k}_9$), Heuristic HR-7 gives birth to another part of the set of prerequisites $\mathbf{R}_{\text{prerequisite}}$, namely $\sum_s \cup \mathbf{r}_s$ where $\mathbf{r}_s$ = ($\mathbf{k}_n$, $\mathbf{k}_j$, *prerequisite*) and i < n < j. The existing relation that led to the prerequisites, can be deleted since it duplicates the information.

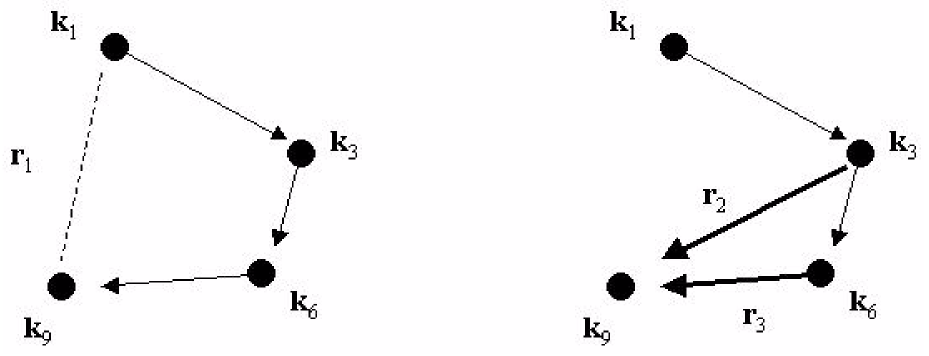


Figure 5–10: The explicitly stated path $\mathbf{P}_{\mathbf{k}1,\ \mathbf{k}9}$ = [$\mathbf{k}_1$, $\mathbf{k}_3$, $\mathbf{k}_6$, $\mathbf{k}_9$] together with the implicitly discovered relation $\mathbf{r}_1$ = ($\mathbf{k}_1$, $\mathbf{k}_9$, *relationship_type*) leads Heuristic HR-7 to find two prerequisite relations, namely $\mathbf{r}_2$ = ($\mathbf{k}_3$, $\mathbf{k}_9$, *prerequisite*) and $\mathbf{r}_3$ = ($\mathbf{k}_6$, $\mathbf{k}_9$, *prerequisite*).

In short, say a parent relation exists between the two concepts `debat` and `agent`, whose knowledge sources are "debating.html" and "agents.html" respectively. Moreover, when there also is a sequence of links from the two sources, e.g. from "debating.html" through "software.html" and "iui.html", to "agents.html", then the following prerequisites should result [1]:

[1] When the sources "software.html" and "iui.html" are conceptualised as *softwar* and *iui*, respectively.

- **r** = (**softwar**, **agent**, *prerequisite*)
- **r** = (**iui**, **agent**, *prerequisite*)

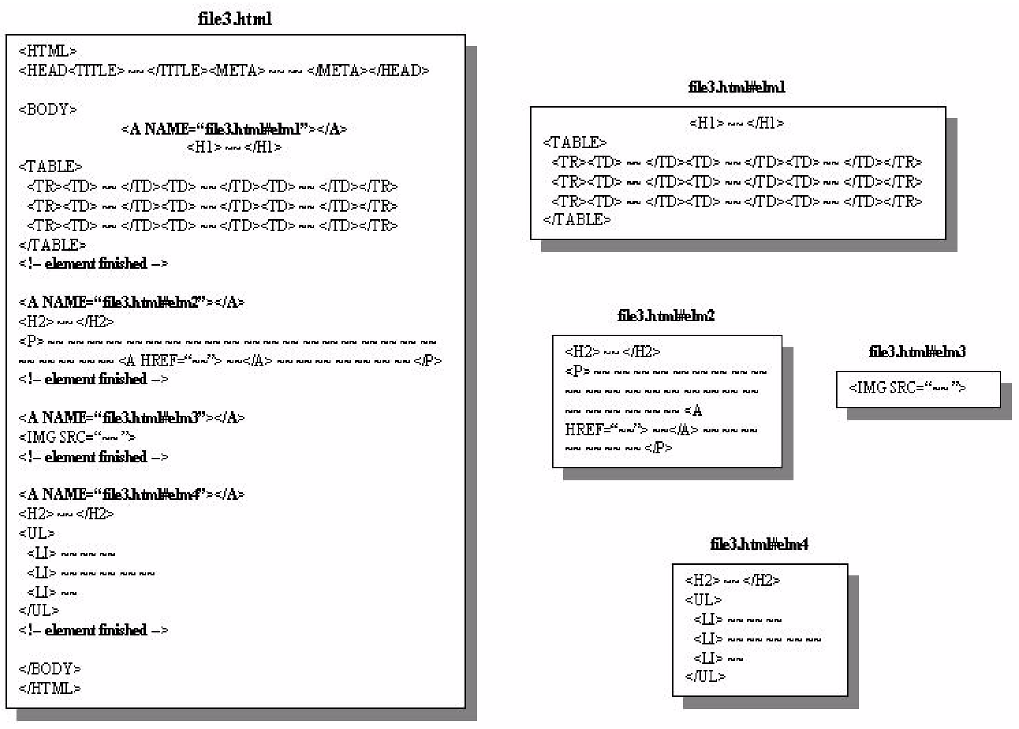
5.2.6 Completing the domain model

By now the system has identified a set of concepts $\mathbf{K}_{domain} = \Sigma_i \cup \mathbf{K}_{doc\ i}$ where and a set of relations $\mathbf{R}_{domain} = \{\mathbf{R}_{has\_deeper\_explanation}, \mathbf{R}_{external}, \mathbf{R}_{parent}, \mathbf{R}_{synonym}, \mathbf{R}_{prerequisite}\}$. Together, the sets constitute the building blocks of the domain model, i.e. representing the knowledge of the domain, which is an important basis for the adaptation engine to perform its tasks producing adaptive presentations for the user. As previsioned in section 4.5.1 "Building a domain model", page 30, the system can not be expected to construct a perfect domain model. Furthermore, as pointed out by Davis et. al, an imperfect model will lead to incorrect conclusions [Davis+93] which even more necessitates the need for revision and manual adjustment from a domain expert (i.e. the author).

When generating hypertextual presentations in the adaptive phase, the appropriate knowledge sources must be called forth. For efficiency reasons, the elements extracted from the documents should be stored in a database or in many small files (one file for each element). First, explicitly storing this information would ensure quicker customization of documents. Second, when in the adaptive phase the system must choose ingredients of the document to be generated. Since each knowledge source is formatted in HTML, scripting languages like `PHP`, `JSP` or `ASP` can be used to generate documents easily.

The notion of "document concepts" helped in the process of relating the different knowledge sources. Tempting as it may seem, merely storing the elements found in individual small files in the final knowledge base yields a potential pitfall. First, it would be difficult to reconstruct the original documents from the extracted, new knowledge sources, despite the parent relation, since only the elements classified as important were subject to conceptualization. In the worst case the original documents might be poorly tagged, hence leaving the system to omit a lot of important information in its domain model. Second, it would be difficult for the system to allow a user to switch between adaptive and original mode.

Retagging the source documents, i.e marking up each element with an unique ID using the <A NAME> tag, would elegantly solve these problems. Even better, retagged documents allow for other variants of adaptations including dynamic link generation to specific areas of interest, adding or removing sections of the original documents and so forth. Retagging the documents therefore yields an essential supplementary source of information for the system, second to the file/database representation. The storage issue is shown in Figure 5–11.



file3.html

```
<HTML>
<HEAD<TITLE> ~~ </TITLE><META> ~~ ~~ </META></HEAD>

<BODY>
                <A NAME="file3.html#elm1"></A>
                        <H1> ~~ </H1>
<TABLE>
  <TR><TD> ~~ </TD><TD> ~~ </TD><TD> ~~ </TD></TR>
  <TR><TD> ~~ </TD><TD> ~~ </TD><TD> ~~ </TD></TR>
  <TR><TD> ~~ </TD><TD> ~~ </TD><TD> ~~ </TD></TR>
</TABLE>
<!-- element finished -->

<A NAME="file3.html#elm2"></A>
<H2> ~~ </H2>
<P> ~~ ~~ ~~ ~~ ~~ ~~ ~~ ~~ ~~ ~~ ~~ ~~ ~~ ~~ ~~ ~~ ~~ ~~ ~~ ~~ ~~ ~~
~~ ~~ ~~ ~~ ~~ ~~ <A HREF="~~"> ~~</A> ~~ ~~ ~~ ~~ ~~ ~~ ~~ ~~ </P>
<!-- element finished -->

<A NAME="file3.html#elm3"></A>
<IMG SRC="~~ ">
<!-- element finished -->

<A NAME="file3.html#elm4"></A>
<H2> ~~ </H2>
<UL>
  <LI> ~~ ~~ ~~
  <LI> ~~ ~~ ~~ ~~ ~~
  <LI> ~~
</UL>
<!-- element finished -->

</BODY>
</HTML>
```

file3.html#elm1

```
                        <H1> ~~ </H1>
<TABLE>
  <TR><TD> ~~ </TD><TD> ~~ </TD><TD> ~~ </TD></TR>
  <TR><TD> ~~ </TD><TD> ~~ </TD><TD> ~~ </TD></TR>
  <TR><TD> ~~ </TD><TD> ~~ </TD><TD> ~~ </TD></TR>
</TABLE>
```

file3.html#elm2

```
<H2> ~~ </H2>
<P> ~~ ~~ ~~ ~~ ~~ ~~ ~~ ~~ ~~
~~ ~~ ~~ ~~ ~~ ~~ ~~ ~~ ~~ ~~
~~ ~~ ~~ ~~ ~~ ~~ ~~ <A
HREF="~~"> ~~</A> ~~ ~~ ~~
~~ ~~ ~~ ~~ </P>
```

file3.html#elm3

```
<IMG SRC="~~ ">
```

file3.html#elm4

```
<H2> ~~ </H2>
<UL>
  <LI> ~~ ~~ ~~
  <LI> ~~ ~~ ~~ ~~ ~~
  <LI> ~~
</UL>
```

Figure 5–11: To the left, an example of a retagged document holding information about the elements. For efficiency reasons, the elements are also stored in individual smaller files (to the right). The representation in total yields flexibility for the adaptation in many senses

5.3 Generating adaptive presentations

With knowledge of the domain model a step is taken towards tailoring documents to each user. The next concerns user modelling. This section *briefly* suggests some overall pragmatic issues in order to place the work from the previous sections in context of an AHS. In particular, we show how some of the previous made commitments ensure intelligence a the system's behaviour.

5.3.1 Structure of the user model

According to Heuristic HC-7, one concept represents many knowledge sources. This implies that a more complex representation of which concepts are actually learned should be embedded in the domain model. An appealing solution is to associate the knowledge state with each concept, so that the degree of user knowledge on a concept may vary. E.g. the user might have *no* knowledge at all, *incomplete, complete* or *deeper* knowledge about a concept.

A user model (UM) is a knowledge representation (KR) that must represent the knowledge state of each user. In general, a KR play five main roles [Davis+93], each leading to important properties. First, since the KR is a surrogate for some real entities, it is a source of error. Second, we need to make some decision on “what to see” in order to hide some of the potential complexity, and these ontological commitments constrain the view of the task at hand. Third, the KR is only part of intelligent reasoning. Different definitions of intelligence contribute both to the selection of what representation to use and to the form and content of the inferences that can be legally made. Finally, the last two roles deal with efficiency and expression issues, which is of less importance for the following discussion.

At first glance, it seems natural to copy the building blocks from the domain model (DM) into UM. However, several questions concerning the interaction are at hand. What should be presented to the user at the first time of interaction? How can adaptations be made based on empty user models? Do all users have equal preferences? When is the content learned? In order to answer these questions, we outline a structure for the user model, keeping the nature of a generic KR in mind.

The first two questions have straightforward solutions. First, the user could browse the original documents leaving for the system to record the concepts encountered (note that this is possible due to the retagging introduced in the previous section). After some initial interaction steps, UM would hold some few concepts (i.e. nodes), and adaptive behaviour can take place. The second option is for the system to present the information associated with a start-node, explicitly specified by the author, or randomly picked. Third, if the user has used the AHS earlier in other, related domains, as might well happen in educational contexts, there is a chance for similar concepts from the user model **UM´** of the already learned domain and the concepts to be learned from the present domain **DM**, thus allowing the system to initialise the present user model **UM**. Figure 5–12 illustrates a situation with a

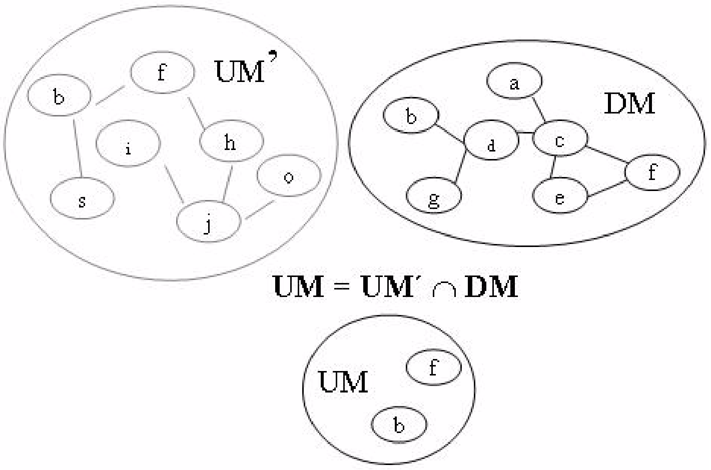


Figure 5–12: The user has already learned a domain UM’, which has some common concepts with the DM at hand. Note that since the concepts “b” and “f” are not directly related in DM, they remain so in the user model.

previously learned user model **UM´** = {b, f, h, i, j, o, s} , where the letters represent concepts, and the domain model to be learned is **DM** = {a, b, c, d, e, f, g}. The system can easily initialise a new user model **UM** through the operation **UM´** $\cap$ **DM,** i.e. **UM** = {b, f}.

The third question points out an additional benefit of embedding user models in adaptive systems. Not only the knowledge level will vary among users. Presentational and learning style preferences are also likely to differ, especially for hypertextual environments. Therefore, user models for an AHS should include preferred learning style or media types, background, skills of the user etc. Additionally, drawing assumptions on the level of the user skills, can be useful. More skilled users should be presented advanced features and immediately understand the underlying concepts, whereas novices would need a mix of many different presentations of the same concept. A stereotype approach as discussed in section 4.3 “Models add power to adaptive systems”, page 24, can be used for this part.

There are many factors to be considered when analysing user behaviour, e.g. how a node (concept) is accessed, which information is contained in the node and the time spent [Brusilovsky96]. Griffin suggests that a time interval is more appropriate, since people also can get distracted from their task [Griffin97].

To summarize, the user model should at least contain information on concepts, user characteristics (expert, intermediate, novice), user preferences (illustrations, text, summaries), state of the concepts (complete, ready-to-learn, incomplete) and the relations.

5.3.2 Adaptation Model

The system should tailor documents for the user in order to fill the gap of knowledge. In its simplest form the selection task constitutes the relative compliment $\mathbf{K}_{DM}$ - $\mathbf{K}_{UM}$, that is picking those concepts from the domain model that has not yet been offered to the user. However, such a solution is not sufficient for an intelligent behaviour. In which order should the concepts be presented? When is a concept actually learned? How to best bridge the gap of knowledge for users with varying skills? Obviously, the system lacks an important component.

According to Wu et. al, the adaptation model states how the system can perform its adaptation based on a set of adaptation rules. In order to separate the implementation dependent aspects from the explicit models AM, DM and UM, some sort of adaptation engine (AE) responsible for performing the adaptation [Wu+01] and updating the user model [Kules00] is necessary. For our AHS, an AE provides the composition of documents on the fly by selecting knowledge sources as pointed to from the concepts in DM, through reasoning on UM, based on rules from AM. Henceforth, the process of tailoring documents is referred to as how to select appropriate concepts.

Due to the presence of the conceptual states in UM and the relationship types in DM, the AE can add intelligence to the process of selecting concepts. Concepts in

UM have values indicating at which level they are understood. The relationship types of DM indicate how the concepts are related. Keep in mind that the goal of serving the user with information would be twofold: both to plan for a sequence of concepts and to complete their respective knowledge states. In the following both parts are intertwined, since the adaptive documents should result based on both considerations.

In order to fill in more knowledge about a concept and complete its state, corresponding sections not yet learned could be offered to the user. Following *has_deeper_explanation* and *parent* relations should extend the user's understanding of a concept, e.g. when presenting a concept that is a deeper explanation of another, the status of the latter should be updated from "incomplete" to "deeper". For concepts that don't have any attached deeper explanations, the deeper level can be obtained when all knowledge sources associated with the concept is learned. Furthermore, traversing nearby concepts, not only increases the respective conceptual states. In general, for the task of bridging the gap between concepts far apart in UM, relations from DM would usefully guide the AE to construct paths that in sum would network the gaps. The rules from AM would specify what to do when encountering different relationship types, hence contributing to the resulting curriculum. In particular, rules in the AE might use the prerequisite relations to verify whether the prerequisite concepts are satisfied every time a concept is asked for, and if not, simply present the prerequisites first. Finally, for expert users external relations would be useful as they most likely extend the context of the domain (c.f. HR-2, page 53). Likewise, external resources should not be presented to novices until all concepts are known.

5.3.3 Various forms of adaptations

Hitherto, the task of our AHS was to dynamically generate documents based on knowledge extracted from the domain. However, the structure of DM and the preservation of the original (retagged) documents presented in this thesis should provide for many sorts of adaptations. It is interesting to briefly explore some of the possible variants. We close this chapter by proposing three options acting as either stand alone solutions or extensions to our AHS.

One option that makes its actions quite transparent, is for the AHS to follow the predefined curriculum and content of the original documents, *greying out* undesired fragments, *insert* new sections, and dynamically adding constructed links (which is referred to as *link construction* in the literature). Greying out an undesired fragment would be based on whether the concept representing that section is learned or not. Sometimes, prerequisite relations would trigger the system to add more information on a page, e.g. based on incomplete knowledge of prerequisites[1]. Link generation can easily be accomplished due to the form of the marked-up sections from the retagging phase (c.f. Figure 5–11), allowing dynamic links to refer to specific areas of the original documents as well as the documents as a whole.

[1] The system might simply embed prerequisite concepts whenever their knowledge states are incomplete.

As a second option, based on the element types, we propose that short summaries can be customised for users asking for a synopsis of the domain or some subject. In particular, a *synopsis* for one concept can be made by generating lists out of knowledge sources that originally were e.g. heading-elements. Likewise, if a *resumé* of the domain is to be made, the system can use the same strategies for concepts it infers to be more important than others, that is for “key concepts”. As an example, an Heuristic which claims the number of relations to and from a concept indicate its importance, would be provide a simple way to identify key concepts. More sophisticated methods are obviously possible, like a more thorough analysis of DM, searching for the most centralised concepts. Remember from the discussion in the first section of this chapter where we debated what form the concepts should take on. Since we stuck to descriptive terms stolen directly from the knowledge sources instead of building a vector representation, *even shorter summaries* can also be provided by listing a range of concepts and present them as a keywords-list. This additional benefit would require no extra work but a few rules to select and display the most suitable concepts.

Lastly, if the user has insufficient or incorrect knowledge about a concept, e.g. revealed through predefined, provisional tests, the system should decrease its state in UM to the appropriate level, so that previously presented concepts can be redisplayed. This process possibly includes a reorganisation of the first curriculum or a more extensive use of illustrations. Furthermore, assume that the system can *visualise* DM, i.e. drawing the domain model network on the screen. When combined with the user model, the knowledge gap could be shown to the user, who would thus be able to place what was already learned into a larger context understanding what is around the next corner. Note that the ability for the user to puzzle the context of the part of DM which is not already walked-through, is again due to the conceptual descriptions.

6 IMPLEMENTATION AND EVALUATION

Up to the present, we have outlined the overall architecture and possibilities of an AHS, focusing on the development of Heuristics for adding quality to the process of building the domain model. In this chapter we first present an implementation of the strategy, demonstrating the resulting concepts and relations based on a small document collection. Next, we discuss and evaluate our work, concluding with some promising fields of future research.

6.1 Realising the domain model

Our strategy for conceptualization presumes that the domain documents are being parsed for important elements in order to build the domain model. Every term in these elements is a candidate concept. Guided by the Heuristics for concepts introduced in section 5.2.3, a conceptualizer module associates values with each candidate, leaving some with a higher score than others, and finally selects the most promising term as the concept. The same process apply at both the document and element level. The results from the parsing is passed on to another module for further analysis, which includes a merge of concepts and identification of relationship types.

Our HTML parser is written in `C` though any language would do for the task. The analysis is performed by `PROLOG`. Due to its properties as a language, `PROLOG` is very suitable for performing effective and powerful reasoning using only a few lines of programming code. The decision to use the combination of `C` and `PROLOG` is based on a wish for integrating the AHS routines into an existing agent based framework written in `C`. The agents at hand are `PROLOG` *machines* that can call `C` *routines* [*Thomassen99*]. Furthermore, `C` and `PHP` come well along, and suggestively, functions from `OpenGL` can easily be called from a `C` agent. All implementation specific details are attached in Appendix E - Implementation of conceptualizer.

6.1.1 A walk through the conceptualization processes

Before parsing, all hyper references in the domain are identified and stored in a file, and the values of the Heuristics for finding concepts are initialised. All the documents in the domain are listed in a file of document names, one name for each line. The process of analysing the domain is therefore controlled by walking through the file line by line while opening the next document and analysing it. Since stemming is used throughout the analysis, the DSL is also stemmed to be on the safe side. Thereafter, the first document is parsed followed by all its elements classified as important, then the second document and its important elements, and so forth. Unless otherwise stated, from now on we refer to both documents and important elements as *knowledge sources*.

According to one of the principles of HCI, it is much easier to agree or disagree with some proposal than coming up with something new from scratch. For instance, a DSL could be constructed semi-automatically by proposing the highest weighted terms from each document, then allowing the author to edit and correct the list. The construction should be rather simple since it can basically be implemented as a text file of words regarded as being important.

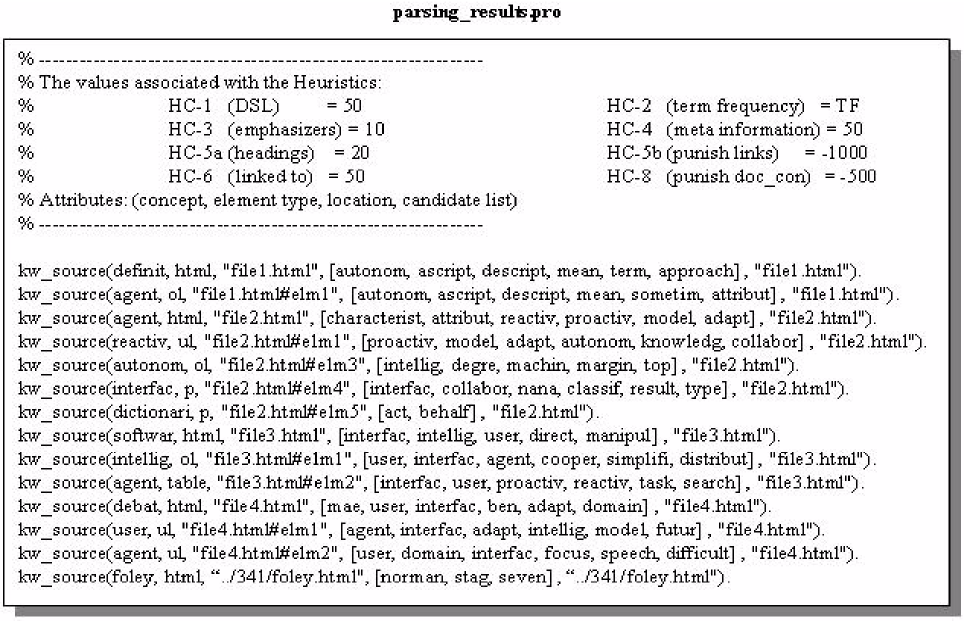
The Heuristics for finding candidates rely on several methods for operating on the knowledge sources in order to incrementally adjust the values of each candidate. All sets of temporal or longer term results are written to text files with one term for each line, allowing for easy manipulation of appropriate sets in the combination of an ADT (abstract data type) List. Regularly used functions include list operations for comparison, searching for specific terms, sorting, counting occurrences of the elements, reading a file to its internal list representation, and removing duplicate members. The step of lexical analysis and removing stopwords has an expensive cost and could account for as much as 50% of the computational expense of compilation [*Frakes+92*]. An extremely efficient implementation is to remove stopwords as part of the lexical analysis, that is, functions for lexical analysis and stopword removal are intertwined through the use of a DFA[1]. All terms surviving this process are given values by counting term frequencies, according to Heuristic HC-2, and membership in the DSL is checked for (HC-1), adding the associated value to members. Similarly, emphasizers in the knowledge sources are identified (HC-3), while meta information is of great importance for documents only (HC-4).

The links play a dual role. To fresh up from the discussion in the previous chapter, terms in hyper reference elements from somewhere in the domain pointing to the document being analysed, are likely to be among its strongest candidate concepts. Since such hyper reference terms are strong candidates for the corresponding destination documents, they are not suitable as candidates for the document hosting the link element. In concordance with Heuristics HC-5 and HC-6, the file of all identified links in the domain guides the punishment and reward of candidates. As mentioned earlier, documents are parsed before its elements, as required by Heuristics HC-8 and HC-11, and occurrences of the selected document concept in the elements of a document are punished by adding negative values.

As the parsing makes progress several things happen at different levels. Of importance, duplicate combinations of the tuple <tagName, conceptName> are not

[1] A Deterministic Finite Automaton (DFA) object represents words in a network of interconnected characters, so that a whole word is found by following paths of linked characters.

allowed for elements nor for documents, so the system rejects such proposals and tries to select among the consecutive candidate concepts until successful. The sections of the documents are being retagged and the knowledge sources identified are stored in individual, uniquely named files, all together constituting a new *knowledge base*. As far as reporting concerns, all candidate concepts are written to individual log files, one log file for each document. Additionally, the concepts selected are added to another file in a form understandable by PROLOG in order to facilitate further analysis. As illustrated in Figure 6–1: the file “parsing_results.pro” has information on each knowledge source, like conceptual representation, the element type[1], an ID for later retrieval from the already established knowledge base, and a *list* of candidates with scores close to that of the concept selected. The latter attribute (also referred to as “the upper list of candidates”) facilitates evaluation and adjustment of the proposed domain model as the system might suggest other candidates if the author is displeased with the proposed DM, and, as seen in the previous chapter, they will also play an important role when it comes to the identification of certain relationship types.



parsing_results.pro

```
% ---------------------------------------------------------------
% The values associated with the Heuristics:
%          HC-1  (DSL)        = 50            HC-2  (term frequency)  = TF
%          HC-3  (emphasizers) = 10           HC-4  (meta information) = 50
%          HC-5a (headings)   = 20            HC-5b (punish links)    = -1000
%          HC-6  (linked to)  = 50            HC-8  (punish doc_con)  = -500
% Attributes: (concept, element type, location, candidate list)
% ---------------------------------------------------------------

kw_source(definit, html, "file1.html", [autonom, ascript, descript, mean, term, approach], "file1.html").
kw_source(agent, ol, "file1.html#elm1", [autonom, ascript, descript, mean, sometim, attribut], "file1.html").
kw_source(agent, html, "file2.html", [characterist, attribut, reactiv, proactiv, model, adapt], "file2.html").
kw_source(reactiv, ul, "file2.html#elm1", [proactiv, model, adapt, autonom, knowledg, collabor], "file2.html").
kw_source(autonom, ol, "file2.html#elm3", [intellig, degre, machin, margin, top], "file2.html").
kw_source(interfac, p, "file2.html#elm4", [interfac, collabor, nana, classif, result, type], "file2.html").
kw_source(dictionari, p, "file2.html#elm5", [act, behalf], "file2.html").
kw_source(softwar, html, "file3.html", [interfac, intellig, user, direct, manipul], "file3.html").
kw_source(intellig, ol, "file3.html#elm1", [user, interfac, agent, cooper, simplifi, distribut], "file3.html").
kw_source(agent, table, "file3.html#elm2", [interfac, user, proactiv, reactiv, task, search], "file3.html").
kw_source(debat, html, "file4.html", [mae, user, interfac, ben, adapt, domain], "file4.html").
kw_source(user, ul, "file4.html#elm1", [agent, interfac, adapt, intellig, model, futur], "file4.html").
kw_source(agent, ul, "file4.html#elm2", [user, domain, interfac, focus, speech, difficult], "file4.html").
kw_source(foley, html, “../341/foley.html", [norman, stag, seven], “../341/foley.html").
```

Figure 6–1: Information about the knowledge sources are written to a file in a format understandable by PROLOG. For the document concepts, the ID’s appear as filenames only. For each element within a document, its ID is composed as follows: “filename” + “#elm” + “number”, where the number is increased as new elements are being analysed. Note that the concepts are stemmed. Also worth noticing is the uniqueness of the combination concept + tag, which agrees to the previous discussion.

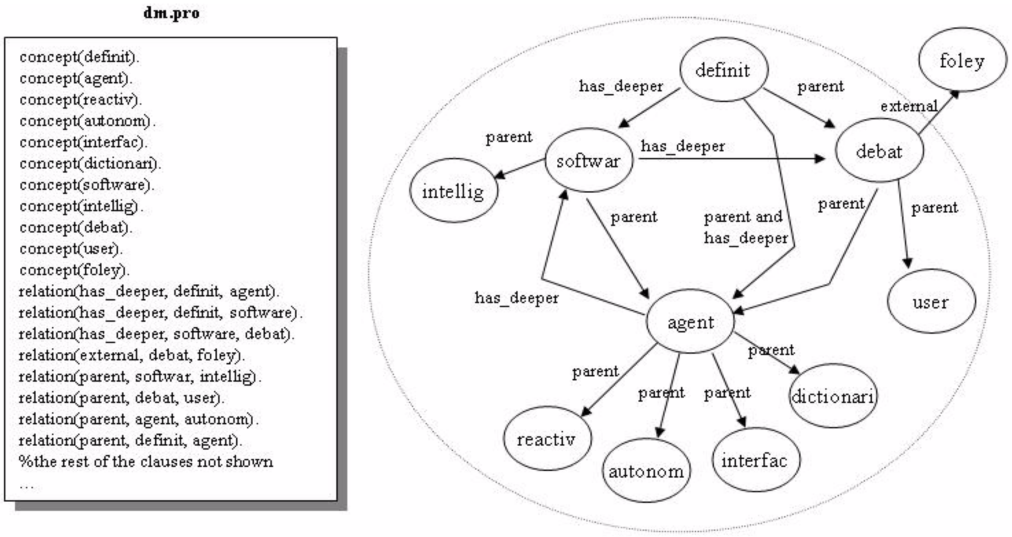
Each element originates from a document. The observant reader might wonder why document names are listed as well. Even though this information is held by the third attribute, or ID, it facilitates the implementation of one of the Heuristics.

[1] Just as element types are labeled with the corresponding tag names, documents are labeled as “html”.

The reason is that PROLOG is weak on string manipulation, even for a simple task like extracting the file name from the ID.

6.1.2 Seaming up the domain model

From the file in Figure 6–1 we see that some conceptual descriptions coincide. Note that the temporary division of document concepts and element concepts (c.f. section 5.1.4) were useful for splitting up the domain knowledge into appropriate knowledge sources, and for discussing how to capture the conceptual hierarchy. For the final DM, however, this division is superfluous since the conceptual hierarchy will be reflected through the relations. Furthermore, the file “parser_results.pro” must be kept intact for the future adaptation phase, since it holds information on which knowledge sources each concept represents. Therefore, listing all the conceptual descriptions as PROLOG facts in a new file while removing duplicates, elegantly solves the first part of completing the DM. We want a PROLOG representation of the domain knowledge in terms of abstracted concepts and how they are related. The left part of Figure 6–2 shows the domain model as such a file. To the right, the DM it is visualised as a conceptual structure.



dm.pro

```
concept(definit).
concept(agent).
concept(reactiv).
concept(autonom).
concept(interfac).
concept(dictionari).
concept(software).
concept(intellig).
concept(debat).
concept(user).
concept(foley).
relation(has_deeper, definit, agent).
relation(has_deeper, definit, software).
relation(has_deeper, software, debat).
relation(external, debat, foley).
relation(parent, softwar, intellig).
relation(parent, debat, user).
relation(parent, agent, autonom).
relation(parent, definit, agent).
%the rest of the clauses not shown
...
```

Figure 6–2: The domain model represented in a PROLOG file.

From the two illustrations shown by now, we see that one goal to be accomplished is to abstract the concepts only, from all the information held by the fact kw_source, where new facts should be written on the form concept(ConceptName). This task can be implemented straightforward as illustrated by the rather simple PROLOG code below:

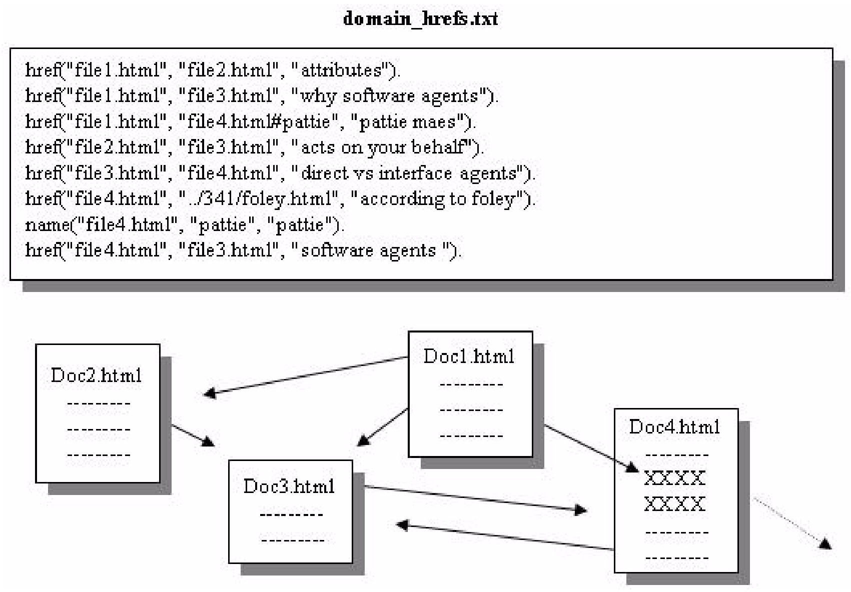
Table 6–1: PROLOG code for collecting all concepts and writing these as new facts.

```
%----------------------------------------------------------------------
% This code collects each concept from the kw_source fact,
% and writes the result as a new fact.

make_concept :-
     kw_source(ConceptName, _ , _ , _ , _ ) ,
     assert( concept(ConceptName) ) ,
     fail.
     % fail forces backtracking so that every combination is tried

make_concept.
     %when everything is tried and the above rule fails, this one succeeds
```

Heuristics HR-1 and HR-2 sought to find deeper explanation relationship types, doing so through the use of already identified hyper references. Remember that the initial step of conceptualization was to file all hyper references of the domain, which was then necessary to help the parser to point out concepts. Now this file becomes a useful source for the relational Heuristics HR-1 and HR-2. Figure 6–3 illustrates a small sample of links generated from the analysis of a domain.



domain_hrefs.txt

```
href("file1.html", "file2.html", "attributes").
href("file1.html", "file3.html", "why software agents").
href("file1.html", "file4.html#pattie", "pattie maes").
href("file2.html", "file3.html", "acts on your behalf").
href("file3.html", "file4.html", "direct vs interface agents").
href("file4.html", "../341/foley.html", "according to foley").
name("file4.html", "pattie", "pattie").
href("file4.html", "file3.html", "software agents ").
```

Figure 6–3: The file based on linkage between the original documents. Note that one link in Doc1.html points to a specific section of Doc4.html, which in turn has a link directed to somewhere outside the domain (namely to a document debating (“foley”). The destination in Doc4.html that is named “pattie” leads to the identification of a name-clause.

As mentioned, the domain was parsed based on a listing of all document filenames. The domain is useful when identifying relations, so this file is transformed to a PROLOG fact using a list. That is, for a domain with e.g. the four files “file1.html”, “file2.html”, “file3.html” and “file4.html”, the corresponding PROLOG fact takes on the form:

```
    domain([“file1.html”,“file2.html”,“file3.html” ,“file4.html”]).
```

a task which can be accomplished e.g. by the C-routines. Furthermore, by combining facts about hyper references with the facts on the knowledge sources, `PROLOG` routines can find out which sections are related and link the corresponding concepts. The built-in predicates `assert` and `retract` makes it possible to update the program with new clauses or delete clauses during program execution, another advantage with the `PROLOG` language. In other words, as the program precedes, it can modify itself. This convenience is used thoroughly in the implementation of the Heuristics, and was also used when making concepts out of the `kw_source` fact (c.f. Table 6–1). The code for the first two Heuristics are illustrated below.

Table 6–2: Heuristics HR-1 and HR-2 expressed in PROLOG.

```
%-----------------------------------------------------------------------
% HR-1: Hyper references between two knowledge sources somewhere within
% the domain are also relations of type has_deeper_explanation between the
% corresponding concepts.

hr1 :-
     href(From, To, _ ) ,
     kw_source(ConceptA, _ , From, _ , _ ) ,
     kw_source(ConceptB, _ , To, _ , _ ) ,
     assert( relation(has_deeper_explanation, ConceptA, ConceptB) ) ,
     fail.    % forces backtracking

hr1. % ensure success

%-----------------------------------------------------------------------
% member is recursively defined and finds out whether an X is in a list.

member(X, [X | Tail]).
member(X, [Head | Tail]) :-
     member(X, Tail).

%-----------------------------------------------------------------------
% The domain is constrained to a list of valid filenames, here exemplified
% with four files only. The rule outsideDomain checks for membership in
% the domain list.

domain( [“file1.html”, “file2.html”, “file3.html”, “file4.html”] ).

outsideDomain(F) :-
     domain(D) ,
     not member(F, D).

%-----------------------------------------------------------------------
% HR-2: External links with destinations outside of the domain
% are slightly different from internal ones, and the corresponding
% relations between such knowledge sources should be labelled as external.

hr2 :-
     href(From, To, _ ) ,
     outsideDomain(To) ,
     kw_source(ConceptA, _ , From, _ , _ ) ,
     kw_source(ConceptB, _ , To, _ , _ ) ,
     assert( relation(external, ConceptA, ConceptB) ),
     fail.    % forces backtracking

hr2. % ensure success
```

As an example, the task for HR-1 is to find concepts that are linked based on document linkage and we therefore start with establishing the source (the variable `From`) and the destination (the variable `To`) of each link by searching the `href` facts listed in Figure 6–3. Then we need the document concept from the source of the link and the document concept from the link destination, so we pick only these from the `kw_source` facts listed in Figure 6–1, elegantly ignoring the element types and the list of candidates using the `PROLOG` understandable underscore character as argument. Note that due to the form of the arguments of `href` and the form of the ID argument from `kw_source`, we don't have to account for element types, i.e. only document concepts will be selected. The set of Heuristic for the conceptual hierarchy can also be easily implemented in `PROLOG`. Before finishing the rule for HR-3, we note that when the filenames equal for two concepts, they denote the same document, and the element type "html" denotes the document concept.

Table 6–3: Implementing the Heuristics for finding parent and synonym relationship types.

```
%-----------------------------------------------------------------------
% HR-3: All element concepts found within a document are children
% of the document concept.

hr3 :-
     kw_source(DocumentConcept, html, _ , _ , File) ,
     kw_source(ElementConcept, _ , _ , _ , File) ,
     not DocumentConcept = ElementConcept ,
     assert( relation(parent, DocumentConcept, ElementConcept) ) ,
     fail.    % forces backtracking

hr3. % ensure success

%-----------------------------------------------------------------------
% HR-4: There is a parent relation if a member from a set of
% candidate concepts equals an already found concept.

hr4 :-
     kw_source(ConceptA, _ , _ , CandidateList, _ ) ,
     kw_source(ConceptB, _ , _ , _ , _ ) ,
     not ConceptA = ConceptB ,   %  the two concepts should not be equal
     member(ConceptB, CandidateList) ,
     not relation(parent, ConceptA, ConceptB) , %only consider once
     assert( relation(parent, ConceptA, ConceptB) ) ,
     fail.    % forces backtracking

hr4. % ensure success

%-----------------------------------------------------------------------
% HR-5: If two concepts have some joint members from their
% set of candidates, the concepts are synonymous.

commonMember(List1, List2) :-
     member(Term, List1) ,
     member(Term, List2).

hr5 :-
     kw_source(ConceptA, _ , _ , CandidateListA, _ ) ,
     kw_source(ConceptB, _ , _ , CandidateListB, _ ) ,
     commonMember(CandidateListA, CandidateListB) ,
     not ConceptA = ConceptB ,
     not relation(synonym, ConceptA, ConceptB) , %only consider once
     assert( relation(synonym, ConceptA, ConceptB) ) ,
     fail.    % forces backtracking

hr5. % ensure success
```

The last two Heuristics will aim at finding prerequisite relations. Similar to converting the file with the domain document listing, we also assume that the C-routines produces a `PROLOG` understandable version of the DSL. In order to make Heuristic HR-6 work, we need information on the terms that is not in the upper list of candidate concepts, in other words those terms that are not present in the last argument of clauses named `kw_source`. Remember from the previous discussion that such a list exists for each element, namely in the log file for each document. Assuming that the C-routines also convert this list to `PROLOG` facts on the form `lowerList(ID, ListOfLowerScoreTerms)`, we are set to implement this Heuristic.

Table 6–4: "Prologging" the first prerequisite relationship type.

```
%----------------------------------------------------------------------
% Lower score terms are listed as PROLOG facts, here illustrated with
% three examples. Note that the lists in lowerList and kw_source
% together constitute all the terms surviving lexical analysis.

lowerList("file4.html", [speech, futur, direct, manipu, design, system]).
lowerList("file4.html#elm1", [design, system, speech, memori, direct]).
lowerList("file4.html#elm2", [softwar, disagr, mainli, due, ben]).

inLowerList(Term, Location) :-
     lowerList(Location, Z) ,
     member(Term, Z).

%----------------------------------------------------------------------
% The domain specific list contains terms that are regarded important,
% here exemplified with seven terms only.

domainSpecificList( [hci, adapt, iui, user, model, domain, intellig] ).

inDsl(Term) :-
     domainSpecificList(DSL) ,
     member(Term, DSL).

%----------------------------------------------------------------------
% HR-6: A term in a knowledge source KS_1 which is neither selected as
% concept nor in the upper list of candidates, yet a member of the DSL
% and chosen as concept for another knowledge source KS_2,
% is prerequisite to the concept of the first knowledge source KS_1.

hr6 :-
     inLowerList(Term, Location) ,
     inDsl(Term) ,
     kw_source(Concept , _ , Location, _ , _ ) ,
     concept(Term) ,
         %does the concept of the source equals the term?
     assert( relation(prerequisite, Term, Concept) ) ,
         %if so, then make a relation between the two.
     fail.    % forces backtracking

hr6. % ensure success
```

The `PROLOG` code of Heuristic HR-6 seems more complex than those of the previous rules. Will it generate the prerequisite relation correctly? Refresh on the situation of Figure 5–9 (which illustrated Heuristic HR-6) where a regular term *klm*, which also occurred in the DSL, was only briefly mentioned in a document. Since *klm* already had been identified as a concept somewhere else, and *goms* was voted as concept of the document in which *klm* lived as a regular term only, we claimed *klm* to be prerequisite to *goms*. In order to explain the implementation, we illustrate by including only the facts necessary. Note that the content of Table 6–5 corresponds to the *klm/goms* situation, where the two clauses `concept(goms).` and `concept(klm).` were obtained by calling the previously stated rule `make_concept.`

Table 6–5: A small portion of the knowledge on two documents as seen from PROLOG.

```
kw_source(goms, ul, "file5.html", [operator, mhp, analysis], "file5.html").
kw_source(klm, p, "file7.html", [keystroke, button, time], "file7.html").

lowerList("file5.html", [mouse, klm, movement]).

domainSpecificList( [hci, foley, norman, goms, klm, mhp, gestalt] ).

concept(goms).
concept(klm).
```

To start, we ask `hr6.` in order to identify the prerequisite relations which conditions to the Heuristic. The first two goals tries to satisfy membership in the `lowerList` and the `domainSpecificList`, using the rules from Table 6–4. For successful bindings to the variable `Term` (in our case the second member of the lower list, that is *klm*), the `Location` variable is further used in order to identify the concept of the knowledge source. Next, we must ask whether the term found is also a concept, simply by posing the question `concept(Term).` Since there is such a clause for the present term, namely `concept(klm).` the question succeeds, and a prerequisite relation can be inserted as part of the knowledge database. Finally, in order to validate, we ask `PROLOG` the question `relation(prerequisite, A, B).` to check whether any relations were found. The result is that the variables `A` and `B` are bound to *klm* and *goms* respectively, in other words `PROLOG` found the newly inserted fact `relation(prerequisite, klm, goms).`

The last Heuristic HR-7 assumes there is a path between two related concepts. Thus, its solution involves the traversal of several paths. We therefore start by defining the code for how to find a path, and doing so requires the use of *edges*, (a path is made of edges between the nodes). Note that we already have the edges established from the `href` facts. If a path exists along with an already discovered relation, the procedure `addToSet` starts to extend the database with new prerequisite relations. Finally, the original relation is removed. The code for all this is summarised below and will not be further explained. Again, notice the compact form due to the properties of the PROLOG language.

Table 6–6: Paths are the basis for finding more prerequisites.

```
%----------------------------------------------------------------------
% There is a path between two nodes if there is an edge from the first
% node to an intermediate node, which again has a path to the destination
% node. This code also adds the nodes to the head of a list.

path(From, To, [To] ) :-
     href(From, To, _ ).

path(From, To, [Intermediate|Tail] ) :-
     href(From, Intermediate, _ ), %adds to head of list
     path(Intermediate, To, Tail).

%----------------------------------------------------------------------
% This rule asserts prerequisite relations from the intermediate
% members of a path to the last node

addToSet([Head|Tail], ToConcept) :-
     kw_source(FromConcept, _ , Head, _ , _ ) ,
     not FromConcept = ToConcept ,  %write relation if the concepts differ
     assert( relation(prerequisite, FromConcept, ToConcept) ) ,
     addToSet(Tail, ToConcept) ,  %move down the path

addToSet([], _). %trivial case

%----------------------------------------------------------------------
% HR-7: In the presence of a relation between two concepts, together with
% an unique, explicitly stated path of documents connecting the two
% corresponding knowledge sources, there is a set of prerequisite
% relations between the concepts of each document (but the first one),
% and the destination of the path.
% Note that the procedure removes the original relationship type if
% some prerequisites are found

hr7 :-
     relation(Type, ConA, ConB) , %if any relation exists
     kw_source(ConA , _ , From, _ , _ ) , %we go get the
     kw_source(ConB , _ , To, _ ,  _ ) ,   %destinations
     path(From, To, Visited) , %is there an explicitly stated path?
     addToSet(Visited, ConB) , %add all relations along this path
     retract( relation(Type, ConA, ConB) ) , %remove original relation
     fail.    % forces backtracking

hr7. % ensure success
```

By now, we have seen that all concepts and relations can be found using PROLOG. Together, they provide the basis for completing the domain model. Once this information is merged into one file (e.g. “dm.pro” used in Figure 6–2), this file essentially constitute the domain model, that is, one single file can represent the domain knowledge. The Heuristics work independently of each other, and hence there is also a chance that some concepts are linked with more than one relation. In particular, some Heuristics produce bi-directional relations. For instance, Heuristic HR-1 produces the relations

```
relation(has_deeper_explanation, software, debat).
relation(has_deeper_explanation, debat, software).
```

In order to fix this problem, one of the superfluous clauses can be retracted from the PROLOG database. The rule removeBidirectional in Table 6–7 shows how.

Table 6–7: Some clauses to clean up bi-directional relations

```
removeBidirectional :-
     relation(Type, ConceptA, ConceptB) ,
     relation(Type, ConceptB, ConceptA) ,
     retract( relation(Type, ConceptB, ConceptA) ) ,
          %removes the latter bidirectional relation
     fail.    % forces backtracking

removeBidirectional. % ensure success
```

A well known principle of design is that human beings find it easier to overrule errors from a proposal than coming up with one themselves from scratch. As noted, someone skilled in the domain should revise the DM in order to secure that only appropriate concepts and relations remain. We suggest a graphical visualization in the form of a conceptual network, though other presentational forms are possible. Note that the number of concepts might get out of hand. Therefore the system should provide means for “switching off” some concepts and hence leaving only part of DM visible. Such a demand can be fulfilled simply through hiding the subordinate concepts of the parent relations, that is zooming in or out until the desired level of detail is achieved (when using a graphical interface). In a similar fashion, if the author prefers a pruned network, the system should offer to permanently delete all concepts and relations not visible. Moreover, for the human reviser, the file “parsing_results.pro” proves to be useful in many ways. As an example, its list attribute, whose members are ranked by value, will facilitate tasks like renaming since new concepts can be proposed as substitutes for those incorrectly suggested (by the system) in the first place.

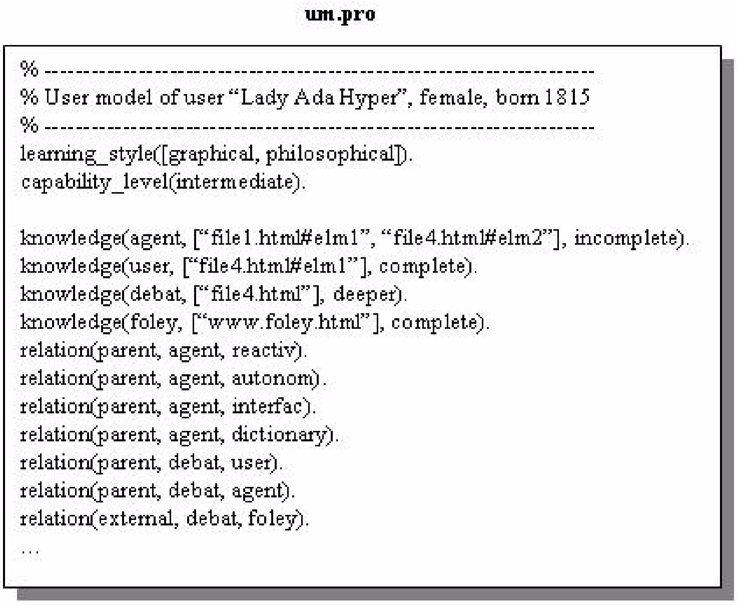
6.2 Possibilities for the AHS

In the following we briefly sketch an example of how a simple user model and adaptation model might look, thus substantiating the choice of a PROLOG based implementation. We focus on the representation of the conceptual knowledge and generic user information and preferences.

6.2.1 Picturing a simple user model

As outlined in the previous chapter, UM must hold information on the user’s level of conceptual knowledge. A simple scheme is to save information on which concepts the user already has knowledge and at what level. The user model should be incrementally built so that it at any time can provide the adaptive hypertext system with information on what the user knows. Since one concept can represent many different knowledge sources, it is important that the user model also records which knowledge sources have been presented for each concept. Continuing with

PROLOG notation, we use a fact knowledge which has attributes on the *concept*, along with a *list* of knowledge sources that have been shown for each concept, and finally a *state* indicating which knowledge level the user has on each concept. For instance, the state of the concept “debat” is set to deeper since the user has accessed the external resource conceptualised as “foley” (c.f Figure 6–2). Since we regard UM as an overlay model of DM, it seems natural to also include the corresponding relations as new concepts are learned, that is, relations to neighbour concepts. Why embed relations in the user model? There are at least one reason. When a concept is learned and its neighbours are not, the knowledge gap should be bridged, and if the relations are kept in the domain model only, the implementation of the adaptation rules becomes cumbersome. Figure 6–4 shows how the file



um.pro

```
% -----------------------------------------------------------------
% User model of user “Lady Ada Hyper”, female, born 1815
% -----------------------------------------------------------------
learning_style([graphical, philosophical]).
capability_level(intermediate).

knowledge(agent, [“file1.html#elm1”, “file4.html#elm2”], incomplete).
knowledge(user, [“file4.html#elm1”], complete).
knowledge(debat, [“file4.html”], deeper).
knowledge(foley, [“www.foley.html”], complete).
relation(parent, agent, reactiv).
relation(parent, agent, autonom).
relation(parent, agent, interfac).
relation(parent, agent, dictionary).
relation(parent, debat, user).
relation(parent, debat, agent).
relation(external, debat, foley).
...
```

Figure 6–4: A simple user model representing user knowledge at various levels. Note that only relations to neighbour concepts of the concepts already learned, are included in UM.

“um.pro” represents a simple user model[1]. Note that due to the uniqueness of the tuple <tagName, conceptName>, the information in the tag list can be either in terms of the ID or as the tag name (we prefer using the ID, in case the author, despite warnings from the system, overrules the recommendations and assigns the same concept for two equal elements).

A user model should present information according to the needs and preferences of each user with regard to presentational form and learning abilities. Thus, by including some generic information on the learning styles and skills of each user in UM, the system is not constrained to a selection among the concepts not yet learned. Along with unknown concepts, also the preferences of each user can influence the adaptive commitments, so that presentational form and degree of difficulty can be further varied.

[1] Possible extensions include e.g. the time spent on each node and interaction history.

From the clauses

```
learning_style( [graphical, philosophical] ).
capability_level(intermediate).
```

we see that the user “Lady Ada Hyper” prefers the system to generate visualizations if possible, a task that can easily be accomplished simply by asking for images and tables of the concept being debated. Remember that this information can be found from the attribute *element-type* in the `kw_source` facts. Likewise, the system has somehow inferred the capability of the user to be `intermediate`. It would therefore e.g. present easier concepts before the more difficult ones, but still challenge the user on more difficult subjects, thus aiming at improving the user knowledge without exaggerating.

6.2.2 Adaptation rules

We believe that the implementation of DM and UM in terms of facts has made a powerful foundation for the adaptive phase, and opened for various and flexible adaptive presentations, which can be fairly quickly generated by means of `PROLOG` rules. According to the introductory part on adaptive systems in section 4.5 “Planning an adaptive hypertext system”, page 29, the actions or tasks of an adaptive hypertext system are controlled by the adaptive engine operating on an adaptation model. In accordance with our work so far, a task would lead the adaptive engine to execute a query, and the rules in the adaptation model would ensure that the correct action can be executed. Three independent tasks to illustrate example rules are proposed in Table 6–8. The tasks will be explained in the following. Note that the clauses of the last rule are explained in detail after discussing the first two rules.

Table 6–8: Examples of how easily powerful adaptation rules can be written in PROLOG.

Description of task	Example of AE call	Rule that fires in AM

`1. Provide graphical information about a concept to the user.`	`?- show(agent,img).`	`show(Concept, ElmType) :-`
		`  kw_source(Concept, ElmType,`
		`            FileID, _ , _ ) ,`
		`  display(FileID).`

`2. Based on a concept in DM, visualise a sub-network of relations and neighbour concepts not yet visited.`	`?- visualise(user).`	`visualise(Concept) :-`
		`  concept(Concept) ,`
		`  relation(Type, Concept, Neighbour) ,`
		`  draw(Concept, Neighbour, Type) ,`
		`  fail. %get all neighbours`
		`visualise(Concept). %ensure success`

`3. Propose new concepts to be presented to the user based on the gap close to a given concept in the user model`	`?- next(softwar).`	`next(Concept) :-`
		`  getNeighboursDM(Concept, DMList) ,`
		`  userKnows(Concept, UMList) ,`
		`  findGap(DMList, UMList, GapList).`

The first task listed concerns to embed graphical information on a concept in the adaptive document being presented to the user. It assumes the parser found exactly that concept for an image element during parsing the documents. If so, the image to be presented is contained as HTML code within the knowledge base in a small file whose reference is kept in the third attribute of `kw_source`, so the solution is simple: Query the Adaptive Engine to display the correct `fileID` if the associated concept and the correct type proves to be true for a knowledge source. Note that we assume that the rule `display(FileID)` exists without focusing on its implementation. Somehow, it should return the `fileID` to the system for generation of the adaptive document to be presented. One way to do this is for display to write include-sentences to a file "includes.php", which e.g. a `PHP` script traverses and uses further for generation of the final documents. Note this is very simple for `PHP` since it can tailor many different documents by simply including their filenames.

The second task provides a visualization of the nearby network of a concept upon request. Note that whereas the information is provided by simple `PROLOG` clauses, the actual routines for drawing might be more complex and typically performed by a routine written in another programming language, typically `OpenGL`. As noted, one reason for choosing `PROLOG` and `C` is that the two can communicate within an already existing agent based framework, and `OpenGL` routines can be called from `C`.

Before considering the third task, we recall that the goal of adaptation is to satisfy the user with respect to knowledge, and in order to do so, a plan for which concepts to present next and how to present them, is needed. There are various strategies for filling up the user model, each leading to different adaptation rules. One strategy is to search the domain model in a breadth first manner and introduce all concepts briefly before focusing on increasing the user understanding of each concept. Cycling through the domain model in order to cover up incomplete user knowledge would gradually perfect the user model. More likely than benefiting from many walk-throughs, the user could experience all the jumping and revisiting to be annoyingly inefficient and loose track of the knowledge to be learned. The opposite strategy follows a depth first like search through the domain, aiming at fully describing each concept in one operation. A problem with finishing off each concept before regarding new ones is that the user could miss sight of the greater picture.

Another strategy for guiding the selection process is based on the relations identified so far. Using relations when searching for new pieces of knowledge includes several subordinate tasks to be accomplished. Remember that the gap of knowledge in the user model consists of *unrelated* areas in the user model with corresponding *related* areas in the domain model. The task then comes to bridge the gap of knowledge using the relations of DM. Note from the below illustration that the concept `softwar` in DM is related to the concepts `intellig`, `agent`, `definit` and `debat`, whereas the user has learned the concepts `software` and `agent` only, as indicated with empty circles. In other words, there is a gap close to the concept `software` which should be bridged, and the concepts `intellig` and `definit` constitute the building blocks to the piece of engineering work at hand.

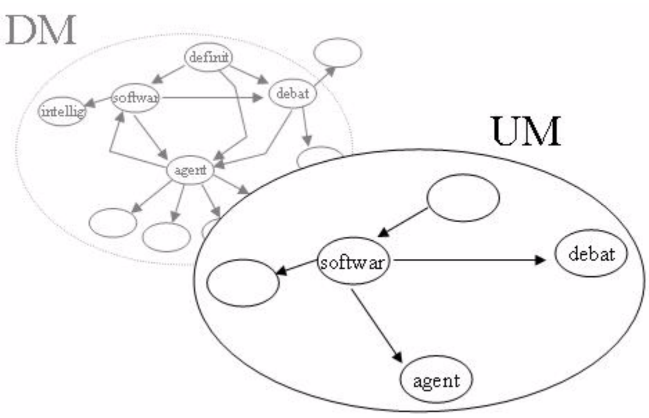


Figure 6–5: The gap in the user model.

The strategy is to let lists represent both UM and DM, and show all members from UM not also in DM. The first step is to identify a list of neighbours from DM close to a concept, doing so by calling the rule `getNeighbours(Concept, DMList).` and now `DMList` holds all nodes close to the concept. Similarly, a list of user model neighbours to the concept must be found. The next step of the strategy is to use the rule `findGap(DMList, UMList, GapList).` to subtract the concepts of the user model from the members of the domain model. This operation is basically a process of deleting every member of `UMList` from `DMList`, resulting in another list `GapList` holding the concepts to be displayed to the user. Using our example, `DMList` initially looks like this:

```
[agent, intellig, definit, debat]
```

and the `UMList` contains

```
[agent, debat]
```

so `GapList` would after the `findgap` rule is run, yield

```
[intellig, definit]
```

Note that the system should also validate whether other concepts are required to be known before displaying the members of `GapList`, placing possible prerequisites along with the gap concepts found in a final list, which should then be traversed and displayed to the user. Table 6–9 sums up the rules needed for bridging gaps, but does not embed the rules needed to validate prerequisites.

Table 6–9: Examples of adaptation rules written in PROLOG.

```
%------------------------------------------------------------------------
% The rule next(Concept) finds a gap and builds a bridge, that is a list
% of concepts that can be visited next. Note that this routine does not
% validate the prerequisites of the concepts in the gap.

deletepossible(Item, _ , [] ).
deletepossible(Item, [Item|Tail] , Tail).
deletepossible(Item, [Y|Tail] , [Y|Tail2] ) :-
     deletepossible(Item, Tail, Tail2).

insert(X, List, BiggerList) :-    %insert is the inverse of delete
     delete(X, BiggerList, List).

findGap( [] , R, R).  %trivial case
findGap( DMList, [Head|Tail], GapList):-
     deletepossible(Head, DMList, IntermediateList) ,
             %deletes common members
     findGap(Tail, IntermediateList, GapList).

all(Concept, ListIn, ListOut) :-   %go get all neighbours
     relation( _ , Concept, Next) ,
     not member(Concept, ListIn) ,
     all(Concept, [Next|ListIn], ListOut).
all(C, List, List). %trivial case

getNeighbour(Concept, DMList) :-  %get neigbours to a concept in DM
     all(Concept, [], DMList).

userKnows(Concept, UMList) :-    %get concepts known to the user.
     all2(Concept, [], UMList).  %almost equal to all, except that
                                 %relation in all is the DM relation,
                                 %and relation i all2 is UM relation.

next(Concept) :-
     getNeighboursDM(Concept, DMList) ,    %list all neighbours from DM
     userKnows(Concept, UMList) ,  % build list of neighb. known to user
     findGap(DMList, UMList, GapList) ,  %subtract UM from DM
```

6.3 Evaluation

The goal of the conceptualizer module was to abstract the contents of documents and the sections as good as possible. Rules were then used to complete DM with relations between the appropriate concepts. Fine, but are the results reliable? How can we justify the actions taken? What if the structure of a collection of documents deviate from the assumptions of some Heuristics? This section discusses the influence of the commitments made.

6.3.1 Optimizing and justifying the performance

In section 5.2.4 we suggested that it is possible to experiment with the total outcome of the conceptualization process by varying the values associated with each Heuristic. The candidate concepts are terms, where the one with the highest value after all Heuristics are performed, is selected as concept. For every term $\mathbf{t}_i$ an Heuristic can either apply or not, indicated in Table 6–10 with "Yes" and "No", respectively. The columns indicates which of the Heuristics do fire for each term. Based on some few possible combinations, this section tries to draw some conclusions and apply suitable values for an optimal combination of the values donated from each Heuristic. Note that HC-5 is implemented in two portions in order to account for its dual nature while Heuristics HC-7, HC-9, HC-10 and HC-11 are left out since they mainly deal with issues like the level of abstraction, sequence of analysis etc.

Table 6–10: The outcome of a term is determined by the total score, which is due to the effect from each Heuristic. Some of the possible combinations are listed in the columns.

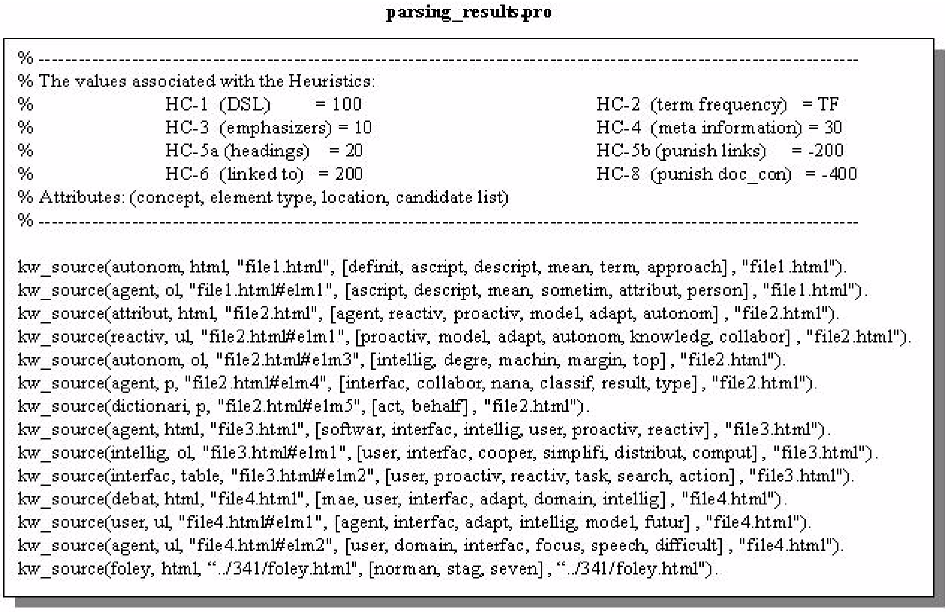
Heuristic	$\mathbf{t}_1$	$\mathbf{t}_2$	$\mathbf{t}_3$	$\mathbf{t}_4$
HC-1 (Domain specific list)	No	Yes	No	Yes
HC-2 (Term frequency)	Yes	Yes	Yes	Yes
HC-3 (Emphasizers)	No	No	No	Yes
HC-4 (Meta and title)	No	No	No	No
HC-5a (Occurrences in different elements)	No	No	No	No
HC-5b (Punish outgoing links)	No	Yes	Yes	No
HC-6 (Incoming links pointing here?)	No	No	Yes	Yes
HC-8 (Punish document concept)	No	No	No	Yes

For instance, the term $\mathbf{t}_1$ does not appear in the DSL, nor in any important element, so the only Heuristic that strikes, is Heuristic HC-2, as shown in its column. We make several notes from the above combinations:

1. All candidates have a term frequency (HC-2), but the number will vary from one and up. In the worst case, no Heuristics but the second one proves correct, as for $\mathbf{t}_1$.
2. A term might well occur both in the DSL and in outgoing links, as illustrated with $\mathbf{t}_2$.
3. There are terms like $\mathbf{t}_3$ that occur both in outgoing and incoming links (HC-5b and HC-6) at the same time. With incoming links, we think of links from elsewhere pointing to a specific section hosting the term.
4. For an element, all Heuristics but the last one suggests assigning positive values to the candidate $\mathbf{t}_4$.

Instead of testing empirically which combination of values yield the better results, we try to analyse the points observed above. From the first we conclude that the values must agree with the term frequencies so that they can affect the total outcome. The second point reveals a problem. Outgoing links are subject for punishment (negative value) whereas DSL members obviously should yield a high, positive value. Which Heuristic should veto? Since the DSL apply for all documents, we infer that HC-5b should overrule HC-1. The third case concerns instances where a candidate is regarded likely to be the concept of another document and the present one at the same time, due to membership in hyper references. In order to resolve this conflict, we note that since links are only assumed, not for certain, to be descriptive of their targets, the sum of the two should neutralize each other. Finally, even though the first Heuristics seem to assign plenty of positive values for an element, the same candidate is already chosen as the document concept. It should therefore not be selected again as concept for any element in the document. It is necessary for HC-8 to have a negative value able to beat the sum of the positively minded Heuristics HC-1, HC-3 and HC-6, also accounting for relatively high term frequencies.

Almost like solving a mathematical equation, appropriate values can be set. Note a little trial and error will do in order to find values that satisfy all of the above demands. As one solution, Figure 6–6 reveals how the results (the `kw_source` facts) of an optimal set of Heuristics (the header section) differ from the setup used for the other examples of this chapter, thus outlining the importance of committing to appropriate values. Note that this is favourable rather than a limitation for the system since it allows for a variation of proposed models made by the system. Additionally the author can decide which Heuristics to apply or omit, the latter being fruitful in case a collection of documents uses some tags for other purposes than outlined in this thesis.



parsing_results.pro

```
% -------------------------------------------------------------------------------------------------------------------
% The values associated with the Heuristics:
%                     HC-1  (DSL)        = 100                        HC-2  (term frequency)   = TF
%                     HC-3  (emphasizers) = 10                        HC-4  (meta information) = 30
%                     HC-5a (headings)    = 20                        HC-5b (punish links)     = -200
%                     HC-6  (linked to)   = 200                       HC-8  (punish doc_con)   = -400
% Attributes: (concept, element type, location, candidate list)
% -------------------------------------------------------------------------------------------------------------------

kw_source(autonom, html, "file1.html", [definit, ascript, descript, mean, term, approach], "file1.html").
kw_source(agent, ol, "file1.html#elm1", [ascript, descript, mean, sometim, attribut, person], "file1.html").
kw_source(attribut, html, "file2.html", [agent, reactiv, proactiv, model, adapt, autonom], "file2.html").
kw_source(reactiv, ul, "file2.html#elm1", [proactiv, model, adapt, autonom, knowledg, collabor], "file2.html").
kw_source(autonom, ol, "file2.html#elm3", [intellig, degre, machin, margin, top], "file2.html").
kw_source(agent, p, "file2.html#elm4", [interfac, collabor, nana, classif, result, type], "file2.html").
kw_source(dictionari, p, "file2.html#elm5", [act, behalf], "file2.html").
kw_source(agent, html, "file3.html", [softwar, interfac, intellig, user, proactiv, reactiv], "file3.html").
kw_source(intellig, ol, "file3.html#elm1", [user, interfac, cooper, simplifi, distribut, comput], "file3.html").
kw_source(interfac, table, "file3.html#elm2", [user, proactiv, reactiv, task, search, action], "file3.html").
kw_source(debat, html, "file4.html", [mae, user, interfac, adapt, domain, intellig], "file4.html").
kw_source(user, ul, "file4.html#elm1", [agent, interfac, adapt, intellig, model, futur], "file4.html").
kw_source(agent, ul, "file4.html#elm2", [user, domain, interfac, focus, speech, difficult], "file4.html").
kw_source(foley, html, "../341/foley.html", [norman, stag, seven], "../341/foley.html").
```

Figure 6–6: The values should be optimal in accordance with the observations listed in Table 6–10. Note that these values result in different knowledge sources than the values used in Figure 6–1. Hence, other concepts and relations are produced for the final domain model.

By leaving Heuristic HC-1 with a value of zero, i.e. neglecting the DSL, the final DM would again turn out different. Note, however, that in the absence of a DSL, both the conceptualization and the identification of the prerequisite relationship type would suffer. In a similar fashion, we may expect the granularity of the DSL regarding content to affect the grand total. Therefore both the DSL and the values of the Heuristics can be regarded as tools for which we can experiment on the final outcome.

6.3.2 An ocean of relations

The code for the relations can very easily be tested using any `PROLOG` environment. For the four sample documents listed in appendix C (which is also the basis for most of the screen-shots from this chapter), the results are clear: quite a lot of relations were found, especially due to the parent- and synonym contributions:

- 4 instances of the *has_deeper_explanation* relationship type were found.
- 1 instance of the *external* relationship type was found.
- 21 parent relations were found, some duplicates and bi-directional occurrences were removed.
- 23 synonym relations were found.
- 0 prerequisite relations were found.

Note that even though zero prerequisites were found for the four documents, we believe that the rules of Heuristics HR-6 and HR-7 are promising. During testing, the rules correctly caught the prerequisite relationship as outlined in Figure 5–9 and Figure 5–10. Anyway, the figures are scary. Altogether 69 relations were found. In comparison, only 11 unique concepts were found, which means that on the average each concept has more than 6 relations. Of course these numbers may differ on larger collections. Another important factor is the construction of the sample documents. More profound testing and possible revision of the rules is needed, but beyond the scope of this thesis. Questionable, therefore, how else can we secure that the system performs tolerably well given the proposed framework? Remember that the author has been quite silent all the way through, except when creating the DSL and pushing some buttons in order to start the processes of building the domain model. Let us therefore close up with putting a load on the author.

6.3.3 Document concepts vs. element concepts

An early commitment as part of the process towards a domain model, was to separate document concepts from element concepts. The decision was based on the observation that documents have a context superior to its sections and other subordinate documents. Indeed, the separation gave birth to the three different sets of Heuristics for conceptualization, and later on proved convenient for the identification of the parent relationship type.

As a side-effect, the separation also produced knowledge sources of different granularity. First, the original documents were preserved and conceptualised with the second attribute set to `html`, as exemplified in the `kw_source` fact

```
kw_source(softwar, html, "file3.html",
        [interfac, intellig, user, direct, manipul], "file3.html").
```

but also the elements of each document were written to individual smaller files, like in

```
        kw_source(dictionari, p, “file2.html#elm5”, [act, behalf] , “file2.html”).
```

From the latter example we see that both the `ID` and the element type `p` indicates that this knowledge source originally was part of another document.

Is it useful to preserve both knowledge sources in the new knowledge base? One goal of adaptation is to tailor new documents that differ from the original ones, if more suitable for the user. The issue on document concepts versus element concepts are important as they are an essential part of the foundation for the work. We believed the system would *benefit* from this separation, but another view is that the resulting domain model actually will be limited *due to* the choice. The separation could in the worst case preserve the original hierarchy and structure so much that the resulting domain model will not capture anything but the original link structure. On the other hand, as indicated in Figure 6–7 there are two chooses for how to view the present knowledge base:

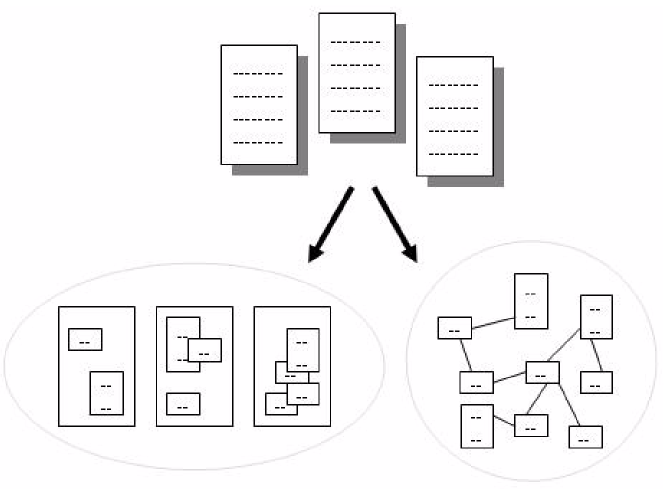


Figure 6–7: The original domain consists of documents. After the conceptualization, a knowledge base consisting of documents and elements (to the left) are the result. Another view is to regard the knowledge sources stemming from the elements only (to the right).

Preserving original document structure along with an identification of elements, are useful as it provides flexibility and the choice to suddenly “turn off” the adaptive guidance provided from the system. The other view is that of the left side from the figure, indicating that only the knowledge sources stemming from element concepts should be regarded when generating adaptive documents. Note that in this case the document concepts are still necessary for the identification of the parent relations. We conclude that more empirical research is needed in order to evaluate the positive and negative consequences of the separation of document concepts and element concepts, along with the possibilities for adaptation.

6.3.4 Clean up the mess before the guests pay you a visit

According to Zibell, Klare's notion of "useful information" is still useful for web designers today, almost 40 years later [*Zibelloo*]. For traditional systems, knowledge of the user's knowledge level would help the author to suggest the correct depth of concepts and organization of content. The resulting domain will be static. For an adaptive hypertext system, the dynamic organization depends upon how well the DM can be constructed. Starting with the DSL construction, the following processes of analysis, element division and generation of relations are up to the system, with a resulting domain model as the gift to the author. As seen, the number of relations will probably get out of hand, but this does not mean that those found are not useful for the author. Obviously, the task of removing incorrect or unwanted relations and/or concepts, are a task a lot more pleasant for the author, than creating an entire domain model from scratch. Additionally, recall that all the important elements are sorted out of their original files and saved in a knowledge base, with the references kept intact in the `kw_source` facts. In other words, the present prototype of the system will indeed be expected to help the author in preparing a static domain for dynamic, adaptive presentations.

Due to the possible imperfect outcome of the proposed DM and its implicit assumption of well organised, consistently marked up documents, we would also have to require a proper use of tags in order for the AHS to perform at its best. At least the author should prepare existing documents for the system initially before the automation starts[1]. For future adaptation we propose that new documents should be composed with some guidelines in mind:

- Hyperlinks should be labeled so as to predict the content of the target.
- The domain should be well structured and decomposed into smaller documents.
- Knowledge sources on the same subject should be varied in different elements like lists, images, tables, text etc. in order to enrichen the knowledge base.
- Emphasizers should be used with caution.
- If applied, the title element and meta information should predict the content of each individual document well, and be varied among the domain documents.

The prototype does not handle poorly tagged documents, nested lists, and simply skips advanced features like scripts and css.

[1] The Tidy program is available from www.w3.org and helps to fix incorrect use of mark up tags.

7 Conclusion

In this thesis we have explored means for assisting an author on the way towards a domain model for an adaptive hypertext system. After identifying the main goal of this thesis and exploring related research from the field of adaptivity, we proposed an overall architecture for an AHS and then zeroed in on techniques for achieving a domain model. More specifically, we found that the tags play an important role regarding content of the documents, leading to Heuristics which secure the extraction of concepts and the identification of relations among the concepts selected. After implementing the conceptualizer module of the system, we briefly sketched how easily relatively powerful adaptive presentations and variations of adaptivity can be realised, facilitated by the use of the `PROLOG` language.

When hiking on the future trip towards adaptive systems, we believe the first hill to climb is that of convincing authors that making an adaptive system is actually manageable. From the author's point of view, the effort should not be put into redesigning existing documents, but rather in planning what sort of adaptive behaviour to use.

The main goal of this research was to design an adaptive hypertext system that could bridge user gap of knowledge in the context of a domain, developing methods that facilitated the use of existing HTML-documents. In order to achieve the desired flexibility, a domain model was needed. An analysis of some randomly picked documents led to the observation that the tags actually would help to sort out important keywords, leading to the hypothesis that the relative importance of the elements combined with some basic IR-techniques would yield positive results for the system. The rules that guided this process was based on Heuristics, which in turn were drawn up from the characteristics and patterns of a relatively small set of hypertext documents, along with our previous experience with HTML.

After implementing a prototype, an evaluation of the system in total revealed that some rules produced far too many relations. Moreover, one might assume that the methodology outlined in this thesis would not work equally well with all collections of documents. Still we believe that the idea of combining the results of rules and IR-techniques with more intelligent reasoning is fruitful, and that authors will benefit from our methods.

Finally, as a digression, how can an AHS based on our design be published online? The task should, theoretically, not be as complex as it might seem. Small scripts (e.g. written in `PHP`) can arrange for the layout in order to prepare the system for the Web. The plan is straightforward:

- The adaptive engine reasons on which knowledge sources are most appropriate to present and instructs the script with a list of the respective IDs (i.e. filenames) to be presented.
- The script in turn stitches together documents simply by including files from the knowledge base into a standard template HTML file.

The decision to use the combination of `C` and `PROLOG` was based on a wish for integrating the AHS routines into an existing agent based framework. Due to the ease for `C` to communicate with both `PHP` and `OpenGL` both generation of adaptive hypertext documents for the Internet today and interesting 2D or 3D visualizations of the domain model should be achievable using agents.

As mentioned several times throughout the thesis, more research and intensive testing is obviously needed in order to improve both the rules and some of the framework, and validate the system. In addition to an online implementation of the work of this thesis, the following are subject to future research:

- The development of an inference engine able to develop new rules to guide the conceptualization
- The embedding of other techniques for extracting information in order to increment the reliability of the system
- Applying better Heuristics for conceptualization that would fit a larger audience
- Test whether the development of various adaptive behaviours is indeed as simple and promising as predicted in this thesis.
- Justify the selection of the split between document concepts and element concepts, possibly trying other solutions.
- Embed techniques for visualization of the domain model and the user conceptual models, e.g. allowing users to navigate their own mental models in a three dimensional landscape.

References

Arens+93 Arens, Y. - Hovy, E. - Vossers, M. *On the Knowledge Underlying Multimedia Presentations*, Intelligent Multimedia Interfaces, Maybury, M. (ed.), AAAI/MIT Press, 1993 pp.280-306.

Ashish+97 Ashish, N. - Knoblock, C. *Semi-Automatic Wrapper Generation for Internet Information Sources*, In Conference on Cooperative Information Systems, 1997, pp. 160-169.

Ashman+99 Ashman, H - Simpson, R. *Computing Surveys' Electronic Symposium on Hypertext and Hypermedia: Editorial*, ACM Computing Surveys, Vol. 31 (4), 1999.

Beck+96 Beck, J. - Stern, M. - Haugsjaa, E. *Applications of AI in Education*. ACM/ Crossroads, Vol 3, Issue 1, 1996.

Berners-Lee95 Berners-Lee, T. *Style Guide for Online Hypertext*. www.w3.org/provider/ style/all.html, unpublished, 1995.

Berners-Lee96 Berners-Lee, T. *The World Wide Web: Past Present and Future*. www.w3.org/people/Berners-Lee/1996/ppf.html, unpublished1996.

Bodner+00 Bodner, R. - Chignell, M. *Dynamic Hypertext: Querying and Linking*. In ACM Computing Surveys, Vol 31 (4), 1999, art 15.

Bradshaw97 Bradshaw, J. *An introduction to Software Agents*. AAAI Press/MIT Press, 1997, pp. 4-27.

Brusilovsky96 Brusilovsky, P. *Methods and Techniques of Adaptive Hypermedia*. User modeling and user-adapted interaction vol 6, Kluwer Academic Publishers, 1996, pp. 87-129.

Brusilovsky01 Brusilovsky, P. *Adaptive Hypermedia*. User modeling and user-adapted interaction 11, Kluwer Academic Publishers, 2001, pp. 87-110.

Brusilovsky+02 Brusilovsky, P. - Maybury, M. *From Adaptive Hypermedia to the Adaptive Web*. Communications of the ACM, Vol 45 (5), 2002, pp. 30-33.

Bush45 Bush, V. *As We May Think*. Atlantic Monthly, 1945.

Chin91 Chin, D. *Intelligent Interfaces as Agents*, Intelligent User Interfaces, Sullivan, J. - Tyler, S. (eds.), 1991, pp. 177-206.

Cohen00 Cohen, W. *Automatically extracting features for concept learning from the web*, Seventeenth International Conference on Machine Learning, 2000.

da Silva+98 da Silva, D - van Durm, R - Duval, E - Olivié, H. *Adaptive navigational facilities in educational hypermedia*, UK Conference on hypertext, 1998, pp.291-292.

Davis+93 Davis, R - Shrobe, H - Szolovits, P. *What is a Knowledge Representation?*, AI Magazine, 14(1), 1993, pp 17-33.

De Bra+99 De Bra, P - Brusilovski, P - Houben, G. *Adaptive Hypermedia: From Systems to Framework*, ACM Computing Surveys 31, 4, www.acm.org/pubs/articles/ journals/surveys/1999-31-4es/a12-de_bra/a12-de_bra.pdf , 1999.

Denning99 Denning, P. *Computer Science: the Discipline*, In Encyclopedia of Computer Science, 2000 Edition, 1999.

Fink+97 Fink, J - Kobsa, A, Schreck, J. *Personalized Hypermedia Information Provision through Adaptive and Adaptable System Features: User Modeling, Privacy and Security Issues*. Proceedings of International Conference on User Modelling, Italy, 1997.

Fischer00 Fischer, G. *User Modeling in Human Computer Interaction*, User modeling and user-adapted interaction 11, Kluwer Academic Publishers, 2001, pp. 65-86.

Foley+88 Foley, J. - Gibbs, C. - Kim, W. - Kovacevic, S. A Knowledge-Based User Interface Management System, In Proceedings of the 1988 Conference on Human Factors in Computer Systems (CHI'88), ACM, Inc., 1988, pp. 67-72.

Frakes+92 Frakes, W. - Baeza-Yates, R. *Information Retrieval, Data Structures & Algorithms*, Prentice Hall, New Jersey, 1992.

Griffin97 Griffin, D. *Adaptivity and the Cohesive Nature of Hypertext Activity*. Dissertation Proposal, 1997.

Höök+99 Höök, K. - Svensson, M. *Evaluating Adaptive Navigation Support*. In Maybury (Ed.) ACM Press, Proceedings of the IUI'99, 1999, pp 187.

Jennings99 Jennings, N. *Agent-Based Computing: Promise and Perils*, Proceedings of Sixteenth International Joint Conference on Artifical Inelligence, IJCAI-99, Vol. 2, 1999, pp 1429-1436.

Kietz+00 Kietz, J - Maedche, A - Volz, R. *A Method for Semi-Automatic Ontology Acquisition from a Corporate Intranet*, In Aussenac-Gilles N., Biébow B., Szulman S., EKAW'2000 Workshop Ontologies and Texts, Juan-les-Pins, 2000, pp 37-50.

Kobsa+94 Kobsa, A. - Müller, D. - Nill, A. *KN-AHS: An Adaptive Hyphertext Client of The User Modeling System BGP-MS*, Proceedings of the Fourth International Conference on User Modeling, Hynnais, Association of Computing Machinery, 1994, pp.99-105.

Kobsa01 Kobsa, A. *Generic User Modeling Systems*. In User Modeling and User-Adapted Interaction, 11(1-2), 2001, pp. 49-63.

Koons+93 Koons, D. - Sparrell, C. - Thórisson, K. *Integrating Simultaneous Input from Speech, Gaze, and Hand Gestures*, Intelligent Multimedia Interfaces, Maybury, M. (ed)., AAAI/MIT Press, 1993, pp. 257-276.

Kules00 Kules. *User Modeling For Adaptive And Adaptable Software Systems.* Available from http://www.otal.umd.edu/UUGuide/wmk/

Maes94 Maes, P. *Agents that reduce work and information overload*. Communication of the ACM July 1994/Vol. 37, No. 7, 1994, pp. 31-40.

Malinowski+92 Malinowski, U. - Kuhme, T. - Dietrich, H. - Schneider-Hufschmidt, M. *A taxonomi of adaptive user interfaces*. In Monk, Diaper, Harrison, editors, People and Computers VII, Cambridge University Press, 1992, pp. 391-414.

Marinilli+99 Marinilli, M. - Micarelli, A. - Sciarrone, F. *A Case Based Approach to Adaptive Information Filtering for the WWW*, in Proc. of the 2nd Workshop on Adaptive Systems and User Modeling on the World Wide Web, Sixth International Conference on User Modeling UM-99, Canada, 1999.

Maybury+98 Maybury, M. - Wahlster, W (eds.). *Intelligent User Interfaces: An Introduction*. Readings in Intelligent User Interfaces, Morgan Kaufmann Publisher, 1998, pp. 1-13.

McTear93 McTear, M. *User Modelling for Adaptive Computer Systems: A Survey of Recent Developments*. Artificial Intelligence Review. (Special issue on User Modelling, edited by M McTear), Vol 7, 1993, pp. 157-184.

Milosavljevic+98 Milosavljevic, M. - Oberlander, J. *Dynamic Hypertext Catalogues: Helping Users to Help Themselves*. Proc 9th ACM Conference on Hypertext and Hypermedia, Pittsburg, 1998.

Möller Möller, R. *User Interface Management Systems: The CLIM Perspectiv*e, University of Hamburg,http://kogs-www.informatik.uni-hamburg.de/~moeller/uims-clim/clim-intro.html

Moore+01 Moore, A. - Brailsford, T. - Craig, D. *Adaptive navigational facilities in educational hypermedia*, Hypertext '01, Denmark, 2001.

Myers94 Myers, B. *Challenges of HCI Design and Implementation*, ACM Interactions, Jan 1994, pp. 73-83.

Nechest+93 Nechest R. - Foley J. - Szekely P. - Sukaviriya P. - Luo P. - Kovacevic S. - Hudson S. Knowledgeable Development Environments using Shared Design Models, Proceedings of Intelligent User Interfaces IUI93, 1993, pp. 63-93.

Nielsen92 Nielsen, J. *The Usability Engineering Lifecycle*, IEEE Computer, March 1992, pp. 12-22.

Perkowitz+00 Perkowitz, M. - Etzioni, O. *Adaptive Web Sites*. Communications of the ACM, Vol 43 (8), 2000, pp. 152-158.

Rautenbach+90 Rautenbach, P. - Totterdell, P. *Adaptation as a problem of design*. In Adaptive User Interfaces, London: Academic Press, Browne, D. - Totterdell, P. - Norman, M. (eds.) 1990, pp. 59-84.

Rich79 Rich, E. *User Modeling via Stereotypes*, Cognitive Science. 3, 1979, pp. 329-354.

Schank95 Schank, R. *Information is Surprises*. www.edge.org/documents/ThirdCulture/q-Ch.9.html.

Schneiderman+97 Schneiderman, B, Maes, P. *Direct Manipulation vs Interface Agents*, ACM Interactions. Volume IV.6, Nov 1997. pp. 42-61.

Schneiderman00 Schneiderman, B. *Universal Usability*, Communications of the ACM. Volume 43, May 2000. pp. 85-91.

Seligman+91 Seligman, D. - Feiner, S. *Automated Generation of Intent-Based 3D illustrations*. Computer Graphics, 25(4), 1991, pp. 123-132.

Sharma01 Sharma, A. *A generic Architecture for User Modeling Systems and Adaptive web services*. Delhi College of Engineering, New Delhi.

Stefani+99 Stefani, A - Strapparava, C. *Exploiting NLP techniques to build user model for Web sites: the use of WordNet in SiteIF Project*. Proceedings of the 2nd Workshop on Adaptive Systems and User Modeling on the WWW, Canada, 1999, Edited by Peter Brusilovsky and Paul De Bra, pp. 13-20.

Sukaviriya+93 Sukaviriya, P. - Foley, J. *Supporting Adaptive Interfaces in a Knowledge-based User Interface Environment*. Proceedings of Intelligent User Interfaces IUI93, 1993, pp. 107-113.

Thomassen99 Thomassen, A. *Automating GOMS evaluation in agent based user interfaces*. Report, Norwegian University of Science and Technology, 1999.

Tveit01 Tveit, A. *A survey of Agent-Oriented Software Engineering*. Report, Norwegian University of Science and Technology, 2001.

Wahlster91 Wahlster, W. *User and Discourse Models for Multimodal Communication*. Sullivan, J. and Tyler, S.,(eds). Intelligent User Interfaces, ACM Press, 1991, pp. 45-67.

Wahlster+93 Wahlster, W. - André, E. - Finkler, W. - Profitlich, H. - Rist, T. *Plan-Based Integration of Natural Language and Graphics Generation*, Artificial Intelligence 63(1-2), Elsevier Science-NL, 1993, pp. 387-427.

Weber+97 Weber, B. - Specht, M. *User Modeling and Adaptive Navigation Support in www-based Tutoring Systems*, Proceedings of User Modeling '97, 1997, pp. 289-300.

Wu+01 Wu, H. - de Kort, E - De Bra, P. *Design issues for general-purpose adaptive hypermedia systems*, Proceedings of the ACM Conference on Hypertext and Hypermedia, Denmark 2001, pp. 141-150.

w3 Home page of the World Wide Web Consoritum, www.w3.org

whatis online computer dictionary, www.whatis.com

Zibell00 Zibell, K. *Klare's "Useful Information" is Useful for Web Designers*, ACM journal of Computer Documentation, vol 24 (3), 2000, pp. 141-147.

Appendix A - Source documents

file1.html

```
<HTML>
<HEAD>
    <TITLE>Agent overview</TITLE>
    <META>Software agents, agent definition</META>
</HEAD>

<BODY>

<H1>What is an agent?</H1>
There are several interpretations of the meaning behind the term <I>agent</I>, and
therefore the term is not a strong one. Two main approaches attempt to define an
agent:
<OL TYpe="I">
    <LI> <B>Ascription</B> made of a person (what they are). An agent has different
         meanings for two people. Sometimes the agent-based approach fits the
         expectations of the programmer, sometimes it does not and should not be
         used. It should take the situation into account and provide feasibilities
         like reasoning.

    <LI> <B>Description</B> of the attributes of an agent (what they do).  A
        software entity that functions continuosly, autonomously and inhabitated by
        other agents. This means it should learn from experience and cooperate with
         his friends. Here you can find some <A HREF="file2.html">attributes</A> of
         an agent
</OL>


Do you wonder <A HREF="file3.html">why software agents</A> are appealing?
<A HREF="file4.html#pattie">Pattie Maes</A> is often debating agenthood.

</BODY>
</HTML>
```

file2.html

```
<HTML>
<HEAD>
    <META>Agent characteristics</META>
</HEAD>

<BODY>

<H1>Agent characteristics</H1>

The following <B>attributes</B> are common to find in an agent:
  <ul>
    <li> <I>Reactivity</I> - ability to sense and then act on its environment (it
         reacts on some stimuli it senses)
    <li> <I>Proactivity</I> - ability to start something itself, autonomously
    <li> <I>Collaboration</I> - work in concert with others
    <li> <I>Communication</I> with a person should be non-symbolic, but rather
         natural language-like
    <li> Use of <I>models</I> to infer new knowledge
    <li> <I>Continuity</I> persistent over time
    <li> <I>Adapting</I> to its environment, and learning from experience
    <li> <I>Mobility</I> - move itself from one place to another
  </ul>

<H1>Definitions</H1>

Gilbert uses a three-dimensional space to characterize agenthood:
  <ol style='margin-top:0cm' >
    <li> Degree of Agency - how autonomous is the agent?
    <li> Degree of Mobility - how much travel from machine to machine does the
         agent do?
    <li> Degree of Intelligence - Is reasoning and learning provided?
  </ol>

<P>Nana uses another classification resulting in four possible agent-types:
   Smart, Collaborative, Collaborative that learns and Interface-agents. </P>

<p>Dictionary: One that <A HREF="file3.html">acts on your behalf</A></p>



</BODY>
</HTML>
```

file3.html

```
<HTML>
<HEAD>

    <META>Software agents, direct manipulation interface agent</META>
</HEAD>

<BODY>

<H1>Why software agents?</H1>

Two motivations:
<OL>
    <LI> <B>Simplifying distributed computing</B>. Today's applications only
         cooperate in the most basic ways (file transfer, DB-queries etc). The web
         has evoluted from this basic communication to the
         <EM ONCLICK="ordlisteVindu('../ordliste.html#adhoc')" STYLE="cursor:hand">
         ad-hoc</EM> to the encapsulated message passing systems, all meaning low
         levels of interoperability. There is a need for <B>intelligent
        cooperation</B> among systems to optimize the work-processes towards goals.
         To increase the level of interoperability in small systems, an agent could
         serve as a global <B>resource manager</B>. For larger systems,  embedding
         peer-agents for each system may increase intelligence.
    <LI> <B>Overcoming user interface problems</B>. Direct manipulation has
         limitations like
</OL>


<TABLE border=1 CELLSPACING=2 WIDTH="90%" ALIGN="center">
<TR><TH>Limitations of direct manipulation</TH><TH>Advantages of agents</TH></TR>
  <TR>
  <TD>
    <ul><li>large spaces to be searched
        <li>difficult to schedule tasks
        <li>hard to make basic actions higher-level ones
        <li>consistency means predictable interfaces, but this is not so for complex
        tasks
        <li>software is function oriented rather than concerned with context of the
        task and situation
        <li>repetetive actions are not learned
    </ul>
  </TD>

  <TD>
    <ul><li>search and filtering mechanisms of the agent run in the background, help
            constrain the search space
        <li>event-driven actions/wake up on response
        <li>share our goals, they don't simply process our commands
        <li>may work around unforseen problems
        <li>account for context of the user's tasks and situation
        <li>learn from repetetive patterns
    </ul>
  </TD>
  </TR>
</TABLE>

As <A HREF="file4.html">debated</A> in the article "Direct Manipulation vs
Interface Agents" the two are complementary rather than mutually exclusive.
It is difficult to find a golden way between proactive and reactive behavior.

</BODY>
</HTML>
```

file4.html

```
<HTML>
<HEAD>
    <META>Ben Schneiderman Pattie Maes Debating</META>
</HEAD>

<BODY>

<H1>Debating...</H1>

Ben:
<UL>
    <LI> Anthropomorphic interfaces are not the future of computing.
    <LI> Great that Pattie is moving away from the "agent as living entity on
         screen"-vision
    <LI> Collaborative filtering will be important in the future.
    <LI> Adaptive features should appear as non-adaptation to the user, to be
         predictable. So adaptation must not lead to unpredictability.
    <LI> The user need to feel he did the job himself (not some magical agent)
    <LI> Words like smart, agent, intelligent etc mislead the designer to leave out
         important things in the user interface.
    <LI> A good thing to make the user model available for the user, but that is not
         being done today in most agent-systems
    <LI> Speech (NL) is not the future because it make use of the short-term memory
         and working memory. This degrades the level of performance. You do problem
         solving better when you use direct manipulation than speech.
    <LI> When it comes to the issue of critical time-restricted systems that should
        avoid mistakes, I think the essence is in designing a very simple interface
         (<A HREF="../341/foley.html">according to Foley</A>)
    <LI> Even blind people may use direct manipulation, because they are strong at
         spatial processing.
    <LI> Agent litterature does not focus enough on the user interface!
</UL>

<A NAME="pattie"></A>
Pattie:
<UL>
    <LI> Agents could work below the table, with a nice, possibly direct
         manipulation interface, that the user sees.
    <LI> Important to distinguish <A HREF="file3.html">software agents</A> from
         other agents.
    <LI> The disagreement is mainly due to us focusing on different problem domains.
         Ben looks at a structured task-domain with professional users, while I
         focus with end-users that are novices in a dynamic domain.
    <LI> Agree that speech is difficult. A lot of ambiguity has to be solved. But
        the agent-approach could use speech in <A HREF="cubricon.html">multilingual
         input-features</A>.
    <LI> It is difficult for an agent to always do the right thing, so therefore i
         have focused on areas where things need not be 100 % correct.
    <LI> As complexity increases, so does the need for delegation.
</UL>

</BODY>
</HTML>
```

Appendix B - Results from conceptualization

Original document size: 289 words

The technique proposed is based on two interesting observations concerning the very nature of hyperspaces. Firstly, documents in a hyperspace don’t need a predefined curriculum (like text-books do), but allow for linkage among documents. This in turn leads to a more natural decomposition of large documents into smaller ones that are linked together. Secondly, formatting a document normally involves the identification of important concepts of the text, and these normally get emphasized in some way by the author in order to improve readability. Now the importance of the HTML-tags should come clear: As noted above, HTML-files contain the formatting tags along with the text. Assuming that the author have structured and linked the documents (intelligently) while marking up important concepts within a document, we now have another (untraditional) way of extracting information from a document regarding its content. Different tags have different purposes and therefore is of varying importance. Therefore a predefined base of rules that tells what to do when encountering tags, yields both power and flexibility to the process of analysing HTML-documents. New rules may be easily added to the base without changing the implemented adaptive system if the base is explicit to the system. An appealing strategy seems to be as follows: "parse the document until a useful tag is run into. Analyse the information within the tag based on the rules from the rulebase. Do the appropriate action as told by the rule (i.e. retag/conceptualize the information), and continue parsing. When the end of the document is reached, a list of concepts and relations should be the result. This list is the basis for constructing the semantic network. (Perhaps that list is the semantic network, if nodes are written to a prolog-knowledgebase as new concepts are encountered...?)

When stopwords and lexical analysis is perfomed, 133 terms remain

```
technique proposed based observations concerning nature hyperspaces firstly documents
hyperspace don predefined curriculum text books allow linkage documents leads natural
decomposition documents ones linked secondly formatting document normally involves
identification concepts text normally emphasized author improve readability importance
html tags noted html files contain formatting tags text assuming author structured linked
documents intelligently marking concepts document untraditional extracting information
document regarding content tags purposes varying importance predefined base rules tells
encountering tags yields power flexibility process analysing html documents rules easily
added base changing implemented adaptive system base explicit system appealing strategy
follows parse document useful tag run analyse information tag based rules rulebase
appropriate action told rule retag conceptualize information continue parsing document
reached list concepts relations result list basis constructing semantic network list
semantic network nodes written prolog knowledgebase concepts encountered
```

When stemming is applied, 133 words (46% of original size) remain

```
techniqu propos base observ concern natur hyperspac firstli docum hyperspac don predefin
curriculum text book allow linkag docum lead natur decomposit docum on link secondli
format docum normal involv identif concept text normal emphas author improv readabl
import html tag note html file contain format tag text assum author structur link docum
intellig mark concept docum untradit extract inform docum regard content tag purpos vary
import predefin base rule tell encount tag yield power flexibl process analys html docum
rule easili ad base chang implem adapt system base explicit system appeal strategi follow
pars docum us tag run analys inform tag base rule rulebas appropri action told rule retag
conceptu inform continu pars docum reach list concept relat result list basi construct
semant network list semant network node written prolog knowledgebas concept encount
```

Removing duplicates further improves slightly: 87 words (30% of the original size)

```
action ad adapt allow analys appeal appropri assum author base basi book chang concept
conceptu concern construct contain content continu curriculum decomposit docum don easili
emphas encount explicit extract file firstli flexibl follow format html hyperspac identif
implem import improv inform intellig involv knowledgebas lead link linkag list mark natur
network node normal note observ on pars power predefin process prolog propos purpos reach
readabl regard relat result retag rule rulebas run secondli semant strategi structur
system tag techniqu tell text told untradit us vary written yield
```

Example of Report:

Document candidates of file4.html

```
debat:  252
mae:  231
user:  109
interfac:  105
adapt:  103
domain:  103
intellig:  101
model:  101
:  51
patti:  34
ben:  33
schneiderman:  31
speech:  4
futur:  3
direct:  3
manipul:  3
design:  2
system:  2
memori:  2
solv:  2
focu:  2
focus:  2
difficult:  2
anthropomorph:  1
comput:  1
move:  1
live:  1
entiti:  1
screen:  1
vision:  1
collabor:  1
filter:  1
appear:  1
predict:  1
lead:  1
unpredict:  1
feel:  1
job:  1
magic:  1
word:  1
smart:  1
mislead:  1
leav:  1
avail:  1
nl:  1
short:  1
term:  1
degrad:  1
```

```
level:  1
perform:  1
come:  1
issu:  1
critic:  1
time:  1
restrict:  1
avoid:  1
mistak:  1
essenc:  1
simpl:  1
blind:  1
peopl:  1
spatial:  1
process:  1
litteratur:  1
below:  1
nice:  1
possibli:  1
distinguish:  1
disagr:  1
mainli:  1
due:  1
look:  1
structur:  1
task:  1
profession:  1
novic:  1
dynam:  1
agre:  1
lot:  1
ambigu:  1
approach:  1
correct:  1
complex:  1
increas:  1
deleg:  1
agent:  -90
featur:  -198
folei:  -198
accord:  -199
softwar:  -199
cubricon:  -199
multilingu:  -199
*********************************
```

Element candidates of file4.html#elm0 of elementtype: h1

```
debat:  -399
:  -399
-----------------
```

Element candidates of file4.html#elm1 of elementtype: ul

```
user:  106
agent:  105
interfac:  104
adapt:  103
intellig:  101
model:  101
futur:  3
design:  2
system:  2
speech:  2
memori:  2
direct:  2
manipul:  2
folei:  2
anthropomorph:  1
```

```
comput:  1
patti:  1
move:  1
live:  1
entiti:  1
screen:  1
vision:  1
collabor:  1
filter:  1
featur:  1
appear:  1
predict:  1
lead:  1
unpredict:  1
feel:  1
job:  1
magic:  1
word:  1
smart:  1
mislead:  1
leav:  1
avail:  1
nl:  1
short:  1
term:  1
degrad:  1
level:  1
perform:  1
solv:  1
come:  1
issu:  1
critic:  1
time:  1
restrict:  1
avoid:  1
mistak:  1
essenc:  1
simpl:  1
accord:  1
blind:  1
peopl:  1
spatial:  1
process:  1
litteratur:  1
focu:  1
:  -399
-----------------
```

Element candidates of file4.html#elm2 of elementtype: ul

```
agent:  105
user:  103
domain:  103
interfac:  101
focus:  2
speech:  2
difficult:  2
below:  1
nice:  1
possibli:  1
direct:  1
manipul:  1
distinguish:  1
softwar:  1
disagr:  1
mainli:  1
due:  1
ben:  1
look:  1
```

```
structur:  1
task:  1
profession:  1
focu:  1
novic:  1
dynam:  1
agre:  1
lot:  1
ambigu:  1
solv:  1
approach:  1
cubricon:  1
multilingu:  1
featur:  1
correct:  1
complex:  1
increas:  1
deleg:  1
:  -399
-----------------
```

Appendix C - Results from PROLOG rules

Facts produced by the PROLOG rules. Altogether, the relations, the concepts and the knowledge sources constitute the domain model.

We start by making concepts, and thereafter ask which relations were found.

```
?- reconsult('C:\\svend prolog\\hr1_2.pro')
yes
?- make_concept.
yes


?- concept(C).

C = definit ;
C = agent ;
C = agent ;
C = reactiv ;
C = autonom ;
C = interfac ;
C = dictionari ;
C = softwar ;
C = intellig ;
C = agent ;
C = debat ;
C = user ;
C = agent ;
C = foley ;
no
```

This is the output of the PROLOG environment used, but internally,
the facts hold the form listed below:

```
concept(agent).
concept(autonom).
concept(debat).
concept(definit).
concept(dictionari).
concept(foley ;
concept(intellig).
concept(interfac).
concept(reactiv).
concept(softwar).
concept(user).

?- reconsult('C:\\svend prolog\\hr1_2.pro')
yes
?- hr1.
yes
?- hr2.
yes
?- relation(R,E,T).

relation(has_deeper_explanation, definit, agent).
relation(has_deeper_explanation, definit, softwar).
relation(has_deeper_explanation, agent, softwar).
relation(has_deeper_explanation, softwar, debat).

relation(external, debat, foley).
```

The same procedure with finding the relations and asking which were found, yields the following results:

```
relation(parent, agent, reactiv).
relation(parent, agent, autom).
relation(parent, agent, dictionari).
relation(parent, agent, interfac).
relation(parent, agent, user).
relation(parent, autom, intellig).
relation(parent, debat, interfac).
relation(parent, debat, user).
relation(parent, debat, agent).
relation(parent, definit, agent).
relation(parent, definit, autom).
relation(parent, intellig, agent).
relation(parent, intellig, interfac).
relation(parent, intellig, user).
relation(parent, reactiv, autom).
relation(parent, softwar, intellig).
relation(parent, softwar, agent).
relation(parent, softwar, interfac).
relation(parent, softwar, intellig).
relation(parent, softwar, user).
relation(parent, user, interfac).

relation(synonym, agent, debat).
relation(synonym, agent, reactiv).
relation(synonym, agent, user).
relation(synonym, autonom, softwar).
relation(synonym, autonom, user).
relation(synonym, debat, user).
relation(synonym, definit, agent).
relation(synonym, definit, reactiv).
relation(synonym, intellig, agent).
relation(synonym, intellig, debat).
relation(synonym, intellig, user).
relation(synonym, interfac, agent).
relation(synonym, interfac, debat).
relation(synonym, interfac, intellig).
relation(synonym, interfac, softwar).
relation(synonym, interfac, user).
relation(synonym, reactiv, debat).
relation(synonym, reactiv, interfac).
relation(synonym, reactiv, user).
relation(synonym, softwar, agent).
relation(synonym, softwar, debat).
relation(synonym, softwar, intellig).
relation(synonym, softwar, user).
```

Finally, when asking for HR6 and HR7, nothing is found.

```
?- reconsult('C:\\svend prolog\\hr6.pro')
yes
?- hr6.

no
?- reconsult('C:\\svend prolog\\hr7.pro')
yes
?- hr7.

no
```

Appendix D - PROLOG implementation

```
kw_source(definit, html, "file1.html", [autonom, ascript, descript, mean, term, approach] , "file1.html").
kw_source(agent, ol, "file1.html#elm1", [autonom, ascript, descript, mean, sometim, attribut] , "file1.html").
kw_source(agent, html, "file2.html", [characterist, attribut, reactiv, proactiv, model, adapt] , "file2.html").
kw_source(reactiv, ul, "file2.html#elm1", [proactiv, model, adapt, autonom, knowledg, collabor] , "file2.html").
kw_source(autonom, ol, "file2.html#elm3", [intellig, degre, machin, margin, top] , "file2.html").
kw_source(interfac, p, "file2.html#elm4", [interfac, collabor, nana, classif, result, type] , "file2.html").
kw_source(dictionari, p, "file2.html#elm5", [act, behalf] , "file2.html").
kw_source(softwar, html, "file3.html", [interfac, intellig, user, direct, manipul] , "file3.html").
kw_source(intellig, ol, "file3.html#elm1", [user, interfac, agent, cooper, simplifi, distribut] , "file3.html").
kw_source(agent, table, "file3.html#elm2", [interfac, user, proactiv, reactiv, task, search] , "file3.html").
kw_source(debat, html, "file4.html", [mae, user, interfac, ben, adapt, domain] , "file4.html").
kw_source(user, ul, "file4.html#elm1", [agent, interfac, adapt, intellig, model, futur] , "file4.html").
kw_source(agent, ul, "file4.html#elm2", [user, domain, interfac, focus, speech, difficult] , "file4.html").
kw_source(foley, html, "../341/foley.html", [norman] , "../341/foley.html").

lowerList("file4.html", [speech, futur, direct, manipul, design, system]).
lowerList("file4.html#elm1", [design, system, speech, memori, direct]).
lowerList("file4.html#elm2", [softwar, disagr, mainli, due, ben]).


href("file1.html", "file2.html", "attributes").
href("file1.html", "file3.html", "why software agents").
href("file1.html", "file4.html#pattie", "pattie maes").
href("file2.html", "file3.html", "acts on your behalf").
href("file3.html", "file4.html", "direct vs interface agents").
href("file4.html", "file3.html", "software agents ").
href("file4.html", "../341/foley.html", "according to foley").
name("file4.html", "pattie", "pattie").


make_concept :-
     kw_source(ConceptName, _ , _ , _ , _ ) ,
     assert( concept(ConceptName) ) ,
     fail.
     % fail forces backtracking so that every combination is tried

make_concept.
     %when everything is tried and the above rule fails, this one succeeds

%------------------------------------------------------------------------
% HR-1: Hyper references between two knowledge entities somewhere within
% the domain are relations of type has_deeper_explanation
% between the corresponding concepts.
%---

hr1 :-
     href(From, To, _ ) ,
     kw_source(ConceptA, _ , From, _ , _ ) ,
     kw_source(ConceptB, _ , To, _ , _ ) ,
     assert( relation(has_deeper_explanation, ConceptA, ConceptB) ) ,
     fail.    % forces backtracking

hr1. % ensure success


%------------------------------------------------------------------------
% member is recursively defined and finds out whether an X is in a list.
% It is used thoroughly in the following.
%---

member(X, [X | Tail]).
member(X, [Head | Tail]) :-
        member(X, Tail).

%------------------------------------------------------------------------
% The domain is constrained to a list of valid filenames, here exemplified
% with three files only. The rule outsideDomain checks for membership in
% the domain list.
%---

domain( ["file1.html", "file2.html", "file3.html", "file4.html"] ).

outsideDomain(F) :-
     domain(D) ,
     not member(F, D).


%------------------------------------------------------------------------
% HR-2: External links with destinations outside of the domain
% are slightly different from internal ones, and the corresponding
% relations should be labelled as external.
%---

hr2 :-
href(From, To, _ ) ,
     outsideDomain(To) ,
     kw_source(ConceptA, _ , From, _ , _) ,
     kw_source(ConceptB, _ , To, _ , _) ,
     assert( relation(external, ConceptA, ConceptB) ),
     fail.    % forces backtracking

hr2. % ensure success
```

```
%------------------------------------------------------------------------
% HR-3: All element concepts found within a document are children
% of the document concept.
%---

hr3 :-
     kw_source(DocumentConcept, html, _ , _ , File) ,
     kw_source(ElementConcept, _ , _ , _ , File) ,
     not DocumentConcept = ElementConcept ,
     assert( relation(parent, DocumentConcept, ElementConcept) ) ,
     fail.    % forces backtracking

hr3. % ensure success


%------------------------------------------------------------------------
% member is recursively defined and finds out whether an X is in a list.
% It is used thoroughly in the following.
%---

member(X, [X | Tail]).
member(X, [Head | Tail]) :-
     member(X, Tail).

%------------------------------------------------------------------------
% HR-4: There is a parent relation if a member from a set of
% candidate concepts equals an already found concept.
%---

hr4 :-
     kw_source(ConceptA, _ , _ , CandidateList, _ ) ,
     kw_source(ConceptB, _ , _ , _ , _ ) ,
     not ConceptA = ConceptB ,   %  the two concepts should not be equal
     member(ConceptB, CandidateList) ,
     not relation(parent, ConceptA, ConceptB) , %only consider once
     assert( relation(parent, ConceptA, ConceptB) ) ,
     fail.    % forces backtracking

hr4. % ensure success



%------------------------------------------------------------------------
% Removes bi-directional relations
%---

removeBidirectional :-
     relation(Type, ConceptA, ConceptB) ,
     relation(Type, ConceptB, ConceptA) ,
     retract( relation(Type, ConceptB, ConceptA) ) ,  %remove one
     fail.    % forces backtracking

removeBidirectional. % ensure success




%------------------------------------------------------------------------
% HR-5: If two concepts have some joint members from their
% set of candidates, the concepts are synonymous.
%---

commonMember(List1, List2) :-
     member(Term, List1) ,
     member(Term, List2).

hr5 :-
     kw_source(ConceptA, _ , _ , CandidateListA, _ ) ,
     kw_source(ConceptB, _ , _ , CandidateListB, _ ) ,
     commonMember(CandidateListA, CandidateListB) ,
     not ConceptA = ConceptB ,
     not relation(synonym, ConceptA, ConceptB) ,
     assert( relation(synonym, ConceptA, ConceptB) ) ,
     fail.    % forces backtracking

hr5. % ensure success


make_concept :-
     kw_source(ConceptName, _ , _ , _ , _ ) ,
     assert( concept(ConceptName) ) ,
     fail.
     % fail forces backtracking so that every combination is tried

make_concept.
     %when everything is tried and the above rule fails, this one succeeds

%------------------------------------------------------------------------
% member is recursively defined and finds out whether an X is in a list.
% It is used thoroughly in the following.
%---

member(X, [X | Tail]).
member(X, [Head | Tail]) :-
     member(X, Tail).
```

```
%------------------------------------------------------------------------
% Lower score terms are listed as PROLOG facts, here illustrated with
% three examples. Note that the lists in lowerList and kw_source
% together constitute all the terms surviving lexical analysis.
%---

lowerList("file4.html", [speech, futur, direct, manipul, design, system]).
lowerList("file4.html#elm1", [design, system, speech, memori, direct]).
lowerList("file4.html#elm2", [softwar, disagr, mainli, due, ben]).

inLowerList(Term, Location) :-
     lowerList(Location, Z) ,
     member(Term, Z).


%------------------------------------------------------------------------
% The domain specific list contains terms that are regarded important,
% here exemplified with seven terms only.
%---

domainSpecificList( [hci, adapt, iui, user, model, domain, intellig] ).

inDsl(Term) :-
     domainSpecificList(DSL) ,
     member(Term, DSL).


%------------------------------------------------------------------------
% HR-6: A term in a knowledge source KS_1 which is neither selected as
% concept nor in the upper list of candidates, yet a member of the DSL
% and chosen as concept for another knowledge source KS_2,
% is prerequisite to the concept of the first knowledge source KS_1.
%---

hr6 :-
     inLowerList(Term, Location) ,
     inDsl(Term) ,
     kw_source(Concept , _ , Location, _ , _ ) ,
     concept(Term) ,
         %does the concept of the source equals the term?
     assert( relation(prerequisite, Term, Concept) ) ,
         %if so, then make a relation between the two.
     fail.    % forces backtracking

hr6. % ensure success


%------------------------------------------------------------------------
% There is a path between two nodes if there is an edge from the first
% node to an intermediate node, which again has a path to the destination
% node. This code also adds the nodes to the head of a list.
%---

path(From, To, [To] ) :-
     href(From, To, _ ).

path(From, To, [Intermediate|Tail] ) :-
     href(From, Intermediate, _ ),
     path(Intermediate, To, Tail).


%------------------------------------------------------------------------
% This rule asserts prerequisite relations from the intermediate
% members of a path to the last node
%---

addToSet([Head|Tail], ToConcept) :-
     kw_source(FromConcept, _ , Head, _ , _ ) ,
     not FromConcept = ToConcept ,  %write relation if the concepts differ
     assert( relation(prerequisite, FromConcept, ToConcept) ) ,
     addToSet(Tail, ToConcept) ,  %move down the path

addToSet([], _). %trivial case
```

```
%-------------------------------------------------------------------------
% HR-7: In the presence of a relation between two concepts, together with
% an unique, explicitly stated path through interlinked nodes connecting
% the two corresponding knowledge sources, there is a set of prerequisite
% relations between the concepts of each node (but the first one), and
% the destination of the path.
% Note that the procedure removes the original relationship type if
% some prerequisites are found
%---

hr7 :-
     relation(Type, ConA, ConB) , %if any relation exists
     kw_source(ConA , _ , From, _ , _ ) , %we go get the
     kw_source(ConB , _ , To, _ , _ ) ,   %destinations
     path(From, To, Visited) , %is there an explicitly stated path?
     addToSet(Visited, ConB) , %add all relations along this path
     retract( relation(Type, ConA, ConB) ) , %remove original relation
     fail.    % forces backtracking

hr7. % ensure success
```

```
%------------------------------------------------------------------------
% The rule next(Concept) finds a gap and builds a bridge, that is a list
% of concepts that can be visited next. Note that this routine does not
% validate the prerequisites of the concepts in the gap.
%---

deletepossible(Item, _ , [] ).
deletepossible(Item, [Item|Tail] , Tail).
deletepossible(Item, [Y|Tail] , [Y|Tail2] ) :-
     deletepossible(Item, Tail, Tail2).

insert(X, List, BiggerList) :-    %insert is the inverse of delete
     delete(X, BiggerList, List).

findGap( [] , R, R).  %trivial case
findGap( DMList, [Head|Tail], GapList):-
     deletepossible(Head, DMList, IntermediateList) ,
             %deletes common members
     findGap(Tail, IntermediateList, GapList).

all(Concept, ListIn, ListOut) :-   %go get all neighbours
     relation( _ , Concept, Next) ,
     not member(Concept, ListIn) ,
     all(Concept, [Next|ListIn], ListOut).
all(C, List, List). %trivial case

getNeighbour(Concept, DMList) :-  %get neigbours to a concept in DM
     all(Concept, [], DMList).

userKnows(Concept, UMList) :-    %get concepts known to the user.
     all2(Concept, [], UMList).  %almost equal to all, except that
                                 %relation in all is the DM relation,
                                 %and relation i all2 is UM relation.

next(Concept) :-
     getNeighboursDM(Concept, DMList) ,   %list all neighbours from DM
     userKnows(Concept, UMList) ,  % build list of neighb. known to user
     findGap(DMList, UMList, GapList) ,  %subtract UM from DM
```

Appendix E - Implementation of conceptualizer

```
/*******************************   conceptualization.c  ********************************
 *
 *  Purpose: Program to walk through a domain and identify concepts at both
 *           document and element level according to heuristical rules
 *  Date:   June 30th-July 2nd, 2002
 *  Input:  List of files in the domain
 *  Output:  Prolog files on the form:
 *               node(software, file3.html, [agents, agent, user, ]).
 *               node(agents, file3.html#sub0, [software, ]).
 *               node(interface, file3.html#sub1, [user, intelligent, agents, ]).
 *               etc...
 *           Report files with all information (one file for each document)
 *  Uses:    ruleadt.h- implementation of rules
 *  Notes:   Temporary files are used due to easy manipulation with stringlists
 *           The functions ReportDocCandidates, ReportElmCandidates, InitPrologFile
 *           and ReportPrologFile, are used in main.
***************************************************************************************/

#include <stdio.h>
#include <string.h>
#include "ruleadt.h"


/***************************************************************************************
 *  Name:    ReportDocCandidates
 *  Purpose: Update reportfile with all candidates and values in a document
***************************************************************************************/
ReportDocCandidates(CVR *record, char filename[NAME_SIZE],
                    char current[NAME_SIZE], int boundary){
  char tuple[NAME_SIZE];
  CVR *temp;
  FILE *reportfile;
  int i = 0;

  reportfile = fopen(filename, "a+");
  fprintf(reportfile, "\t\t\t Document candidates of %s \n", current);
  strcpy(tuple, "");
  temp=record;  //don't want to miss the array for good

  while (i++ < boundary){
    sprintf(tuple, "%s:  %d\n", temp->candidate, temp->value);
    fputs(tuple, reportfile);
    temp++;
  }//print
  fputs("**********************************\n\n\n", reportfile);
  fclose(reportfile);
}//ReportDocCandidates


/***************************************************************************************
 *  Name:    ReportElmCandidates
 *  Purpose: Update reportfile with all candidates and values in an element
***************************************************************************************/
ReportElmCandidates(CVR *record, char filename[NAME_SIZE],
                    char id[NAME_SIZE], char tag[TAG_SIZE], int boundary){
  char tuple[NAME_SIZE];
  CVR *temp;
  FILE *reportfile;
  int i = 0;

  reportfile = fopen(filename, "a+");
  fprintf(reportfile, "\t\t\t Element candidates of %s of elementtype: %s \n", id, tag);
  strcpy(tuple, "");
  temp=record;  //don't want to miss the array for good!

  while (i++ < boundary){
    sprintf(tuple, "%s:  %d\n", temp->candidate, temp->value);
    fputs(tuple, reportfile);
    temp++;
  }//print
  fputs("-----------------\n\n\n", reportfile);
  fclose(reportfile);
}//ReportElmCandidates


/***************************************************************************************
 *  Name:    InitPrologfiles
 *  Purpose: Writes the values as a comment to the prolog-file.
***************************************************************************************/
InitPrologfiles(char f[NAME_SIZE], char f2[NAME_SIZE],
                int H1, int H3, int H4, int H5a, int H5b, int H6, int H8){
  FILE *fp, *fp2;

  fp = fopen(f, "w");
  fprintf(fp, "%% -----------------------------------------------------------------\n");
  fprintf(fp, "%% The points associated with the Heuristics: \n");
  fprintf(fp, "%% \t HC-1  (DSL)         = %d \t\t HC-2  (term frequency)   = TF \n", H1);
  fprintf(fp, "%% \t HC-3  (emphasizers) = %d \t\t HC-4  (meta information) = %d \n", H3, H4);
  fprintf(fp, "%% \t HC-5a (headings)    = %d \t\t HC-5b (punish links)     = %d \n", H5a, H5b);
  fprintf(fp, "%% \t HC-6  (linked to)   = %d \t\t HC-8  (punish doc_con)   = %d \n", H6, H8);
  fprintf(fp, "%% Attributes: (concept, element type, location, candidate list)\n");
  fprintf(fp, "%% -----------------------------------------------------------------\n\n");
  fclose(fp);

  fp2 = fopen(f2, "w");
  fprintf(fp2, "%% -----------------------------------------------------------------\n");
  fprintf(fp2, "%% All hyper references in the domain \n");
  fprintf(fp2, "%% Attributes: (from, to, tag)\n");
  fprintf(fp2, "%% -----------------------------------------------------------------\n\n");
  fclose(fp2);
} //InitPrologfiles


/**********************************************************************************************
 *  Name:    ReportPrologFile
 *  Purpose: Writes concept to file in prolog-format. Should have the form:
 *           node(concept, element-type, loc, [cand1, cand2, ...], filename).
 *           in order to represent the node, the location and other candidates
 *  Note:    a bit tricky to achieve the correct layout form
 *  Returns: True if this element was successfully written to the file
*********************************************************************************************/
int ReportPrologFile(char fprolog[NAME_SIZE], char fprolog_non[NAME_SIZE],
                     char tagName[TAG_SIZE], char location[NAME_SIZE],
                     CVR *array, int boundary, char tempfile[NAME_SIZE],
                     char combfile[NAME_SIZE], char thisfilename[NAME_SIZE]){
  FILE *fp;
  char high[NAME_SIZE];
  char lowerstring[NAME_SIZE];
  char candidates[NAME_SIZE * 3];
  char noncandidates[NAME_SIZE *6];
  char concept[NAME_SIZE];
  int p=0; //argument of SelectConcept
  int i=0;
  int listsize=7;//number of candidates produced is listsize-1

  strcpy(high, ""); strcpy(candidates, ""); strcpy(lowerstring, ""); strcpy(noncandidates, "");

  if (boundary<=1) return;

    /* Else, go get the concept unless the combination tag+concept found before */
  do {
    SelectConcept(array, i++, concept, &p);  //highest value = concept
  }
  while ( IsCombination(concept, tagName, combfile) && (i<listsize)  );

    /* Next, get first candidate and add to string */
  do {
    SelectConcept(array, i++, high, &p); //get candidate no 1
  }
  while (0 == strcmp(high,"") && i<listsize);
  if (p<=0)
    return FALSE;//don't print if value is less than zero
  strcat(candidates, high);

    /* Get other candidates with high scores */
  while( i<listsize && boundary > 0){
    SelectConcept(array, i++, high,  &p);  //pick next candidate
    if (0 == strcmp(high,""))
      continue;//don't consider empty elements
    strcat(candidates, ", ");
    strcat(candidates, high);
  }//find additional candidates

    /* Write information to files */
  fp = fopen(fprolog, "a+");
  fprintf(fp, "knowledge_source(%s, %s, \"%s\", [%s], \"%s\").\n",
                                concept, tagName, location, candidates, thisfilename);
  fclose(fp);

  fp = fopen(tempfile, "w");//conceptfile is used later
  fputs(concept, fp); fputs("\n\n", fp);  //very impt with two \n due to stringlist requirement
  fclose(fp);
  return TRUE;
}//ReportPrologFile
```

```
/********************************************************************************************
 *  Name: Main
 *  Purpose: Walk through the domain and get concepts, candidates and write
 *  information to files
 *  Plan:
 *    1. Initialise
 *      a:  Prepare files to be used
 *      b:  Read the weights of the rules (extend to "from file")
 *      c:  Get all links
 *    2. For each document, do (according to the Heuristics for finding concepts)
 *      H2: Get term frequency (uses stemming)
 *      H1: Check membership in DSL (implemented as opposite of a stoplist)
 *      H3: Identify emphasizers
 *      H4: Check for meta information and title
 *      H5a:Top level headings are important
 *      H5b:Remove href candidates from consideration
 *      H6: Check whether this document is linked to
 *      H7: If the same concept and type occur for two elements, then choose another concept
 *      (H9:Decide level of abstraction / analyse headings used - not implemented
 *      Write concept with highest value and ID to prolog file
 *   3. For each element within a document, do (if level of abstraction allows it to)
 *      H1: Check membership in DSL (anti-stoplist)
 *      H2: Get term frequency (may introduce stemming here)
 *      H3: Identify emphasizers
 *      H8: Remove occurences of the document concept selected
 *      (H10:Lists, tables and images (concept in image is "alt" or filename) - not implemented
 *      Write information on concepts, ID, tag and upper candidate list to prolog file
 *      Write additional information to report file
 *    Repeat loop
 *
 *  Notes: Heuristic H7 is implemented as a rule inside function ReportPrologFile
************************************************************************/

int main( void ) {

/* declaring filepointers */
  FILE *prologfile,  *domaindocs, *tempfp, *docfile;

/* declaring variables to hold filenames */
  char current[NAME_SIZE];            //the name of the current document
  char f1[NAME_SIZE], f2[NAME_SIZE], f3[NAME_SIZE], f4[NAME_SIZE],
       f5[NAME_SIZE], f6[NAME_SIZE], f7[NAME_SIZE];     //temporary results
  char dsl[NAME_SIZE];                //domain specific list
  char cf[NAME_SIZE];                 //file with concepts selected
  char freport[NAME_SIZE];            //reporting all info found
  char ft[NAME_SIZE];                 //temporary file of no use at all
  char fprolog[NAME_SIZE];            //holds concepts found in prolog format
  char fprolog_non[NAME_SIZE];        //holds lowerList candidates in prolog format
  char fprologdomain[NAME_SIZE];      //holds all filenames of the domain in prolog format
  char fdomain[NAME_SIZE];            //all filenames in the domain
  char fweight[NAME_SIZE];            //weights associated with the rules
  char fdomainhrefs[NAME_SIZE];       //all hyper references in a domain
  char fprologhrefs[NAME_SIZE];       //all href information in prolog format
  char combinations[NAME_SIZE];       //for the combination concept+tag
  char newfilename[NAME_SIZE];        //new filename that element is written to

/* declaring variables to hold strings */
  char elementid[NAME_SIZE];          //ID for an element
  char tagName[TAG_SIZE];
  char document[LINE_SIZE], element[LINE_SIZE]; //document and element in long lines

/* declaring variables to hold integers */
  int doc_used, elm_used, id_nr;   //global counters used in many routines
  int H1, H3, H4, H5a, H5b, H6, H8;//points associated with Heuristical rules
  int cvalue=0;                       //not used, but necessary as argument of SelectConcept

/* declaring other structures */
  CVR doc_conceptvalues[MAX_CANDIDATES];  //an array of records for concepts and values
  CVR elm_conceptvalues[MAX_CANDIDATES];  //the element array for concepts and values
  CVR *cand_pt, *elm_cand_pt;




/* Part 1: Initialising */
  cand_pt = &doc_conceptvalues[0];   //now cand_pt points to the first array-element
  elm_cand_pt = &elm_conceptvalues[0];  //same for the element pointer
  doc_used = elm_used = id_nr = 0;      //number of elements in array and elementcounter

/* Part 1a. Prepare files for use and init filenames */
  strcpy(dsl, "domain_specific_list.txt");
  strcpy(cf, "res/concepts_file.txt");          strcpy(ft, "res/t");
  strcpy(f1, "res/temp_bold.txt");              strcpy(f2, "res/temp_italic.txt");
  strcpy(f3, "res/temp_ahref.txt");             strcpy(f4, "res/temp_meta.txt");
  strcpy(f5, "res/temp_title.txt");             strcpy(f6, "res/temp_header.txt");
  strcpy(f7, "res/temp_linkedto.txt");          strcpy(combinations, "res/combinations.txt");
  strcpy(fdomain, "domaindocs.txt");            strcpy(fprolog, "res/prolog.pro");
  strcpy(fweight, "res/weights.txt");           strcpy(fdomainhrefs, "res/domainhrefs.txt");
  strcpy(fprologhrefs, "res/prologhrefs.pro");  strcpy(fprologdomain, "res/prologdomain.pro");

  tempfp = fopen(combinations, "w");
  fclose(tempfp);  //init this file to empty

  prologfile = fopen(fprolog, "a+");  //open prolog file

  (void) Stemmer(dsl); //all intermediate results are stemmed, so must the DSL be

/* Part 1b. Assign weights with the different Heuristics and initialise the Prologfiles */
  H1 = 100;  H3 = 10;  H4 = 30;  H5a = 20; H5b = -200; H6 = 200;  H8 = -400;
  InitPrologfiles(fprolog, fprologhrefs, H1, H3, H4, H5a, H5b, H6, H8);

/* Part 1c. Write all links in domain to file and confirm on screen */
  AllHrefsInDomain(fdomain, fdomainhrefs, fprologhrefs);
  printf("Hyper-refs written to the file domainhrefs.txt\n------\n");

/* Part 1d. Write all filenames from the domain to prolog understandable form */
  DomainToProlog(fdomain, fprologdomain);




/* Part 2: Collect the document concepts, walk through the domain */
  domaindocs = fopen(fdomain, "r");
  while ( fgets(current, NAME_SIZE, domaindocs) != NULL){
    strip_slash_n(current);  //must remove \n from filename.html\n
    FileToStr(current, document); //read document into one single string
    WriteInfoToFiles(document, f1, f2, f3, f4, f5, f6);

    InitArray(document, doc_conceptvalues, MAX_CANDIDATES, &doc_used);  // H2 - term frequency

    AssignPoints(doc_conceptvalues, dsl, H1,  doc_used);                // H1  - DSL
    AssignPoints(doc_conceptvalues, f1,  H3,  doc_used);                // H3  - bold
    AssignPoints(doc_conceptvalues, f2,  H3,  doc_used);                // H3  - italic
    AssignPoints(doc_conceptvalues, f3,  H5b, doc_used);                // H5b - punish links
    AssignPoints(doc_conceptvalues, f4,  H4,  doc_used);                // H4  - meta
    AssignPoints(doc_conceptvalues, f5,  H4,  doc_used);                // H4  - title
    AssignPoints(doc_conceptvalues, f6,  H5a, doc_used);                // H5a - Headings level 1
    LinkedTo(current, f7, fdomainhrefs);                                // is this linked to?
    AssignPoints(doc_conceptvalues, f7,  H6,  doc_used);                // H6  - linked to

    BubbleSort(doc_conceptvalues, doc_used);//conceptvalues == &conceptvalues[0]
    sprintf(freport, "res/report_%s.txt", current);                     // "report_file1.html"
    ReportDocCandidates(doc_conceptvalues, freport, current, doc_used);
    (void) ReportPrologFile(fprolog, fprolog_non, "html", current,
                            doc_conceptvalues, doc_used, cf, combinations, current);




/* Part 3: Walk through each section, search for element concepts, still in outer loop */
    docfile = fopen(current, "r");
    id_nr = 0;
    //start an inner loop
    while (GetNextElement(docfile, element, tagName) > 0 ) {  //tag found
       InitArray(element, elm_conceptvalues, MAX_CANDIDATES, &elm_used); // H2 - term frequency
       WriteInfoToFiles(element, f1, f2, f3, f4, f5, f6); // not all info will be used...
       AssignPoints(elm_conceptvalues, dsl, H1, elm_used);//H1  - DSL
       AssignPoints(elm_conceptvalues, f1,  H3, elm_used); //H3  - bold
       AssignPoints(elm_conceptvalues, f2,  H3, elm_used);//H3  - italic
       AssignPoints(elm_conceptvalues, cf,  H8, elm_used);//H8  - remove doc_concept

       BubbleSort(elm_conceptvalues, elm_used);
       Id(current, &id_nr, elementid);
       ReportElmCandidates(elm_conceptvalues, freport, elementid, tagName, elm_used);
       if (ReportPrologFile(fprolog, fprolog_non, tagName, elementid,
                           elm_conceptvalues, elm_used, ft, combinations, current) ) {
           strcpy(newfilename, "");
           sprintf(newfilename, "kwbase/%s", elementid);  //make string "kwbase/elementid"
           tempfp = fopen(newfilename, "w");
           fputs(element, tempfp);
           fclose(tempfp);
       }//if, store in knowledge base

    }//end inner while for the elements
    fclose(docfile);

    printf("The result is written to file %s\n", freport);
  } //while get domaindocs

  fclose(domaindocs);  //close files
  fclose(prologfile);

}//end of main program.
```

```
/*******************************   parseradt.h   *********************************

    Purpose: ADT for extracting elements and information from an HTML-document

*******************************************************************************/

/****************************   Public Constants   ****************************/
#include "constants.h"  //has all public constants
#include "ruleadt.h"

/****************************   Public Routines   *****************************/
externint  AllImages(char input[], char output[], int *counter);
extern  int  AntiStoplist(char inputfile[], char output[], char antiFilename[], int *counter);
extern  void FileToStr(char filename[NAME_SIZE], char output[]);
extern  int  GetFirstTag(char input[], char output[], char tag[]);
extern  int  GetNextElement(FILE *docfile, char output[], char tag[]);
extern  int Id(char docname[ID_SIZE], int *counter, char tagid[NAME_SIZE]);
extern  int  LexicalAnalysis(char input[], char output[], char FileOut[NAME_SIZE], int *counter);
extern int  SubElementTerms(char tag[TAG_SIZE], char input[], char output[], char tempfile[NAME_SIZE],
int *size);
extern  int  TagsFromFile(char filename[], char tag[TAG_SIZE], char pro[LINE_SIZE], char
ut2[LINE_SIZE]);
extern  int  TagsFromString(char tag[TAG_SIZE], char input[], char output[]);
extern  int  WordCount(char filename[], int *size);
extern  int  Stemmer(char filename[NAME_SIZE]);



/*******************************   ruleadt.h   *********************************/

/****************************   Public Constants   ****************************/
#include "constants.h" //has all public constants
#include "strlist.h"

/******************************   Public Types   ******************************/

struct _HeuristicRecord {
    char heuristicId[HEUR_SIZE];/* Heuristic identifier */
    int  points;/* Points associated with the Heuristic */
               }  ;
typedef struct _HeuristicRecord HR;

struct _ConceptValueRecord {
    char candidate[NAME_SIZE];
    int  value;
    };
typedef struct _ConceptValueRecord CVR;

struct _ElementRecord {
    char id[ID_SIZE];/* ID for which this tag is retagged */
    char tagName[TAG_SIZE];/* the name of the element */
    char conceptName[NAME_SIZE];/* the concept name of the element */
    char content[LINE_SIZE];/* the original content of the element */
    char href[LINE_SIZE]; /* does this element contain a href? */
    char tempfile[NAME_SIZE];/* reference to a temporary file with words */
    int  stopcount;/* number of remaining words after lexical analysis */
    char stopwords[LINE_SIZE];/* all the remaining words after lexical analysis */
    int  subcount;/* number of words in identified sub-element */
    char subwords[LINE_SIZE];/* words in sub-element */
    int  anticount; /* number of words after anti-stoplist */
    char antiwords[LINE_SIZE];/* words after anti-stoplist */
    char maxwords[LINE_SIZE];/* words with more than two occurrences */
    int  maxcount;/* number of maximum occurrences of a word in the string */
               }  ;
typedef struct _ElementRecord ER;

/****************************   Public Routines   *****************************/
extern int  OlderParser(char docname[NAME_SIZE]);
extern void AllHrefsInDomain(char domainfilename[NAME_SIZE],
                             char linkfile[NAME_SIZE], char prologfilename[NAME_SIZE]);
extern void strip_slash_n(char temp[]);
extern void InitArray(char current[NAME_SIZE], CVR *conceptvalues, int arraysize, int *used);
extern int  MaxWords(char tempfile[], char output[], int *maximum ,
                    CVR *termfrequency, int *used);
extern void AssignPoints(CVR *candidatevalues, char incomingfile[], int points, int boundary);
extern void BubbleSort(CVR record[], int max);
extern void WriteInfoToFiles(char element[LINE_SIZE], char file1[NAME_SIZE],
                             char file2[NAME_SIZE], char file3[NAME_SIZE],
                             char file4[NAME_SIZE], char file5[NAME_SIZE],
                             char file6[NAME_SIZE] );
extern void SelectConcept(CVR *doc_conceptvalues, int location, char concept[], int *pt);
extern void LinkedTo(char current[NAME_SIZE], char fout[NAME_SIZE], char hrefs[NAME_SIZE]);
extern void DomainToProlog(char domainfile[NAME_SIZE], char output[NAME_SIZE]);
```

```
/********************************   stop.h   ********************************
    Purpose: Stop list DFA generator and driver module header.
    Notes:   This module implements a fast finite state machine generator,
             and a driver, for implementing stop list filters.
*****************************************************************************/

#ifndef STOP_H
#define STOP_H

#include "strlist.h"              /* this code relies on the StrList package */

/******************************   Public Types   ******************************/

typedef struct _DfaStruct *DFA;      /* Deterministic Finite Automaton object */

/****************************   Public Routines   *****************************/

#ifdef __STDC__

extern DFA   BuildDFA( StrList words );
extern char *GetTerm( FILE *stream, DFA machine, int size, char *output );

#else

extern DFA   BuildDFA();
extern char *GetTerm();

#endif
#endif


/*******************************   strlist.h   *********************************

    Purpose: Simple string list abstract data type module header

    Notes:   This module implements a straightforward string ordered list
             abstract data type.  It is optimized for appending and deleting
             from the end of the list.  Since they are ordered lists, string
             lists may be sorted, and their members are addressed by ordinal
             position (starting from 0).
******************************************************************************/

#ifndef STRLIST_H
#define STRLIST_H

/****************************   Public Constants   ****************************/

#define NULL_INDEX                     -1        /* invalid string index */

/******************************   Public Types   ******************************/

typedef struct _StrListStruct *StrList;          /* the base string list type */

/****************************   Public Routines   *****************************/

#ifdef __STDC__

extern void    StrListAppend( StrList list, char *string );
extern void    StrListAppendFile( StrList list, char *filename );
extern StrList StrListCreate( void );
extern void    StrListDestroy( StrList list );
extern int     StrListEqual( StrList list1, StrList list2 );
extern int     StrListElementEqual( int pos, StrList list1, StrList list2 );
extern int     StrListCount( int pos, StrList list1, StrList list2 );
extern char *  StrListPeek( StrList list, int index );
extern int     StrListSize( StrList list );
extern void    StrListSort( StrList list );
extern void    StrListUnique( StrList list );
extern int     StrListMember(char term[], StrList list);

#else

extern void    StrListAppend( /* list, string */ );
extern void    StrListAppendFile( /* list, filename */ );
extern StrList StrListCreate( /* void */ );
extern void    StrListDestroy( /* list */ );
extern int     StrListEqual( /* list1, list2 */ );
extern int     StrListElementEqual( /* pos list1, list2 */ );
extern int     StrListCount( /*pos, list1, list2 */ );
extern char *  StrListPeek( /* list, index */ );
extern int     StrListSize( /* list */ );
extern void    StrListSort( /* list */ );
extern void    StrListUnique( /* list */ );
extern int     StrListMember(/* char term[], StrList list*/ );

#endif
#endif
```

```
/*******************************   stem.h   ***********************************

   Purpose: Header file for an implementation of the Porter stemming
            algorithm.

   Notes:   This module implemnts a fast stemming function whose results
            are about as good as any other.
******************************************************************************/

#ifndef STEM_H
#define STEM_H


/******************************************************************************/
/****************************   Public Routines   *****************************/

#ifdef __STDC__

extern int Stem( char *word );      /* returns 1 on success, 0 otherwise */

#else

extern int Stem();

#endif

#endif



/*******************************   constants.h   *********************************/


/****************************   Public Constants   ****************************/
#define HEUR_SIZE 5
#define EOS  '\0'
#define FALSE  0
#define TRUE   1
#define LINE_SIZE 50000// how many characters can an element hold
#define TAG_SIZE    20// how many characters in a tag-name
#define NAME_SIZE    50// number of characters in a file / concept name
#define DOC_SIZE 100000// size of a document
#define MAX_CANDIDATES 1000// maximum number of candidate concepts
#define ID_SIZENAME_SIZE+7// size of ID
#define TEMP_FILE_NAME "temp_output.txt"
```

```
/********************************   ruleadt.c   ******************************/

#include <stdio.h>
#include <string.h>

#include "ruleadt.h"
#include "constants.h"

void strip_slash_n(char temp[]){
  int i = strlen(temp); //locate end of string
  temp[i-1] = EOS;//replace \n with \0
} //strip slash n



/*********************************************************************
 *  Name:    AllHrefsInDomain
 *  Purpose: searches all files in a domain and gets all the links
 *           The result is written to two files: one in prolog-format (prolog)
 *           and one for temporar use throughout the parsing (linkfile).
 *  Calls:   TagsFromFile() from parseradt.c
 *  Plan:
 *   1. Get next filename from domain-file
 *   2. Parse current file for hrefs
 *   3. Write result to file
*********************************************************************/

void AllHrefsInDomain(char domainfilename[NAME_SIZE], char linkfile[NAME_SIZE], char
prologfilename[NAME_SIZE]){
  char tagsearch[TAG_SIZE];
  char result[LINE_SIZE];
  char resultpro[LINE_SIZE];
  FILE *outputfile, *domainfile, *prologfile;
  char current[NAME_SIZE];

  domainfile = fopen(domainfilename, "r"); //prepare to parse domain
  outputfile = fopen(linkfile, "w");  //make sure it is empty
  fclose(outputfile);
  while ( fgets(current, 50, domainfile) != NULL){
    strip_slash_n(current);  //must remove \n from the string filename.html\n

    strcpy(result, "");strcpy(resultpro, "");
    strcpy(tagsearch, "A");

    outputfile = fopen(linkfile, "a+");prologfile = fopen(prologfilename, "a+");

    (void) TagsFromFile(current, tagsearch, resultpro, result);
    if (result) {
    fprintf(outputfile, "%s", result);fprintf(prologfile, "%s", resultpro);
    }
    fclose(outputfile);fclose(prologfile);
  }//while domain is parsed
  fclose(domainfile);
}//AllHrefsInDomain


/*********************************************************************
 *  Name:    BubbleSort
 *  Purpose: Sorts an array of records based on the value-field
 *           into ascending order
 *  Notes:   Ascending order, that is 1-2-3-...-n
*********************************************************************/

void BubbleSort(CVR record[], int max){
int x, y;
CVR temp;//temporary record
// bubble sort the array

  for (x=0; x < max-1; x++)
    for (y=0; y < max-x-1; y++)
        if (record[y+1].value > record[y].value){  //y, y+1 for ascending
            temp = record[y];
            record[y] = record[y+1];
            record[y+1] = temp;
        }//if
}//BubbleSort


/*********************************************************************
 *  Name:    DomainToProlog
 *  Purpose: Writes all filenames of a domain to an output file in
 *           prolog-format.
 *  Notes:   Receives the name of a file with one filename on each line.
*********************************************************************/

void DomainToProlog(char domainfile[NAME_SIZE], char output[NAME_SIZE]){
  FILE *fp;
  char tempstring[LINE_SIZE], current[NAME_SIZE];

  strcpy(tempstring, ""); strcpy(current, "");
  strcat(tempstring, "domain( [");

  fp = fopen(domainfile, "r");
  while ( fgets(current, NAME_SIZE, fp) != NULL){
     strip_slash_n(current);  //must remove \n from filename.html\n
     strcat(tempstring, "\"");
     strcat(tempstring, current);
     strcat(tempstring, "\", ");
  } //while
  strcat(tempstring, "]).");
  fclose(fp);
  fp = fopen(output, "w");
  fprintf(fp, tempstring);
  fclose(fp);
}//DomainToProlog

/************************************************************************
 *  Name:    InitArray
 *  Purpose: Builds a list of all candidate concepts based on the input
 *           string and assigns values according to term frequencies
 *   Calls:   LexicalAnalysis from parseradt.c and MaxWords
 *   Plan:
 *    1. Init array with <candidate, value> tuples set to zero
 *    2. Run lexical analysis
 *    3. Stem the terms
 *    4. Count number of words. Put each <term, frequency> tuple in array
 *    5. Return information found
************************************************************************/

void InitArray(char input[LINE_SIZE], CVR *conceptvalues, int arraysize, int *used){
  char stopwords[LINE_SIZE], maxstring[LINE_SIZE];
  CVR *temp;
  int p=0, i=0;
  int count=0;
  strcpy(stopwords, ""); strcpy(maxstring, "");
  *used = 0;  //how many fields in array is used

  temp = conceptvalues;
  for (i=0; i<arraysize; i++){
    strcpy(temp->candidate, "");  //blank'em all
    temp->value = 0;
    temp++;
  }
  (void) LexicalAnalysis(input, stopwords, "res/temp_lexical.txt", &p);  // 2
  (void) Stemmer("res/temp_lexical.txt");   // 3
  (void) MaxWords("res/temp_lexical.txt", maxstring, &count, conceptvalues, used); // 4
}//InitArray


/************************************************************************
 *  Name:     MaxWords
 *  Purpose:  Identifies all terms in tempfile[] with more than two occurrences
 *             and writes them to the string output[], and overwrites tempfile
 *            with a list of unique words.
 *   Modified: New modified version also includes:
 *              Writes tuples <term, frequency> to array and returns how much
 *            of the array is used.
 *   Returns:  Maximum number of words
 *   Note:     Incoming file should have one word on each line, for instance
 *             by having been exposed to lexical analysis first.
 *   Plan:
 *    1. Make a string list based on the input file
 *    2. Count the number of each word and save result
 *    3. Build array of all candidates with tuples <term, frequency>
 *    4. Return the largest number
************************************************************************/

int MaxWords(char tempfile[], char output[], int *maximum , CVR *termfrequency, int *used) {
   char *term;     // for the next term from the input line
   StrList words, taltOpp;   // string lists of all the words on a file
   int size, antall, i;// counters

     /* Part 1: Create a list of words from a file */
   words = StrListCreate();
   StrListAppendFile( words, tempfile); // tempfile opened/closed inside function
   size = StrListSize( words );

     /* Part 2: Count the number of words and save output */
   taltOpp = StrListCreate();//for the words already counted
   antall = 0;
   *maximum = 0;
   for (i=0; i<size; i++) {  //list traversal
     term = StrListPeek(words, i);
     if ( !StrListElementEqual(i, words, taltOpp) )  { //word not counted yet
    antall = StrListCount(i, words, words);  //function to count frequency of current word
    StrListAppend( taltOpp, term );  //marks current word as counted
        if (antall > 1) {   //save the information:
    sprintf(output, "%s%s - %d ", output, term, antall);  //concatenate through the first %s
    }
        if (antall > (*maximum)) {
            *maximum = antall;
        }

        /* Part 3: Array of <term, frequency> */
        strcpy((termfrequency->candidate), term);
        (termfrequency->value) = antall;
        termfrequency++;
        (*used)++;// update number of terms inserted into the array
    } //if count
   }//for
                  /* Part 4: Return maximum */
   return *maximum;
}


/***********************************************************************
 *  Name:    AssignPoints
 *  Purpose: Assigns "points" to those candidates from "candidatevalues"-array
 *           found in "incomingfile". "boundary" is used for efficient traversal
 *           and holds the number of elements in the array with candidates.
 *  Note:    Incoming file should have one word on each line in order to convert
 *           correctly to stringlist. This method is rather brilliant.
 *           IR-functions all write their results to temporary files, and these have
 *            terms that must be subsets of all words surviving lexical analysis.
 *            Therefore the membership test makes sense: If one of the candidates have
 *           membership in the incoming file, it should be given extra points
 *  Plan:
 *    1. Make stringlist of incoming file
 *    2. Check every candidate concept for membership in the list
 *    3. Update values in array and save pointer to first array-element
***********************************************************************/

void AssignPoints(CVR *candidatevalues, char incomingfile[], int points, int boundary){

  CVR *temp;
  char term[NAME_SIZE];     // for the next term from the input line
  StrList list;   // string list of all the words from an anti-stoplist file
  int i=0;

    /* 1 */
  list = StrListCreate();
  StrListAppendFile( list, incomingfile); // file is opened inside function
  if (StrListSize(list) <= 1)
    ;      // do nothing
  else {
    /* 2 */
    temp = candidatevalues;
    while (i++<boundary){ // boundary has the size of the array
      strcpy(term, temp->candidate);
      if (StrListMember(term, list))
    temp->value += points;// 3
      temp++;
    }//searching for membership
  }//else
}//AssignPoints


/***********************************************************************
 *  Name:    IsCombination
 *  Purpose: Checks for the combination "concept-tagName"
***********************************************************************/

int IsCombination(char concept[NAME_SIZE], char tagName[TAG_SIZE], char file[NAME_SIZE]){
  FILE *fp;
  StrList list;
  char comb[TAG_SIZE+NAME_SIZE];

  strcpy(comb, ""); sprintf(comb, "%s_%s", concept, tagName);

  list = StrListCreate();
  StrListAppendFile(list, file); // file is opened inside function
  if (StrListMember(comb, list))
    return TRUE;
  else {  //did not find a combination
    fp = fopen(file, "a+");
    fputs(comb, fp);  //update file
    fputs("\n", fp);
    fclose(fp);
    return FALSE;
  }//else
} //IsCombination
```

```
/*********************************************************************
 *  Name:    WriteInfoToFiles
 *  Purpose: Collects information about an incoming element:
 *            . emphasizer elements like <B>, <I> etc
 *            . links
 *            . meta and title
 *           The result is placed in files
 *   Note:    Result from sub-element is written to a file, one term for each line
 *   Plan:
 *    1. Get terms from sub-element specified. Returns stemmed terms.
 *    2. Output is not used, but result is written to file
*********************************************************************/

void WriteInfoToFiles(char element[LINE_SIZE], char file1[NAME_SIZE],
                      char file2[NAME_SIZE], char file3[NAME_SIZE],
                      char file4[NAME_SIZE], char file5[NAME_SIZE],
                      char file6[NAME_SIZE]   ){
  char output[LINE_SIZE];
  int p=0;
  strcpy(output, "");

    /* Part 1: Write information to files */
  (void) SubElementTerms("B", element, output, file1, &p);  // get all bold tags
  (void) SubElementTerms("I", element, output, file2, &p); // italic tags from the element
  (void) SubElementTerms("A", element, output, file3, &p); // a-href tags from the element
  (void) SubElementTerms("META", element, output, file4, &p);// meta tags
  (void) SubElementTerms("H1", element, output, file6, &p);// heading1
}//WriteInfoToFiles


/*********************************************************************
 *  Name:    LinkedTo
 *  Purpose: Identifies all href-elements from the file hrefs that contains
 *           the string current. All such occurrences are documents that refers to
 *           the current one, and hence the terms are candidate concepts. The result
 *           is placed in file with name fout
 *   Note:    Result from L.A. is written to a file fout, one term for each line
 *   Plan:
 *    1. Read line by line from hrefs
 *    2. Add up all lines that contains the string "current"
 *    3. Send these lines to Lexical Analysis
 *    4. Stem result
*********************************************************************/

void LinkedTo(char current[NAME_SIZE], char fout[NAME_SIZE], char hrefs[NAME_SIZE]){
  FILE *fp;
  char line[LINE_SIZE], temp[LINE_SIZE], notused[LINE_SIZE];
  int p=0;  //not used

  strcpy(line, ""); strcpy(temp, ""); strcpy(notused, "");
  fp = fopen(hrefs, "r");
  while (fgets(line, LINE_SIZE, fp) != NULL) {// 1.
    if (strstr(line, current) ) // 2.
    strcat(temp, line);
  }//while
  fclose(fp);
  LexicalAnalysis(temp, notused, fout, &p); // 3.
  (void) Stemmer(fout);   // 4

}//LinkedTo



/*********************************************************************
 *  Name:    SelectConcept
 *  Purpose: Selects the concept with the highest score
 *   Plan:
 *    1. Pick concept with highest value
*********************************************************************/

void SelectConcept(CVR *doc_conceptvalues, int location, char concept[], int *pt){

  //don't pick out if the field has an empty name
  if (0 == strcmp(doc_conceptvalues[location].candidate, "") )
    location++;

  strcpy(concept, doc_conceptvalues[location].candidate);
  (*pt) = doc_conceptvalues[location].value;
}//SelectConcept
```

```
/*********************************   parseradt.c   ********************************
    Written by:  Svend Andreas Horgen
    Date:        May 2002
    Purpose:     Functions to assist the parsing of an HTML-document
    Notes:       Does assume that the last text in a line is an end-tag
**************************************************************************************/

#include <stdio.h>
#include <string.h>
#include <ctype.h>
#include "parseradt.h"
#include "constants.h"
#include "strlist.h"
#include "stop.h"
#include "stem.h"

/************************ Private Function Declarations   ********************/

/* Function to make sure every record field is correctly initialized */
int InitRecord(ER *pt){
    /* string fields are initialized */
  strcpy(pt->id, "");  strcpy(pt->tagName, "");
  strcpy(pt->conceptName, "");  strcpy(pt->content, "");
  strcpy(pt->tempfile, TEMP_FILE_NAME); strcpy(pt->href, "");
  strcpy(pt->stopwords, "");  strcpy(pt->subwords, "");
  strcpy(pt->antiwords, "");strcpy(pt->maxwords, "");
    /* integer-fields are set to 0 */
  pt->stopcount = pt->subcount = pt->anticount = pt->maxcount = 0;
  return TRUE;
} //InitRecord

/* Function to produce the string </TAG> */
char *MakeEndTag(char tagName[]){
  char endtag[TAG_SIZE];
  sprintf(endtag, "</%s>", tagName);
  return endtag;
}//EndTag

/* Function to produce the string "<TAG>" */
char *MakeStartTag(char tagName[]){
  char starttag[TAG_SIZE];
  sprintf(starttag, "<%s>", tagName);
  return starttag;
}//StartTag

/* Function to produce the string "<TAG " (accounts for <TAG STYLE="blabla">) */
char *MakeStartTagOpen(char tagName[]){
  char starttag[TAG_SIZE];
  sprintf(starttag, "<%s ", tagName);
  return starttag;
}//StartTag

/* Function to convert a string to lower case */
char *StrToLower(char temp[]){
  int c;
  int i=0;

  while( (c=temp[i]) != '\0') {
    if (isupper(c) )
      temp[i] = tolower(c);
    i++;
  }
  return temp;
}//StrToLower

/* Function to grab the next token from an input stream */
static char *GetNextTerm( FILE *stream, int size, char *term )  {
   char *ptr;  /* for scanning through the term buffer */
   int ch;     /* current character during input scan */

           /* Part 1: Return NULL immediately if there is no input */
   if ( EOF == (ch = getc(stream)) ) return( NULL );

                  /* Part 2: Initialize the local variables */
   *term = EOS;
   ptr = term;

        /* Part 3: Main Loop: Put the next word into the term buffer */
   do {
         /* scan past any leading non-alphabetic characters */
      while ( (EOF != ch ) && !isalpha(ch) ) ch = getc( stream );

         /* copy input to output while reading alphabetic characters */
      while ( (EOF != ch ) && isalpha(ch) ){
         if ( ptr == (term+size-1) ) ptr = term;
         *ptr++ = ch;
         ch = getc( stream );
         }

         /* terminate the output buffer */
      *ptr = EOS;
      }
   while ( (EOF != ch) && !*term );

                    /* Part 4: Return the output buffer */
   return( term );
} /* GetNextTerm */


/************************ Public Function Declarations   ********************/

/************************************************************************
 *  Name:    AllImages
 *  Purpose: Puts all image-tags from the input-string to output[]
 *  Returns: Number of images found
 *   Note:    To find all images in a document, simply convert the entire
 *           document to a single string first, using FileToStr()
 *           Then call AllImages with this string.
 *   Plan:
 *   1. Initializing
 *    2. Identify all image-elements
 *    3. Return
***********************************************************************/

int AllImages(char input[], char output[], int *counter){

  char starttag[TAG_SIZE];
  char tempin[LINE_SIZE], tempout[LINE_SIZE];
  char *ptr, *start, *end;
  int j=0;

    /* Part 1: Initializing */
  strcpy(tempout, "");
  strcpy(tempin, input);
  (void) StrToLower(tempin);
  strcpy(starttag, MakeStartTagOpen("img"));  // <IMG SRC="imagename.jpg" ALT="optional">

    /* Part 2: Search for all sub-elements */
  ptr = tempin;
  while ( start = strstr(ptr, starttag)) {
       end = strstr(start, ">");//find matching ">"
       if (!end) //no endtag found
       break;
    /* Part 3: Catch the contents */
    end++;  //to move pointer behind ">"
       while (start != end)
       tempout[j++] = *start++ ; //shorthand notation that saves two more lines
       ptr = end;
       (*counter)++;  //else
  }//while

    /* Part 4: Update output string and return indication of success */
  tempout[j] = EOS; //if nothing found, then output string is empty since j=0
  strcpy(output, tempout);
  return *counter;
}//AllImages


/************************************************************************
 *  Name:    AntiStoplist
 *   Note:    This is the implementation of the DSL, domain specific list,
 *           which originally was called AntiStopList.
 *           The anti-stoplist itself must have just one \n after the last word.
 *  Purpose: Runs a file against an anti-stoplist given by antiFilename
 *           (An anti-stoplist contains domain specific keywords)
 *           Inputfile[] is a file that has gone through lexical
 *           analysis with one word for each line.
 *           Saves the words that occur both in the file and
 *           in the anti-stoplist to the string output[].
 *   Returns: Number of words found in anti-stoplist.
 *   Plan:
 *    1. Create string lists of the words from the anti-stoplist and inputfile
 *    2. Traverse the input-list and comparing with the anti-list
 *    3. Collect words found into the string pointed to by output[]
 *    4. Returning number of words
***********************************************************************/

int AntiStoplist(char inputfile[], char output[], char antiFilename[], int *counter){
   char term[50];     /* for the next term from the input line */
   StrList words, anti;   /* string lists of all the words on a file */
                          /* and all the the words from an anti-stoplist file */
   int size, i;/* the number of words from the file and the anti-stoplist */

                     /* Part 1: Create a list of words read from a file */
   words = StrListCreate();
   anti = StrListCreate();
   StrListAppendFile( words, inputfile ); // file is opened in function, one word per line
   StrListAppendFile( anti, antiFilename); // correct anti-list file opened in function
   StrListUnique(words); //removing duplicates

                    /* Part 2: Traverse the list while comparing with the anti-list */
   *counter = 0;
   size = StrListSize( words );
   for(i=0; i<size; i++) {
      strcpy(term, StrListPeek(words, i) );

                    /* Part 3: Collect words found */
     if (StrListElementEqual(i, words, anti)) {
                  //the term in position i in words is found in anti
     strcat(term, ", ");
     strcat(output, term);
     (*counter)++;
     } //if
   } //for

                    /* Part 4: Returning number of words */
   return *counter;
} //AntiStoplist


/**********************************************************************
 *  Name:    FileToStr
 *  Purpose: Writes an entire file to a single string
 *   Plan:
 *    1. Open file and initialize output string
 *    2. Read line by line while adding to the string
 *    3. Close file. String is accessible through pointer
**********************************************************************/

void FileToStr(char filename[NAME_SIZE], char output[]){
  FILE *docfile;
  char line[LINE_SIZE];

     /* Part 1: Open file and initialize output string */
  strcpy(output, "");
  docfile = fopen(filename, "r");

     /* Part 2: Read line by line while adding to the string */
  while (fgets(line, LINE_SIZE, docfile) != NULL)
     strcat(output, line);

     /* Part 3: Close file. String is accessible through pointer */
  (void) fclose(docfile);
}//FileToStr


/**********************************************************************
 *  Name:     GetFirstTag
 *  Purspose: Searches for the first occurrence of a tag in the input string
 *            given, and places the result in output[]
 *   Returns:  indication of success, TRUE if a tag was found
 *   Plan:
 *    1. Initializing + avoid tag-errors
 *    2. Search for element
 *    3. Catch the contents of this tag
 *    4. Updating output-string and return
**********************************************************************/

int GetFirstTag(char input[], char output[], char tag[]){
  char *start, *end;
  int size, j, a;
  char temp[LINE_SIZE], temp2[LINE_SIZE]; //in order not to overwrite the contents...
  char starttag[TAG_SIZE], starttagopen[TAG_SIZE], endtag[TAG_SIZE];
                  //need space for tags like </blockquote> etc

     /* Part 1: Initializing and dealing with possible tag-errors */
  (void) StrToLower(tag);
  strcpy(starttag, MakeStartTag(tag));
  strcpy(endtag, MakeEndTag(tag));
  strcpy(starttagopen, MakeStartTagOpen(tag));
                    //to avoid <BR>, <B>, <B Style="blabla"> error
  size = j = 0;  strcpy(temp, "");
  strcpy(temp2, input);

     /* Part 2: Search for element */
  (void) StrToLower(temp2);
  if ( (start =  strstr(temp2, starttag)) || (start = strstr(temp2, starttagopen))   ) {
        end = strstr(temp2, endtag);
        if (!end)  //no endtag found
        return FALSE;
        for (a=0; a<strlen(endtag); a++) end++;
```

```
    /* Part 3: Catch the contents of this tag */
       while (start != end)
       temp[j++] = *start++ ; //shorthand notation that saves two more lines
       temp[j] = EOS; //first copy sub-element, then place a '\0' to mark end of string

        /* Part 4: Updating output-string and indicate success */
       strcpy(output, temp);
       return TRUE;
  }//if
  return FALSE; //if no tag found...
}//GetFirstTag


/************************************************************************
 *  Name:    GetNextElement
 *  Purpose: Collects the text and tagname from the next element from a file stream.
 *  Returns: Indication of success.
 *  Note:    This function does case-insensitive tag-checking,
 *           so ul, Ul, uL and UL are all treated equally.
 *  Plan:
 *   1. Initializing
 *   2. Collect the text from the element found
 *   3. Return the text of the element
************************************************************************/

int GetNextElement(FILE *docfile, char output[], char tag[]) {

  char line[LINE_SIZE], ut[LINE_SIZE]; //can hold LINE_SIZE characters
  int funnet=FALSE;
  char endtag[TAG_SIZE];
  char temp[LINE_SIZE];  //helping to make tags lowercase

    /* Part 1. Initializing the temporary variables  */
  strcpy(ut, ""); strcpy(line, "");

    /* Part 2. Read line by line until end of document or a tag is found */
  while ( fgets(line, LINE_SIZE, docfile) != NULL ) {
    strcpy(temp, line);
    (void) StrToLower(temp);
    if ( strstr(temp, "<h1") ) {  // accounts for tags with attributes
      funnet = TRUE;  // since the > is missing in the test
      strcpy(tag, "h1"); }
    else if ( strstr(temp, "<h2") ) {
      funnet = TRUE;
      strcpy(tag, "h2"); }
    else if ( strstr(temp, "<h3") ) {
      funnet = TRUE;
      strcpy(tag, "h3"); }
    else if ( strstr(temp, "<table") ) {
      funnet = TRUE;
      strcpy(tag, "table"); }
    else if ( strstr(temp, "<ul") ) {
      funnet = TRUE;
      strcpy(tag, "ul"); }
    else if ( strstr(temp, "<ol") ) {
      funnet = TRUE;
      strcpy(tag, "ol"); }
    else if ( strstr(temp, "<p") ) {
      funnet = TRUE;
      strcpy(tag, "p");}
    else if ( strstr(temp, "<blockquote") ) {
      funnet = TRUE;
      strcpy(tag, "blockquote"); }///etc...

    if (funnet) break; //jump out of while-loop
  } //while

    /* Part 2. Collect the text from the element found */
  if (funnet) {
    strcat(ut, line);//add the line to the content of the element (ut)
    strcpy(endtag, MakeEndTag(tag)); // making a string with "</tag>"
    while ( ! strstr(temp, endtag) ) {
    if (fgets(line, LINE_SIZE, docfile) != NULL){ //inserts an /0 by the end of line.
      strcat(ut, line);   //adds line to the string ut
      strcpy(temp, line);
        (void) StrToLower(temp);
    }//if
    else break; //jump out of while if no more lines in file (that is no endtag found)
    } //while

    /* Part 3. Update and return the element found */
    strcpy(output, ut);  //the text of the element
    return 1;  //return and exit function immediately
  }//funnet

  return 0;  //happens when no tags found.
}//GetNextElement
```

```
/*********************************************************************
 *  Name:    Id
 *  Purpose: Constructs an unique ID for a string. The ID takes the following
 *           form: ID = DocumentName + "#" + "sub" + counter.
 *           Example: testfile.html#sub3
 *   Returns: Indication of success
 *   Note:    the form testfile.html#sub3 is used because then it is easy to
 *           generate links in the adaptive document later.
 *   Plan:
 *    1. Compose the ID
 *    2. Update the counter and return
*********************************************************************/

int Id(char docname[ID_SIZE], int *counter, char elementid[NAME_SIZE]) {

  strcpy(elementid, "");

    /* Part 1: Compose the ID */
  sprintf(elementid, "%s#elm%d", docname, *counter);

    /* Part 2: Update the counter first, then return */
  return ++(*counter);
}//Id


/*********************************************************************
 *  Name:    Lexical Analysis
 *  Purpose: Removes stopwords and performs a lexical analysis on the element
 *           specified by the paramater input[]
 *           Output is a pointer to a string where the result is placed.
 *           The result is also written to the file FileOut
 *   Returns: number of words remaining after the analysis
 *   Uses:    parseradt.c
 *   Note:    The temporary file is written over when a new element is being
 *           analysed, but its filename is necessary as input for the function
 *           WordCount.   (char input[] is a pointer)
 *   Plan:
 *    1. Work against a temporary file
 *    2. Building DFA for filtering stopwords
 *    3. Remove stopwords from input file
 *    4. Close input file and return
*********************************************************************/

int LexicalAnalysis(char input[], char output[], char FileOut[NAME_SIZE], int *counter) {

  FILE *stream;/* temporary file */
  FILE *outputfil;/* file to hold remaining words after lexical analysis */
  char FileIn[20];/* logical file name */
  char term[128];/* for the next term found */
  StrList words;/* the stop list filtered */
  DFA machine;/* build DFA from the stop list */

  strcpy(FileIn, "res/temp_lex.txt"); // input string is written to this file

    /* Part 1. Write the content of the string input[] to tempfile */
  stream = fopen(FileIn, "w");
  fputs(input, stream);
  (void) fclose(stream);

    /* Part 2. Building a DFA for filtering stopwords */
  words = StrListCreate();
  StrListAppendFile( words, "stop.wrd" );
  machine = BuildDFA( words ); //create a DFA

    /* Part 3. Remove stopwords from input-file, and preserving information
       in both a file and the output-string */
  if ( !(stream = fopen(FileIn,"r")) ) exit(1); //open temporary file
  outputfil = fopen(FileOut, "w");
  while ( (NULL != GetTerm(stream,machine,128,term)) ){
      fputs(term, outputfil);
      fputs("\n", outputfil);
      strcat(term, ", ");
      strcat(output, term);
  }

    /* Part 4: Closing the files */
  (void)fclose( stream );
  (void)fclose( outputfil );

    /* Part 5: Counting number of words on file and return */
  (void) WordCount(FileOut, counter);  //file that is subject to word count
  return *counter;
} //LexicalAnalysis
```

```
/*********************************************************************
 *  Name:    Stemmer
 *  Purpose: Porter stemming function. Takes a single filename and writes
 *           the stemmed terms to the file result.
 *  Calls:   GetNextTerm
 *  Uses:    Stem (from stem.c)
 *  Plan:
 *   1. Open the input file
 *   2. Process each word in the file
 *   3. Close the input file
*********************************************************************/

int Stemmer(char filename[NAME_SIZE]){
   char term[64];   /* for the next term from the input line */
   char temp[LINE_SIZE];
   FILE *stream;    /* where to read characters from */
   strcpy(temp, "");

                       /* Part 1: Open the input file og outputfil */
   if ( !(stream = fopen(filename,"r")) ) exit(1);  //The file to be stemmed

                  /* Part 2: Process each word in the file */
   while( GetNextTerm(stream,64,term) ) {
      if ( Stem(term) ) {
          strcat(temp, term);
          strcat(temp, "\n");
      } //if
   } //while
                 /* Part 3: Close the file */
   (void)fclose( stream );

                 /* Part 4: Write stemmed result back to file */
   stream = fopen(filename, "w");
   fputs(temp, stream);  /* resulting term to output */
   (void)fclose( stream );
   return(0);
} //Stemmer


/*********************************************************************
 *  Name:    SubElementTerms
 *  Purpose: Locates all sub-elements specified by tag[] in the string input[].
 *           If any found, they are all run through lexical analysis. The resulting
 *           terms are put in the string output[] and in the file tempfile and the number
 *           of elements placed in the integer size
 *  Returns: Success
 *  Calls:   TagsFromString() and LexicalAnalysis()
 *  Note:    Lexical analysis writes the result to a file and the string output[].
 *           Here only the output is used further
 *  Plan:
 *   1. Perform lexical analysis on the sub-elements found
 *   2. Return number of survivors (0 if no remains or if no sub-elements found)
*********************************************************************/

int SubElementTerms(char tag[TAG_SIZE], char input[], char output[],
                    char filein[NAME_SIZE], int *size){

  FILE *fp;
  char result[LINE_SIZE];
  strcpy(result, "");
                  /* Part 1: Identify sub-element and run lexical analysis */
  *size = 0;
  fp = fopen(filein, "w"); fclose(fp);
  if (TagsFromString(tag, input, result) ) {
    (void) LexicalAnalysis(result, output, filein, size); //filein is used further
    (void) Stemmer(filein);     //stem
  }
                  /* Part 2: Return success */
  return *size;
}//SubElement
```

```
/*********************************************************************
 *  Name:    TagsFromFile
 *  Purpose: gets all elements specified by "tag" from the file and writes the result
 *  to the string ut[]
 *  Returns: True if some elements found
 *  Plan:
 *   1. Initialize the variables
 *   2. Read file into one line
 *   3. Get the next tag searching from position ptr
 *   4. Get link target
 *   5. Format link information for PROLOG
 *   6. Return success
*********************************************************************/

int TagsFromFile(char filename[], char tag[TAG_SIZE],
                 char pro[LINE_SIZE], char ut2[LINE_SIZE]){
  char line[LINE_SIZE], temp[LINE_SIZE], resultat[LINE_SIZE], resultat2[LINE_SIZE];
  char *start, *ptr, *a, *b;
  char c[NAME_SIZE];//to hold link target
  int i, j;

    /* Part 1: Initializing the variables */
  strcpy(pro, ""); strcpy(line, ""); strcpy(temp, ""); strcpy(resultat, ""); strcpy(c, "");
  strcpy(ut2, "");

    /* Part 2: Read file into one long line */
  (void) FileToStr(filename, temp);
  (void) StrToLower(temp);

    /* Part 3: Get the next tag searching from position ptr */
  ptr = temp;
  start = NULL;
  while( GetFirstTag(ptr, resultat, tag) ) {

    /* Part 4: Move ptr to the start of remaining string, get link */
    start = strstr(ptr, resultat);
    while (ptr != start)
    ptr++;
    for (i=0; i<strlen(resultat); i++) ++ptr;
    strcat(ut2, resultat);  strcat(ut2, "\n");  //get link

    /* Part 5: Formatting in PROLOG format */
    strcpy(c, "");j = 0;
    a = strchr(resultat, '\"');  a++;  b = strchr(a, '\"');
    while (a != b)
       c[j++] = *a++ ; //shorthand notation that saves two more lines
    c[j] = EOS; //after sub-element is copied, then place a '\0' to mark end of string
    sprintf(resultat2, "*href(\"%s\", \"%s\", %s).\n", filename, c, resultat);
    strcat(pro, resultat2);
  }//while

    /* Part 6: Return success */
  if (start) //an element was found
    return TRUE;
  else
    return FALSE;
} //TagsFromFile


/*********************************************************************
 *  Name:    TagsFromString
 *  Purpose: Searches for all tags requested for in the input string given, and places
 *           the result in the string output[]
 *  Returns: indication of success, TRUE if at least one tag was found
 *  Plan:
 *   1. Initializing + avoid tag-errors
 *   2. Search for all tags
 *   3. Add the contents to the output string
 *   4. Updating output-string and return
*********************************************************************/

int TagsFromString(char tag[TAG_SIZE], char input[], char output[]){
  char *start, *end, *ptr;
  int  size, j, a, success;
  char temp[LINE_SIZE], temp2[LINE_SIZE];
  char starttag[TAG_SIZE], starttagopen[TAG_SIZE], endtag[TAG_SIZE];
             //need space for tags like </blockquote> etc

    /* Part 1: Initializing */
  size = j = 0;   success = FALSE;   strcpy(temp, "");
  strcpy(temp2, input);
  (void) StrToLower(tag);
  (void) StrToLower(temp2);
  strcpy(starttag, MakeStartTag(tag));
  strcpy(endtag, MakeEndTag(tag));
  strcpy(starttagopen, MakeStartTagOpen(tag)); //to avoid <BR>, <B>, <B Style="blabla"> error
```

```
    /* Part 2: Search for all sub-elements */
  ptr = temp2;
  while ( (start = strstr(ptr, starttag)) || (start = strstr(ptr, starttagopen)) ) {
       end = strstr(ptr, endtag);
        if (!end) //no endtag found
        break;
       success = TRUE;

    /* Part 3: Catch the contents */
    for (a=0; a<strlen(endtag); a++) ++end;
       while (start != end)
       temp[j++] = *start++ ; //shorthand notation that saves two more lines
       ptr = end;
  }//while

    /* Part 4: Update output string and return indication of success */
  temp[j] = EOS; //if nothing found, then output string is empty since j=0
  strcpy(output, temp);
  return success; //1 if found, 0 if not
}//TagsFromString


/*************************************************************************
 *  Name:    WordCount
 *  Purpose: Counts the number of words (lines) in a file. The advantage with this
 *           version of word-count is that is removes duplicate terms.
 *  Returns: number of words
 *  Notes: Assumes that the file has one word only in each line. Uses the ADT StrList
 *  Plan:
 *   1. Create string list from file
 *   2. Computing the size
 *   3. Returning the size
*************************************************************************/

int WordCount(char filename[], int *size){

  StrList words;/* string lists of all the words on a file  */

    /* Part 1: Create a string list from the content of a file */
  words = StrListCreate();
  StrListAppendFile( words, filename);
            //read from the specified file and add terms to the list

    /* Part 2: Computing size */
  StrListUnique(words);  //removes duplicates in the list
  *size = StrListSize( words ); //return number of words now as duplicates are removed
  (*size)--;  //somehow one line is empty at the end...

    /* Part 3: Returning the size */
  return *size;
} //WordCount
```

```
/********************************    stop.c   *******************************

    Purpose:     Stop list filter finite state machine generator and driver.

    Provenence:  Written by and unit tested by C. Fox, 1990.
                 Changed by C. Fox, July 1991.
                    - added ANSI C declarations
                    - branch tested to 95% coverage

    Notes:       This module implements a fast finite state machine
                 generator, and a driver, for implementing stop list
                 filters.  The strlist module is a simple string array
                 data type implementation.
**/

#include <stdio.h>
#include <ctype.h>
#include <string.h>
#include <malloc.h>

#include "stop.h"
#include "strlist.h"

/******************************************************************************/

#define FALSE                        0
#define TRUE                         1
#define EOS                        '\0'

         /**************   Character Classification   ***************/
         /* Tokenizing requires that ASCII be broken into character */
         /* classes distinguished for tokenizing.  Delimiter chars  */
         /* separate tokens.  Digits and letters make up the body   */
         /* of search terms.                                        */

typedef enum {

                DELIM_CH,              /* whitespace, punctuation, etc. */
                DIGIT_CH,              /* the digits */
                LETTER_CH,             /* upper and lower case */

            } CharClassType;

static CharClassType char_class[128] = {
      /* ^@ */  DELIM_CH,    /* ^A */  DELIM_CH,    /* ^B */  DELIM_CH,
      /* ^C */  DELIM_CH,    /* ^D */  DELIM_CH,    /* ^E */  DELIM_CH,
      /* ^F */  DELIM_CH,    /* ^G */  DELIM_CH,    /* ^H */  DELIM_CH,
      /* ^I */  DELIM_CH,    /* ^J */  DELIM_CH,    /* ^K */  DELIM_CH,
      /* ^L */  DELIM_CH,    /* ^M */  DELIM_CH,    /* ^N */  DELIM_CH,
      /* ^O */  DELIM_CH,    /* ^P */  DELIM_CH,    /* ^Q */  DELIM_CH,
      /* ^R */  DELIM_CH,    /* ^S */  DELIM_CH,    /* ^T */  DELIM_CH,
      /* ^U */  DELIM_CH,    /* ^V */  DELIM_CH,    /* ^W */  DELIM_CH,
      /* ^X */  DELIM_CH,    /* ^Y */  DELIM_CH,    /* ^Z */  DELIM_CH,
      /* ^[ */  DELIM_CH,    /* ^\ */  DELIM_CH,    /* ^] */  DELIM_CH,
      /* ^^ */  DELIM_CH,    /* ^_ */  DELIM_CH,    /*    */  DELIM_CH,
      /*  ! */  DELIM_CH,    /*  " */  DELIM_CH,    /*  # */  DELIM_CH,
      /*  $ */  DELIM_CH,    /*  % */  DELIM_CH,    /*  & */  DELIM_CH,
      /*  ' */  DELIM_CH,    /*  ( */  DELIM_CH,    /*  ) */  DELIM_CH,
      /*  * */  DELIM_CH,    /*  + */  DELIM_CH,    /*  , */  DELIM_CH,
      /*  - */  DELIM_CH,    /*  . */  DELIM_CH,    /*  / */  DELIM_CH,
      /*  0 */  DIGIT_CH,    /*  1 */  DIGIT_CH,    /*  2 */  DIGIT_CH,
      /*  3 */  DIGIT_CH,    /*  4 */  DIGIT_CH,    /*  5 */  DIGIT_CH,
      /*  6 */  DIGIT_CH,    /*  7 */  DIGIT_CH,    /*  8 */  DIGIT_CH,
      /*  9 */  DIGIT_CH,    /*  : */  DELIM_CH,    /*  ; */  DELIM_CH,
      /*  < */  DELIM_CH,    /*  = */  DELIM_CH,    /*  > */  DELIM_CH,
      /*  ? */  DELIM_CH,    /*  @ */  DELIM_CH,    /*  A */  LETTER_CH,
      /*  B */  LETTER_CH,   /*  C */  LETTER_CH,   /*  D */  LETTER_CH,
      /*  E */  LETTER_CH,   /*  F */  LETTER_CH,   /*  G */  LETTER_CH,
      /*  H */  LETTER_CH,   /*  I */  LETTER_CH,   /*  J */  LETTER_CH,
      /*  K */  LETTER_CH,   /*  L */  LETTER_CH,   /*  M */  LETTER_CH,
      /*  N */  LETTER_CH,   /*  O */  LETTER_CH,   /*  P */  LETTER_CH,
      /*  Q */  LETTER_CH,   /*  R */  LETTER_CH,   /*  S */  LETTER_CH,
      /*  T */  LETTER_CH,   /*  U */  LETTER_CH,   /*  V */  LETTER_CH,
      /*  W */  LETTER_CH,   /*  X */  LETTER_CH,   /*  Y */  LETTER_CH,
      /*  Z */  LETTER_CH,   /*  [ */  DELIM_CH,    /*  \ */  DELIM_CH,
      /*  ] */  DELIM_CH,    /*  ^ */  DELIM_CH,    /*  _ */  DELIM_CH,
      /*  ` */  DELIM_CH,    /*  a */  LETTER_CH,   /*  b */  LETTER_CH,
      /*  c */  LETTER_CH,   /*  d */  LETTER_CH,   /*  e */  LETTER_CH,
      /*  f */  LETTER_CH,   /*  g */  LETTER_CH,   /*  h */  LETTER_CH,
      /*  i */  LETTER_CH,   /*  j */  LETTER_CH,   /*  k */  LETTER_CH,
      /*  l */  LETTER_CH,   /*  m */  LETTER_CH,   /*  n */  LETTER_CH,
      /*  o */  LETTER_CH,   /*  p */  LETTER_CH,   /*  q */  LETTER_CH,
      /*  r */  LETTER_CH,   /*  s */  LETTER_CH,   /*  t */  LETTER_CH,
      /*  u */  LETTER_CH,   /*  v */  LETTER_CH,   /*  w */  LETTER_CH,
      /*  x */  LETTER_CH,   /*  y */  LETTER_CH,   /*  z */  LETTER_CH,
      /*  { */  DELIM_CH,    /*  | */  DELIM_CH,    /*  } */  DELIM_CH,
      /*  ~ */  DELIM_CH,    /* ^? */  DELIM_CH,                          };
```

```
         /**************   Character Case Conversion   **************/
         /* Term text must be accumulated in a single case.  This   */
         /* array is used to convert letter case but otherwise      */
         /* preserve characters.                                    */

static char convert_case[128] = {
      /* ^@ */    0,    /* ^A */    0,    /* ^B */    0,    /* ^C */    0,
      /* ^D */    0,    /* ^E */    0,    /* ^F */    0,    /* ^G */    0,
      /* ^H */    0,    /* ^I */    0,    /* ^J */    0,    /* ^K */    0,
      /* ^L */    0,    /* ^M */    0,    /* ^N */    0,    /* ^O */    0,
      /* ^P */    0,    /* ^Q */    0,    /* ^R */    0,    /* ^S */    0,
      /* ^T */    0,    /* ^U */    0,    /* ^V */    0,    /* ^W */    0,
      /* ^X */    0,    /* ^Y */    0,    /* ^Z */    0,    /* ^[ */    0,
      /* ^\ */    0,    /* ^] */    0,    /* ^^ */    0,    /* ^_ */    0,
      /*    */  ' ',    /*  ! */  '!',    /*  " */  '"',    /*  # */  '#',
      /*  $ */  '$',    /*  % */  '%',    /*  & */  '&',    /*  ' */ '\'',
      /*  ( */  '(',    /*  ) */  ')',    /*  * */  '*',    /*  + */  '+',
      /*  , */  ',',    /*  - */  '-',    /*  . */  '.',    /*  / */  '/',
      /*  0 */  '0',    /*  1 */  '1',    /*  2 */  '2',    /*  3 */  '3',
      /*  4 */  '4',    /*  5 */  '5',    /*  6 */  '6',    /*  7 */  '7',
      /*  8 */  '8',    /*  9 */  '9',    /*  : */  ':',    /*  ; */  ';',
      /*  < */  '<',    /*  = */  '=',    /*  > */  '>',    /*  ? */  '?',
      /*  @ */  '@',    /*  A */  'a',    /*  B */  'b',    /*  C */  'c',
      /*  D */  'd',    /*  E */  'e',    /*  F */  'f',    /*  G */  'g',
      /*  H */  'h',    /*  I */  'i',    /*  J */  'j',    /*  K */  'k',
      /*  L */  'l',    /*  M */  'm',    /*  N */  'n',    /*  O */  'o',
      /*  P */  'p',    /*  Q */  'q',    /*  R */  'r',    /*  S */  's',
      /*  T */  't',    /*  U */  'u',    /*  V */  'v',    /*  W */  'w',
      /*  X */  'x',    /*  Y */  'y',    /*  Z */  'z',    /*  [ */  '[',
      /*  \ */   92,    /*  ] */  ']',    /*  ^ */  '^',    /*  _ */  '_',
      /*  ` */  '`',    /*  a */  'a',    /*  b */  'b',    /*  c */  'c',
      /*  d */  'd',    /*  e */  'e',    /*  f */  'f',    /*  g */  'g',
      /*  h */  'h',    /*  i */  'i',    /*  j */  'j',    /*  k */  'k',
      /*  l */  'l',    /*  m */  'm',    /*  n */  'n',    /*  o */  'o',
      /*  p */  'p',    /*  q */  'q',    /*  r */  'r',    /*  s */  's',
      /*  t */  't',    /*  u */  'u',    /*  v */  'v',    /*  w */  'w',
      /*  x */  'x',    /*  y */  'y',    /*  z */  'z',    /*  { */  '{',
      /*  | */  '|',    /*  } */  '}',    /*  ~ */  '~',    /* ^? */    0, };

#define DEAD_STATE                    -1   /* used to block a DFA */
#define TABLE_INCREMENT              256   /* used to grow tables */


       /*************************   Hashing   *************************/
       /* Sets of suffixes labeling states during the DFA construction */
       /* are hashed to speed searching.  The hashing function uses an */
       /* entire integer variable range as its hash table size;  in an */
       /* effort to get a good spread through this range, hash values  */
       /* start big, and are incremented by a lot with every new word  */
       /* in the list.  The collision rate is low using this method    */

#define HASH_START               5775863
#define HASH_INCREMENT          38873647


       /**************   State Label Binary Search Tree   **************/
       /* During DFA construction, all states must be searched by      */
       /* their labels to make sure that the minimum number of states  */
       /* are used.  This operation is sped up by hashing the labels   */
       /* to a signature value, then storing the signatures and labels */
       /* in a binary search tree.  The tree is destroyed once the DFA */
       /* is fully constructed.                                        */

typedef struct TreeNode {
       StrList label;            /* state label used as search key     */
       unsigned signature;       /* hashed label to speed searching    */
       int state;                /* whose label is representd by node  */
       struct TreeNode *left;    /* left binary search subtree         */
       struct TreeNode *right;   /* right binary search subtree        */
       } SearchTreeNode, *SearchTree;


       /*********************   DFA State Table   **********************/
       /* The state table is an array of structures holding a state    */
       /* label, a count of the arcs out of the state, a pointer into  */
       /* the arc table for these arcs, and a final state flag.  The   */
       /* label field is used only during machine construction.        */

typedef struct {
       StrList label;            /* for this state - used during build */
       int num_arcs;             /* for this state in the arc table    */
       int arc_offset;           /* for finding arcs in the arc table  */
       short is_final;           /* TRUE iff this is a final state     */
       } StateTableEntry, *StateTable;
       /**********************   DFA Arc Table   **********************/
       /* The arc table lists all transitions for all states in a DFA  */
       /* in compacted form.  Each state's transitions are offset from */
       /* the start of the table, then listed in arc label order.      */
       /* Transitions are found by a linear search of the sub-section  */
       /* of the table for a given state.                              */

typedef struct {
       char label;              /* character label on an out-arrow    */
       int target;              /* the target state for the out-arrow */
       } ArcTableEntry, *ArcTable;


       /**********************   DFA Structure   **********************/
       /* A DFA is represented as a pointer to a structure holding the */
       /* machine's state and transition tables, and bookkeepping      */
       /* counters.  The tables are arrays whose space is malloc'd,    */
       /* then realloc'd if more space is required.  Once a machine is */
       /* constructed, the table space is realloc'd one last time to   */
       /* fit the needs of the machine exactly.                        */

typedef struct _DfaStruct {
       int num_states;          /* in the DFA (and state table)       */
       int max_states;          /* now allocated in the state table   */
       int num_arcs;            /* in the arc table for this machine  */
       int max_arcs;            /* now allocated in the arc table     */
       StateTable state_table;  /* the compacted DFA state table      */
       ArcTable arc_table;      /* the compacted DFA transition table */
       SearchTree tree;         /* storing state labels used in build */
       } DFAStruct;

/******************************************************************************/
/************************   Function Declarations   ***************************/

#ifdef __STDC__

static char *GetMemory( char *ptr, int num_bytes );
static void  DestroyTree( SearchTree tree );
static int   GetState( DFA machine, StrList label, unsigned signature );
static void  AddArc( DFA machine, int state, char arc_label,
                     StrList state_label, unsigned state_signature );

extern DFA   BuildDFA( StrList words );
extern char *GetTerm( FILE *stream, DFA machine, int size, char *output );

#else

static char *GetMemory();
static void  DestroyTree();
static int   GetState();
static void  AddArc();

extern DFA   BuildDFA();
extern char *GetTerm();

#endif

/******************************************************************************/
/***********************   Private Function Definitions   *********************/

/*FN***************************************************************************

        GetMemory( ptr, num_bytes )

   Returns: char * -- new/expanded block of memory

   Purpose: Rationalize memory allocation and handle errors

   Plan:    Part 1: Allocate memory with supplied allocation functions
            Part 2: Handle any errors
            Part 3: Return the allocated block of memory

   Notes:   None.
**/

static char *
GetMemory( ptr, num_bytes )
   char *ptr;      /* in: expanded block; NULL if nonesuch */
   int num_bytes;  /* in: number of bytes to allocate */
   {
   char *memory;   /* temporary for holding results */

        /* Part 1: Allocate memory with supplied allocation functions */
   if ( NULL == ptr )
      memory = malloc( (unsigned)num_bytes );
   else
      memory = realloc( ptr, (unsigned)num_bytes );
```

```
                    /* Part 2: Handle any errors */
   if ( NULL == memory )
      {
      (void)fprintf( stderr, "malloc failure--aborting\n" );
      exit(1);
      }

             /* Part 3: Return the allocated block of memory */
   return( memory );

   } /* GetMemory */

/*FN*************************************************************************

        DestroyTree( tree )

   Returns: void

   Purpose: Destroy a binary search tree created during machine construction

   Plan:    Part 1: Return right away of there is no tree
            Part 2: Deallocate the subtrees
            Part 3: Deallocate the root

   Notes:   None.
**/

static void
DestroyTree( tree )
   SearchTree tree;   /* in: search tree destroyed */
   {
              /* Part 1: Return right away of there is no tree */
   if ( NULL == tree ) return;

                     /* Part 2: Deallocate the subtrees */
   if ( NULL != tree->left )  DestroyTree( tree->left );
   if ( NULL != tree->right ) DestroyTree( tree->right );

                      /* Part 3: Deallocate the root */
   tree->left = tree->right = NULL;
   (void)free( (char *)tree );

   } /* DestroyTree */

/*FN*************************************************************************

        GetState( machine, label, signature )

   Returns: int -- state with the given label

   Purpose: Search a machine and return the state with a given state label

   Plan:    Part 1: Search the tree for the requested state
            Part 2: If not found, add the label to the tree
            Part 3: Return the state number

   Notes:   This machine always returns a state with the given label
            because if the machine does not have a state with the given
            label, then one is created.
**/

static int
GetState( machine, label, signature )
   DFA machine;        /* in: DFA whose state labels are searched;*/
   StrList label;      /* in: state label searched for */
   unsigned signature; /* in: signature of the label requested */
   {
   SearchTree *ptr;       /* pointer to a search tree link field */
   SearchTree new_node;   /* for a newly added search tree node */

             /* Part 1: Search the tree for the requested state */
   ptr = &(machine->tree);
   while ( (NULL != *ptr) && (   (signature != (*ptr)->signature)
                              || !StrListEqual(label,(*ptr)->label)) )
      ptr = (signature <= (*ptr)->signature) ? &(*ptr)->left : &(*ptr)->right;

             /* Part 2: If not found, add the label to the tree */
   if ( NULL == *ptr )
      {
         /* create a new node and fill in its fields */
      new_node = (SearchTree)GetMemory( NULL, sizeof(SearchTreeNode) );
      new_node->signature = signature;
      new_node->label = (StrList)label;
      new_node->state = machine->num_states;
      new_node->left = new_node->right = NULL;
```

```
         /* allocate more states if needed, set up the new state */
      if ( machine->num_states == machine->max_states )
         {
         machine->max_states += TABLE_INCREMENT;
         machine->state_table =
            (StateTable)GetMemory( (char*)(machine->state_table), machine-
>max_states*sizeof(StateTableEntry));
          }
      machine->state_table[machine->num_states].label = (StrList)label;
      machine->num_states++;

         /* hook the new node into the binary search tree */
      *ptr = new_node;
      }
   else
      StrListDestroy( label );

                   /* Part 3: Return the state number */
   return( (*ptr)->state );

   } /* GetState */

/*FN***************************************************************************

        AddArc( machine, state, arc_label, state_label, state_signature )

   Returns: void

   Purpose: Add an arc between two states in a DFA

   Plan:    Part 1: Search for the target state among existing states
            Part 2: Make sure the arc table is big enough
            Part 3: Add the new arc

   Notes:   None.
**/

static void
AddArc( DFA machine, int state, char arc_label,
                     StrList state_label, unsigned state_signature )
//   DFA machine;              /* in/out: machine with an arc added */
//   int state;                /* in: with an out arc added */
//   char arc_label;           /* in: label on the new arc */
//   StrList state_label;      /* in: label on the target state */
//   unsigned state_signature; /* in: label hash signature to speed searching */
   {
   register int target;   /* destination state for the new arc */

        /* Part 1: Search for the target state among existing states */
   StrListSort( state_label );
   target = GetState( machine, state_label, state_signature );

               /* Part 2: Make sure the arc table is big enough */
   if ( machine->num_arcs == machine->max_arcs )
      {
      machine->max_arcs += TABLE_INCREMENT;

        machine->arc_table =
        machine->arc_table = (ArcTable) GetMemory( (char*) machine->arc_table,
         machine->max_arcs * sizeof(ArcTableEntry) );
      }

                        /* Part 3: Add the new arc */
   machine->arc_table[machine->num_arcs].label = arc_label;
   machine->arc_table[machine->num_arcs].target = target;
   machine->num_arcs++;
   machine->state_table[state].num_arcs++;

   } /* AddArc */

/*FN***************************************************************************

        BuildDFA( words )

   Returns: DFA -- newly created finite state machine

   Purpose: Build a DFA to recognize a list of words

   Plan:    Part 1: Allocate space and initialize variables
            Part 2: Make and label the DFA start state
            Part 3: Main loop - build the state and arc tables
            Part 4: Deallocate the binary search tree and the state labels
            Part 5: Reallocate the tables to squish them down
            Part 6: Return the newly constructed DFA

   Notes:   None.
**/
```

```
DFA
BuildDFA( words )
   StrList words;  /* in: that the machine is built to recognize */
   {
   DFA machine;               /* local for easier access to machine */
   register int state;        /* current state's state number */
   char arc_label;            /* for the current arc when adding arcs */
   char *string;              /* element in a set of state labels */
   char ch;                   /* the first character in a new string */
   StrList current_label;     /* set of strings labeling a state */
   StrList target_label;      /* labeling the arc target state */
   unsigned target_signature; /* hashed label for binary search tree */
   register int i;            /* for looping through strings */

              /* Part 1: Allocate space and initialize variables */
   machine = (DFA)GetMemory( NULL, sizeof(DFAStruct) );

   machine->max_states = TABLE_INCREMENT;
   machine->state_table =
      (StateTable)GetMemory(NULL, machine->max_states*sizeof(StateTableEntry));
   machine->num_states = 0;

   machine->max_arcs = TABLE_INCREMENT;
   machine->arc_table =
      (ArcTable)GetMemory( NULL, machine->max_arcs * sizeof(ArcTableEntry) );
   machine->num_arcs = 0;

   machine->tree = NULL;

                /* Part 2: Make and label the DFA start state */
   StrListUnique( words );               /* sort and unique the list */
   machine->state_table[0].label = words;
   machine->num_states = 1;

            /* Part 3: Main loop - build the state and arc tables */
   for ( state = 0; state < machine->num_states; state++ )
      {
              /* The current state has nothing but a label, so */
              /* the first order of business is to set up some */
              /* of its other major fields                     */
      machine->state_table[state].is_final = FALSE;
      machine->state_table[state].arc_offset = machine->num_arcs;
      machine->state_table[state].num_arcs = 0;

             /* Add arcs to the arc table for the current state */
             /* based on the state's derived set.  Also set the */
             /* state's final flag if the empty string is found */
             /* in the suffix list                              */
      current_label = machine->state_table[state].label;
      target_label = StrListCreate();
      target_signature = HASH_START;
      arc_label = EOS;
      for ( i = 0; i < StrListSize(current_label); i++ )
         {
            /* get the next string in the label and lop it */
         string = StrListPeek( current_label, i );
         ch = *string++;

            /* the empty string means mark this state as final */
         if ( EOS == ch )
            { machine->state_table[state].is_final = TRUE; continue; }

            /* make sure we have a legitimate arc_label */
         if ( EOS == arc_label ) arc_label = ch;

            /* if the first character is new, then we must */
            /* add an arc for the previous first character */
         if ( ch != arc_label )
            {
            AddArc(machine, state, arc_label, target_label, target_signature);
            target_label = StrListCreate();
            target_signature = HASH_START;
            arc_label = ch;
            }

            /* add the current suffix to the target state label */
         StrListAppend( target_label, string );
         target_signature += (*string + 1) * HASH_INCREMENT;
         while ( *string ) target_signature += *string++;
         }

              /* On loop exit we have not added an arc for the */
              /* last bunch of suffixes, so we must do so, as  */
              /* long as the last set of suffixes is not empty */
              /* (which happens when the current state label   */
              /* is the singleton set of the empty string).    */
      if ( 0 < StrListSize(target_label) )
         AddArc( machine, state, arc_label, target_label, target_signature );
      }

      /* Part 4: Deallocate the binary search tree and the state labels */
   DestroyTree( machine->tree );  machine->tree = NULL;
   for ( i = 0; i < machine->num_states; i++ )
      {
      StrListDestroy( machine->state_table[i].label );
      machine->state_table[i].label = NULL;
      }

            /* Part 5: Reallocate the tables to squish them down */
   machine->state_table = (StateTable)GetMemory( (char *) machine->state_table,
                               machine->num_states * sizeof(StateTableEntry) );
   machine->arc_table = (ArcTable)GetMemory( (char *) machine->arc_table,
                               (machine->num_arcs * sizeof(ArcTableEntry)) );

                /* Part 6: Return the newly constructed DFA */
   return( machine );

   } /* BuildDFA */

/*FN***********************************************************************

        GetTerm( stream, machine, size, output )

   Returns: char * -- NULL if stream is exhausted, otherwise output buffer

   Purpose: Get the next token from an input stream, filtering stop words

   Plan:    Part 1: Return NULL immediately if there is no input
            Part 2: Initialize the local variables
            Part 3: Main Loop: Put an unfiltered word into the output buffer
            Part 4: Return the output buffer

   Notes:   This routine runs the DFA provided as the machine parameter,
            and collects the text of any term in the output buffer.  If
            a stop word is recognized in this process, it is skipped.
            Care is also taken to be sure not to overrun the output buffer.
**/

char *
GetTerm( stream, machine, size, output )
   FILE *stream;    /* in: source of input characters */
   DFA machine;     /* in: finite state machine driving process */
   int size;        /* in: bytes in the output buffer */
   char *output;    /* in/out: where the next token in placed */
   {
   char *outptr;        /* for scanning through the output buffer */
   int ch;              /* current character during input scan */
   register int state;  /* current state during DFA execution */

           /* Part 1: Return NULL immediately if there is no input */
   if ( EOF == (ch = getc(stream)) ) return( NULL );

                  /* Part 2: Initialize the local variables */
   outptr = output;

     /* Part 3: Main Loop: Put an unfiltered word into the output buffer */
   do
      {
         /* scan past any leading delimiters */
      while ( (EOF != ch ) &&
              ((DELIM_CH == char_class[ch]) ||
               (DIGIT_CH == char_class[ch])) ) ch = getc( stream );

         /* start the machine in its start state */
      state = 0;

         /* copy input to output until reaching a delimiter, and also */
         /* run the DFA on the input to watch for filtered words      */
      while ( (EOF != ch) && (DELIM_CH != char_class[ch]) )
         {
         if ( outptr == (output+size-1) ) { outptr = output; state = 0; }
         *outptr++ = convert_case[ch];

         if ( DEAD_STATE != state )
            {
            register int i;   /* for scanning through arc labels */
            int arc_start;    /* where the arc label list starts */
            int arc_end;      /* where the arc label list ends */

            arc_start = machine->state_table[state].arc_offset;
            arc_end = arc_start + machine->state_table[state].num_arcs;

            for ( i = arc_start; i < arc_end; i++ )
               if ( convert_case[ch] == machine->arc_table[i].label )
                  { state = machine->arc_table[i].target; break; }

            if ( i == arc_end ) state = DEAD_STATE;
            }

         ch = getc( stream );
         }

         /* start from scratch if a stop word is recognized */
      if ( (DEAD_STATE != state) && machine->state_table[state].is_final )
         outptr = output;

         /* terminate the output buffer */
      *outptr = EOS;
      }
   while ( (EOF != ch) && !*output );

                    /* Part 4: Return the output buffer */
   return( output );

   } /* GetTerm */
```

```
/*******************************   strlist.c   *********************************

    Purpose: String list abstract data type implementation.

    Notes:   This module implements a straightforward string ordered list
             abstract data type.  It is optimized for appending and deleting
             from the end of the list.  Since they are ordered lists, string
             lists may be sorted, and their members are addressed by ordinal
             position (starting from 0).
**/

#include <stdio.h>
#include <malloc.h>
#include <string.h>

#include "strlist.h"

/******************************************************************************/
/******************   Private Defines and Data Structures   *******************/

#define FALSE                          0
#define TRUE                           1
#define EOS                          '\0'

#define INCREMENT                     32    /* increase size by this much */
#define MAX_LINE                     128    /* when reading text files */

typedef struct _StrListStruct {

               short size;             /* current length of the list */
               short max_size;         /* room for this many strings */
               char **string;          /* the string array */

               } StrListStruct;

            /*********   GetMemory and FreeMemory Macros   ********/

#define GetMemory(b,s)                ( (b) ? realloc(b,s) : malloc(s) )
#define FreeMemory(b)                 ( (void)free( b ) )

/******************************************************************************/
/**********************   Private Routine Declarations   **********************/

#ifdef __STDC__

static int  ExpandArray( StrList list );
static void ISort( char **string, int lb, int ub );
static void QSort( char **string, int lb, int ub );

#else

static int  ExpandArray( /* list */ );
static void ISort( /* string, lb, ub */ );
static void QSort( /* string, lb, ub */ );

#endif

/*FN**************************************************************************

         ExpandArray( list )

   Returns: int -- TRUE (1) on success, FALSE (0) otherwise

   Purpose: Increase the string array to hold more data

   Plan:    Part 1: Increase the maximum list size to its new value
            Part 2: Allocate a new chunk of memory
            Part 3: Return an indication of success

   Notes:   None
**/

static int
ExpandArray( list )
   StrList list;   /* in: string list whose string array is enlarged */
   {

         /* Part 1: Increase the maximum list size to its new value */
   list->max_size += INCREMENT;

                  /* Part 2: Allocate a new chunk of memory */
   list->string = (char **)GetMemory( (char *)list->string,
                                      (list->max_size*sizeof(char *)) );

                 /* Part 3: Return an indication of success */
   return( (list->string) ? TRUE : FALSE );
   } /* ExpandArray */

/*FN**************************************************************************

         ISort( string, lb, ub )

   Returns: void

   Purpose: Insertion sort a string array forward using strcmp ordering

   Plan:    Part 1: Put smallest in place as a sentinal
            Part 2: Insert as necessary

   Notes:   None
**/

static void
ISort( string, lb, ub )
   char **string;   /* in/out: string array sorted */
   int lb,ub;       /* in: array bounds for sort */
   {
   register int i,j;  /* for scanning through the list */
   char *tmp;         /* for swaps */

               /* Part 1: Put smallest in place as a sentinal */
   for ( j = lb, i = lb+1; i <= ub; i++ )
      if ( 0 < strcmp(string[j],string[i]) ) j = i;
   tmp = string[lb]; string[lb] = string[j]; string[j] = tmp;

                       /* Part 2: Insert as necessary */
   for ( i = lb+2; i <= ub; i++ )
      {
      tmp = string[i];
      for ( j = i; 0 < strcmp(string[j-1],tmp); j-- ) string[j] = string[j-1];
      string[j] = tmp;
      }

   } /* ISort*/

/*FN**************************************************************************

         QSort( string, lb, ub )

   Returns: void

   Purpose: Quicksort an array of strings forward using strcmp ordering

   Plan:    Part 1: Use insertion sort of the list is short
            Part 2: Do median of three pivot value selection
            Part 3: Put the pivot out of the way at the top
            Part 5: Swap the pivot back into the mid of the list
            Part 6: Recursively sort the sublists

   Notes:   Standard quicksort function with the two main enhancements:
            median of three partitioning to find a good pivot value,
            and sorting small arrays with insertion sort.
**/

static void
QSort(char **string, int lb, int ub )
   //char **string;  /* in/out: string array sorted */
   //int lb,ub;      /* in: array bounds for sort */
   {
   register int lft;  /* list pointer that closes from the left */
   register int rgt;  /* list pointer that closes from the right */
   register int mid;  /* index of the median of three value */
   char *tmp;         /* for string pointer swaps */
   char *pivot;       /* the pivot value string */

              /* Part 1: Use insertion sort of the list is short */
   if ( ub-lb < 12 ) { ISort( string, lb, ub ); return; }

             /* Part 2: Do median of three pivot value selection */
   mid = (lb+ub)/2;
   if ( strcmp(string[mid],string[lb]) < 0 )
      { tmp = string[mid]; string[mid] = string[lb]; string[lb] = tmp; }
   if ( strcmp(string[ub],string[mid]) < 0 )
      { tmp = string[mid]; string[mid] = string[ub]; string[ub] = tmp; }
   if ( strcmp(string[mid],string[lb]) < 0 )
      { tmp = string[mid]; string[mid] = string[lb]; string[lb] = tmp; }

             /* Part 3: Put the pivot out of the way at the top */
   tmp = string[mid]; string[mid] = string[ub-1]; string[ub-1] = tmp;

                 /* Part 4: Partition around the pivot value */
   lft = lb;
   rgt = ub-1;
   pivot = string[ub-1];
   do
      {
      do lft++; while ( strcmp(string[lft],pivot) < 0 );
      do rgt--; while ( strcmp(pivot,string[rgt]) < 0 );
      tmp = string[lft]; string[lft] = string[rgt]; string[rgt] = tmp;
      }
   while ( lft < rgt );

         /* Part 5: Swap the pivot back into the mid of the list */
   string[rgt] = string[lft]; string[lft] = string[ub-1]; string[ub-1] = tmp;

                  /* Part 6: Recursively sort the sublists */
   QSort( string, lb, lft-1 );
   QSort( string, rgt+1, ub );

   } /* QSort */

/******************************************************************************/
/**********************   Public Routine Declarations   ***********************/

/*FN****************************************************************************

         StrListAppend( list, string )

   Returns: void

   Purpose: Place a string on the end of a string list

   Plan:    Part 1: Standard parameter sanity check
            Part 2: Expand the list as necessary
            Part 3: Append the new string to the tail

   Notes:   None
**/

void
StrListAppend( list, string )
   StrList list;   /* in/out: list appended to */
   char *string;   /* in: the appended string */
   {
   int length;  /* of the added string and its terminator */

                 /* Part 1: Standard parameter sanity check */
   if ( !list || !string ) return;

                   /* Part 2: Expand the list as necessary */
   if ( (list->size == list->max_size) && !ExpandArray(list) ) return;

                 /* Part 3: Append the new string to the tail */
   length = strlen( string ) + 1;
   list->string[list->size] = GetMemory( NULL, length );
   (void)memcpy( list->string[list->size], string, length );
   list->size++;

   } /* StrListAppend */

/*FN****************************************************************************

         StrListAppendFile( list, filename )

   Returns: void

   Purpose: Place all lines from a file on the end of a string list

   Plan:    Part 1: Standard parameter sanity check
            Part 2: Expand the list as necessary
            Part 3: Append the new string to the tail

   Notes:   None
**/

void
StrListAppendFile( list, filename )
   StrList list;     /* in/out: list appended to */
   char *filename;   /* in: the appended file */
   {
   FILE *file;            /* file handle for the text input file */
   char buffer[MAX_LINE]; /* for storing text input file lines */
   int length;            /* of the added string and its terminator */
   register int i;        /* for looping through the TextBlock lines */

                 /* Part 1: Standard parameter sanity check */
   if ( !list || !filename ) return;

             /* Part 2: Open the text input file; check for error */
   if ( NULL == (file = fopen(filename,"r")) ) return;
```

```
              /* Part 3: Append to the list, checking for errors */
   while ( NULL != fgets(buffer,MAX_LINE,file) )
      {
      if ( (list->size == list->max_size) && !ExpandArray(list) ) return;

      i = list->size;
      length = strlen( buffer );
      list->string[i] = GetMemory( NULL, (unsigned)length );
      if ( NULL == list->string[i] ) return;
      (void)memcpy( list->string[i], buffer, length );
      list->string[i][length-1] = EOS;
      list->size++;
      }

                    /* Part 4: Close the text input file */
   (void)fclose( file );

   } /* StrListAppendFile */

/*FN***************************************************************************

         StrListCreate()

   Returns: StrList -- a new structure, or NULL on failure

   Purpose: Allocate and initialize a new string list structure

   Plan:    Part 1: Allocate space for the string list object
            Part 2: Initialize the structure fields
            Part 3: Return the new string list

   Notes:   None
**/

StrList
StrListCreate()
   {
   StrList list;  /* the new list returned */

            /* Part 1: Allocate space for the string list object */
   if ( !(list = (StrList)GetMemory(NULL,sizeof(StrListStruct))) )
      return( NULL );

                 /* Part 2: Initialize the structure fields */
   list->string = NULL;
   list->size = list->max_size = 0;
   if ( !ExpandArray(list) )
      { FreeMemory( (char *)list ); return( NULL ); }

                    /* Part 3: Return the new string list */
   return( list );

   } /* StrListCreate */

/*FN***************************************************************************

         StrListDestroy( list )

   Returns: void

   Purpose: Deallocate the space used for a string list

   Plan:    Part 1: Standard parameter sanity check
            Part 2: Free all the space

   Notes:   None
**/

void
StrListDestroy( list )
   StrList list;    /* in: the list destroyed */
   {
   register int i;  /* for scanning through the list */

                 /* Part 1: Standard parameter sanity check */
   if ( !list ) return;

                        /* Part 2: Free all the space */
   for ( i = 0; i < list->size; i++ ) FreeMemory( (char *)(list->string[i]) );
   FreeMemory( (char *)list );

   } /* StrListDestroy */



/*FN***************************************************************************
```

```
         StrListElementEqual( list1, list2 )

   Returns: int -- TRUE if the term in position "pos" from list1 is found in list2

   Purpose: See if the term in position "pos" from list1 also exists in list2

   Plan:    Part 1: Say not equal if the parameters are bad
            Part 2: Compare the element in the first list with every element in list2
            Part 3: Say false if nothing is found

   Notes:   None
**/

int
StrListElementEqual( pos, list1, list2 )
   StrList list1,list2;   /* in: lists compared */
   int pos;
   {
   register int i;  /* for scanning through the anti-stoplist */
   int success;

             /* Part 1: Say not equal if the parameters are bad */
   if ( !list1 || !list2 ) return( FALSE );

              /* Part 3: Compare the lists element by element */
   success = 0;
   for ( i = 0; i < list2->size; i++ ){
      if ( strcmp(list1->string[pos],list2->string[i]) ==0 )  // compares two strings
         success = 1;
   }
   if (success) return( TRUE );
   else return (FALSE);

   } /* StrListElementEqual */


/*FN***********************************************************************

         StrListCount( pos, list1, list2, start2 )

**/

int
StrListCount( pos, list1, list2, start2 )
   StrList list1,list2;   /* in: lists compared */
   int pos, start2;
   {
   register int i;  /* for scanning through the anti-stoplist */
   int antall;

             /* Part 1: Say not equal if the parameters are bad */
   if ( !list1 || !list2 ) return( FALSE );

              /* Part 3: Compare the lists element by element */
   antall = 0;
   for ( i = 0; i < (list2->size); i++ ){
//   while ( i < list2->size){
      if ( strcmp(list1->string[pos],list2->string[i]) ==0 )  // compares two strings
         antall++;
   }
   return antall;

   } /* StrListCount */

/*FN***********************************************************************

         StrListEqual( list1, list2 )

   Returns: int -- TRUE if the lists are equivalent, FALSE otherwise

   Purpose: See if two lists have identical elements

   Plan:    Part 1: Say not equal if the parameters are bad
            Part 2: Say not equal if sizes are different
            Part 3: Compare lists element by element
            Part 4: Say equal if everything checks out

   Notes:   None
**/

int
StrListEqual( list1, list2 )
   StrList list1,list2;   /* in: lists compared */
   {
   register int i;  /* for scanning through the lists */
```

```
             /* Part 1: Say not equal if the parameters are bad */
   if ( !list1 || !list2 ) return( FALSE );

               /* Part 2: Say not equal if sizes are different */
   if ( list1->size != list2->size ) return( FALSE );

              /* Part 3: Compare the lists element by element */
   for ( i = 0; i < list1->size; i++ )
      if ( *(list1->string[i]) != *(list2->string[i]) )
         return( FALSE );
      else if ( 0 != strcmp(list1->string[i],list2->string[i]) )
         return( FALSE );

               /* Part 5: Say equal if everything checks out */
   return( TRUE );

   } /* StrListEqual */


/*FN***********************************************************************
SAH july 2002: Check if term is included in list. Return false otherwise
**/

int
StrListMember(char term[], StrList list){
register int i;

  for ( i = 0; i < list->size; i++ ) {
    if ( 0 == strcmp(term,list->string[i]) )  //strcmp returns zero if strings are identical
       return TRUE;  //found
  }
  return FALSE; //if word not found


} //StrListMember

/*FN***********************************************************************

         StrListPeek( list, index )

   Returns: char * -- pointer to the requested string; NULL on error

   Purpose: Peek a string by its list index. (Get an element from the list)

   Plan:    Part 1: Standard parameter sanity check
            Part 2: Return the requested string

   Notes:   Note that this function is a hole in the data type encapsulation:
            it should return a copy, but this would force the consumer to
            deallocate the string.  Design call.
**/

char *
StrListPeek( list, index )
   StrList list;   /* in: list retrieved from */
   int index;      /* in: which string to fetch */
   {
                 /* Part 1: Standard parameter sanity check */
   if ( !list || (index < 0) || (list->size <= index) ) return( NULL );

                   /* Part 2: Return the requested string */
   return( list->string[index] );

   } /* StrListPeek */

/*FN***********************************************************************

         StrListSize( list )

   Returns: int -- the size of the list, 0 on error

   Purpose: Grab the list size

   Plan:    Return the list size field

   Notes:   None
**/

int
StrListSize( list )
   StrList list;   /* in: list queried */
   {

   if ( !list ) return( 0 ); else return( list->size );

   } /* StrListSize */
```

```
/*FN***************************************************************************

         StrListSort( list )

   Returns: void

   Purpose: Sort a single string list using strcmp ordering

   Plan:    Part 1: Do parameter sanity checks, then sort

   Notes:   None
**/

void
StrListSort( list )
   StrList list;   /* in/out: list sorted */
   {
             /* Part 1: Do parameter sanity checks, then sort */
   if ( !list ) return;
   QSort( list->string, 0, list->size-1 );

   } /* StrListSort */

/*FN***************************************************************************

         StrListUnique( list )

   Returns: void

   Purpose: Sort a single string list using strcmp ordering, then remove
            duplicates.

   Plan:    Part 1: Do parameters sanity checks
            Part 2: Sort the list
            Part 3: Remove duplicate strings

   Notes:   None
**/

void
StrListUnique( list )
   StrList list;   /* in/out: list sorted and uniqued */
   {
   register i,j;   /* counters for copying down over duplicates */

                    /* Part 1: Do parameter sanity checks */
   if ( !list ) return;

                       /* Part 2: Sort the list */
   QSort( list->string, 0, list->size-1 );

                   /* Part 3: Remove duplicate strings */
   if ( 1 < list->size )
      {
      for ( j = 0, i = 1; i < list->size; i++ )
         {
         if ( 0 == strcmp(list->string[i],list->string[j]) )
            (void)free( list->string[j] );
         else
            j++;
         if ( j < i ) list->string[j] = list->string[i];
         }
      list->size = j + 1;
      }

   } /* StrListUnique */
```

```
/*******************************   stem.c   **********************************

   Purpose:    Implementation of the Porter stemming algorithm documented
               in: Porter, M.F., "An Algorithm For Suffix Stripping,"
               Program 14 (3), July 1980, pp. 130-137.

   Provenance: Written by B. Frakes and C. Cox, 1986.
               Changed by C. Fox, 1990.
                  - made measure function a DFA
                  - restructured structs
                  - renamed functions and variables
                  - restricted function and variable scopes
               Changed by C. Fox, July, 1991.
                  - added ANSI C declarations
                  - branch tested to 90% coverage

   Notes:      This code will make little sense without the the Porter
               article.  The stemming function converts its input to
               lower case.
**/

/************************   Standard Include Files   *************************/

#include <stdio.h>
#include <string.h>
#include <ctype.h>

/*****************************************************************************/
/*****************   Private Defines and Data Structures   *******************/

#define FALSE                         0
#define TRUE                          1
#define EOS                         '\0'

#define IsVowel(c)        ('a'==(c)||'e'==(c)||'i'==(c)||'o'==(c)||'u'==(c))

typedef struct {
           int id;                 /* returned if rule fired */
           char *old_end;          /* suffix replaced */
           char *new_end;          /* suffix replacement */
           int old_offset;         /* from end of word to start of suffix */
           int new_offset;         /* from beginning to end of new suffix */
           int min_root_size;      /* min root word size for replacement */
           int (*condition)();     /* the replacement test function */
           } RuleList;

static char LAMBDA[1] = "";        /* the constant empty string */
static char *end;                  /* pointer to the end of the word */

/*****************************************************************************/
/********************   Private Function Declarations   **********************/

#ifdef __STDC__

static int WordSize( char *word );
static int ContainsVowel( char *word );
static int EndsWithCVC( char *word );
static int AddAnE( char *word );
static int RemoveAnE( char *word );
static int ReplaceEnd( char *word, RuleList *rule );
//her var det ikke samsvar mellom prototype og funksjonens typer.
//Stod slik før (har nå modifisert til at pekeren er med) :
//static int ReplaceEnd( char *word, RuleList rule );  //rule skal være *rule

#else

static int WordSize( /* word */ );
static int ContainsVowel( /* word */ );
static int EndsWithCVC( /* word */ );
static int AddAnE( /* word */ );
static int RemoveAnE( /* word */ );
static int ReplaceEnd( /* word, rule */ );

#endif

/******************************************************************************/
/*****************   Initialized Private Data Structures   ********************/

static RuleList step1a_rules[] =
           {
             101,  "sses",      "ss",    3,  1, -1,  NULL,
             102,  "ies",       "i",     2,  0, -1,  NULL,
             103,  "ss",        "ss",    1,  1, -1,  NULL,
             104,  "s",         LAMBDA,  0, -1, -1,  NULL,
             000,  NULL,        NULL,    0,  0,  0,  NULL,
           };
```

```
static RuleList step1b_rules[] =
           {
             105,  "eed",       "ee",     2,  1,  0,  NULL,
             106,  "ed",        LAMBDA,  1, -1, -1,  ContainsVowel,
             107,  "ing",       LAMBDA,  2, -1, -1,  ContainsVowel,
             000,  NULL,        NULL,    0,  0,  0,  NULL,
           };

static RuleList step1b1_rules[] =
           {
             108,  "at",        "ate",   1,  2, -1,  NULL,
             109,  "bl",        "ble",   1,  2, -1,  NULL,
             110,  "iz",        "ize",   1,  2, -1,  NULL,
             111,  "bb",        "b",     1,  0, -1,  NULL,
             112,  "dd",        "d",     1,  0, -1,  NULL,
             113,  "ff",        "f",     1,  0, -1,  NULL,
             114,  "gg",        "g",     1,  0, -1,  NULL,
             115,  "mm",        "m",     1,  0, -1,  NULL,
             116,  "nn",        "n",     1,  0, -1,  NULL,
             117,  "pp",        "p",     1,  0, -1,  NULL,
             118,  "rr",        "r",     1,  0, -1,  NULL,
             119,  "tt",        "t",     1,  0, -1,  NULL,
             120,  "ww",        "w",     1,  0, -1,  NULL,
             121,  "xx",        "x",     1,  0, -1,  NULL,
             122,  LAMBDA,      "e",    -1,  0, -1,  AddAnE,
             000,  NULL,        NULL,    0,  0,  0,  NULL,
             };

static RuleList step1c_rules[] =
           {
             123,  "y",         "i",      0,  0, -1,  ContainsVowel,
             000,  NULL,        NULL,    0,  0,  0,  NULL,
           };

static RuleList step2_rules[] =
           {
             203,  "ational",   "ate",   6,  2,  0,  NULL,
             204,  "tional",    "tion",  5,  3,  0,  NULL,
             205,  "enci",      "ence",  3,  3,  0,  NULL,
             206,  "anci",      "ance",  3,  3,  0,  NULL,
             207,  "izer",      "ize",   3,  2,  0,  NULL,
             208,  "abli",      "able",  3,  3,  0,  NULL,
             209,  "alli",      "al",    3,  1,  0,  NULL,
             210,  "entli",     "ent",   4,  2,  0,  NULL,
             211,  "eli",       "e",     2,  0,  0,  NULL,
             213,  "ousli",     "ous",   4,  2,  0,  NULL,
             214,  "ization",   "ize",   6,  2,  0,  NULL,
             215,  "ation",     "ate",   4,  2,  0,  NULL,
             216,  "ator",      "ate",   3,  2,  0,  NULL,
             217,  "alism",     "al",    4,  1,  0,  NULL,
             218,  "iveness",   "ive",   6,  2,  0,  NULL,
             219,  "fulnes",    "ful",   5,  2,  0,  NULL,
             220,  "ousness",   "ous",   6,  2,  0,  NULL,
             221,  "aliti",     "al",    4,  1,  0,  NULL,
             222,  "iviti",     "ive",   4,  2,  0,  NULL,
             223,  "biliti",    "ble",   5,  2,  0,  NULL,
             000,  NULL,        NULL,    0,  0,  0,  NULL,
           };

static RuleList step3_rules[] =
           {
             301,  "icate",     "ic",    4,  1,  0,  NULL,
             302,  "ative",     LAMBDA,  4, -1,  0,  NULL,
             303,  "alize",     "al",    4,  1,  0,  NULL,
             304,  "iciti",     "ic",    4,  1,  0,  NULL,
             305,  "ical",      "ic",    3,  1,  0,  NULL,
             308,  "ful",       LAMBDA,  2, -1,  0,  NULL,
             309,  "ness",      LAMBDA,  3, -1,  0,  NULL,
             000,  NULL,        NULL,    0,  0,  0,  NULL,
           };

static RuleList step4_rules[] =
           {
             401,  "al",        LAMBDA,  1, -1,  1,  NULL,
             402,  "ance",      LAMBDA,  3, -1,  1,  NULL,
             403,  "ence",      LAMBDA,  3, -1,  1,  NULL,
             405,  "er",        LAMBDA,  1, -1,  1,  NULL,
             406,  "ic",        LAMBDA,  1, -1,  1,  NULL,
             407,  "able",      LAMBDA,  3, -1,  1,  NULL,
             408,  "ible",      LAMBDA,  3, -1,  1,  NULL,
             409,  "ant",       LAMBDA,  2, -1,  1,  NULL,
             410,  "ement",     LAMBDA,  4, -1,  1,  NULL,
             411,  "ment",      LAMBDA,  3, -1,  1,  NULL,
             412,  "ent",       LAMBDA,  2, -1,  1,  NULL,
             423,  "sion",      "s",     3,  0,  1,  NULL,
             424,  "tion",      "t",     3,  0,  1,  NULL,
             415,  "ou",        LAMBDA,  1, -1,  1,  NULL,
             416,  "ism",       LAMBDA,  2, -1,  1,  NULL,
             417,  "ate",       LAMBDA,  2, -1,  1,  NULL,
             418,  "iti",       LAMBDA,  2, -1,  1,  NULL,
             419,  "ous",       LAMBDA,  2, -1,  1,  NULL,
             420,  "ive",       LAMBDA,  2, -1,  1,  NULL,
             421,  "ize",       LAMBDA,  2, -1,  1,  NULL,
             000,  NULL,        NULL,    0,  0,  0,  NULL,
           };

static RuleList step5a_rules[] =
           {
             501,  "e",         LAMBDA,  0, -1,  1,  NULL,
             502,  "e",         LAMBDA,  0, -1, -1,  RemoveAnE,
             000,  NULL,        NULL,    0,  0,  0,  NULL,
           };

static RuleList step5b_rules[] =
           {
             503,  "ll",        "l",     1,  0,  1,  NULL,
             000,  NULL,        NULL,    0,  0,  0,  NULL,
           };

/*****************************************************************************/
/********************   Private Function Declarations   **********************/

/*FN*************************************************************************

       WordSize( word )

   Returns: int -- a weird count of word size in adjusted syllables

   Purpose: Count syllables in a special way:  count the number
            vowel-consonant pairs in a word, disregarding initial
            consonants and final vowels.  The letter "y" counts as a
            consonant at the beginning of a word and when it has a vowel
            in front of it; otherwise (when it follows a consonant) it
            is treated as a vowel.  For example, the WordSize of "cat"
            is 1, of "any" is 1, of "amount" is 2, of "anything" is 3.

   Plan:    Run a DFA to compute the word size

   Notes:   The easiest and fastest way to compute this funny measure is
            with a finite state machine.  The initial state 0 checks
            the first letter.  If it is a vowel, then the machine changes
            to state 1, which is the "last letter was a vowel" state.
            If the first letter is a consonant or y, then it changes
            to state 2, the "last letter was a consonant state".  In
            state 1, a y is treated as a consonant (since it follows
            a vowel), but in state 2, y is treated as a vowel (since
            it follows a consonant.  The result counter is incremented
            on the transition from state 1 to state 2, since this
            transition only occurs after a vowel-consonant pair, which
            is what we are counting.
**/

static int
WordSize( word )
   char *word;   /* in: word having its WordSize taken */
   {
   register int result;   /* WordSize of the word */
   register int state;    /* current state in machine */

   result = 0;
   state = 0;

                 /* Run a DFA to compute the word size */
   while ( EOS != *word )
      {
      switch ( state )
         {
         case 0: state = (IsVowel(*word)) ? 1 : 2;
                 break;
         case 1: state = (IsVowel(*word)) ? 1 : 2;
                 if ( 2 == state ) result++;
                 break;
         case 2: state = (IsVowel(*word) || ('y' == *word)) ? 1 : 2;
                 break;
         }
      word++;
      }

   return( result );

   } /* WordSize */

/*FN************************************************************************
```

```
       ContainsVowel( word )

   Returns: int -- TRUE (1) if the word parameter contains a vowel,
            FALSE (0) otherwise.

   Purpose: Some of the rewrite rules apply only to a root containing
            a vowel, where a vowel is one of "aeiou" or y with a
            consonant in front of it.

   Plan:    Obviously, under the definition of a vowel, a word contains
            a vowel iff either its first letter is one of "aeiou", or
            any of its other letters are "aeiouy".  The plan is to
            test this condition.

   Notes:   None
**/

static int
ContainsVowel( word )
   char *word;   /* in: buffer with word checked */
   {

   if ( EOS == *word )
      return( FALSE );
   else
      return( IsVowel(*word) || (NULL != strpbrk(word+1,"aeiouy")) );


   } /* ContainsVowel */

/*FN**************************************************************************

       EndsWithCVC( word )

   Returns: int -- TRUE (1) if the current word ends with a
            consonant-vowel-consonant combination, and the second
            consonant is not w, x, or y, FALSE (0) otherwise.

   Purpose: Some of the rewrite rules apply only to a root with
            this characteristic.

   Plan:    Look at the last three characters.

   Notes:   None
**/

static int
EndsWithCVC( word )
   char *word;   /* in: buffer with the word checked */
   {
   int length;         /* for finding the last three characters */

   if ( (length = strlen(word)) < 2 )
      return( FALSE );
   else
      {
      end = word + length - 1;
      return(    (NULL == strchr("aeiouwxy",*end--))      /* consonant */
              && (NULL != strchr("aeiouy",  *end--))        /* vowel */
              && (NULL == strchr("aeiou",   *end  )) );   /* consonant */
      }

   } /* EndsWithCVC */

/*FN**************************************************************************

       AddAnE( word )

   Returns: int -- TRUE (1) if the current word meets special conditions
            for adding an e.

   Purpose: Rule 122 applies only to a root with this characteristic.

   Plan:    Check for size of 1 and a consonant-vowel-consonant ending.

   Notes:   None
**/

static int
AddAnE( word )
   char *word;
   {

   return( (1 == WordSize(word)) && EndsWithCVC(word) );

   } /* AddAnE */
```

```
/*FN************************************************************************

       RemoveAnE( word )

   Returns: int -- TRUE (1) if the current word meets special conditions
            for removing an e.

   Purpose: Rule 502 applies only to a root with this characteristic.

   Plan:    Check for size of 1 and no consonant-vowel-consonant ending.

   Notes:   None
**/

static int
RemoveAnE( word )
   char *word;
   {

   return( (1 == WordSize(word)) && !EndsWithCVC(word) );

   } /* RemoveAnE */

/*FN************************************************************************

       ReplaceEnd( word, rule )

   Returns: int -- the id for the rule fired, 0 is none is fired

   Purpose: Apply a set of rules to replace the suffix of a word

   Plan:    Loop through the rule set until a match meeting all conditions
            is found.  If a rule fires, return its id, otherwise return 0.
            Connditions on the length of the root are checked as part of this
            function's processing because this check is so often made.

   Notes:   This is the main routine driving the stemmer.  It goes through
            a set of suffix replacement rules looking for a match on the
            current suffix.  When it finds one, if the root of the word
            is long enough, and it meets whatever other conditions are
            required, then the suffix is replaced, and the function returns.
**/

static int
ReplaceEnd( word, rule )
   char *word;        /* in/out: buffer with the stemmed word */
   RuleList *rule;    /* in: data structure with replacement rules
     rule er en peker til en RuleList struct... */
   {
   register char *ending;   /* set to start of possible stemmed suffix */
   char tmp_ch;             /* save replaced character when testing */

   while ( 0 != rule->id )
      {
      ending = end - rule->old_offset;
      if ( word <= ending )
         if ( 0 == strcmp(ending,rule->old_end) )
            {
            tmp_ch = *ending;
            *ending = EOS;
            if ( rule->min_root_size < WordSize(word) )
               if ( !rule->condition || (*rule->condition)(word) )
                  {
                  (void)strcat( word, rule->new_end );
                  end = ending + rule->new_offset;
                  break;
                  }
            *ending = tmp_ch;
            }
      rule++;
      }

   return( rule->id );

   } /* ReplaceEnd */
```

```
/*****************************************************************************/
/********************   Public Function Declarations   ********************/

/*FN*************************************************************************

       Stem( word )

   Returns: int -- FALSE (0) if the word contains non-alphabetic characters
            and hence is not stemmed, TRUE (1) otherwise

   Purpose: Stem a word

   Plan:    Part 1: Check to ensure the word is all alphabetic
            Part 2: Run through the Porter algorithm
            Part 3: Return an indication of successful stemming

   Notes:   This function implements the Porter stemming algorithm, with
            a few additions here and there.  See:

               Porter, M.F., "An Algorithm For Suffix Stripping,"
               Program 14 (3), July 1980, pp. 130-137.

            Porter's algorithm is an ad hoc set of rewrite rules with
            various conditions on rule firing.  The terminology of
            "step 1a" and so on, is taken directly from Porter's
            article, which unfortunately gives almost no justification
            for the various steps.  Thus this function more or less
            faithfully refects the opaque presentation in the article.
            Changes from the article amount to a few additions to the
            rewrite rules;  these are marked in the RuleList data
            structures with comments.
**/

int
Stem( word )
   char *word;  /* in/out: the word stemmed */
   {
   int rule;    /* which rule is fired in replacing an end */

            /* Part 1: Check to ensure the word is all alphabetic */
   for ( end = word; *end != EOS; end++ )
      if ( !isalpha(*end) ) return( FALSE );
      else *end = tolower( *end );
   end--;

                /*  Part 2: Run through the Porter algorithm */
   (void)ReplaceEnd( word, step1a_rules );  //dette er vel pekeren til arrayet?
   rule = ReplaceEnd( word, step1b_rules );
   if ( (106 == rule) || (107 == rule) )
      (void)ReplaceEnd( word, step1b1_rules );
   (void)ReplaceEnd( word, step1c_rules );

   (void)ReplaceEnd( word, step2_rules );

   (void)ReplaceEnd( word, step3_rules );

   (void)ReplaceEnd( word, step4_rules );

   (void)ReplaceEnd( word, step5a_rules );
   (void)ReplaceEnd( word, step5b_rules );

   /* Prøver med dette å sende inn verdiene til pekerne. Det blir selvsagt feil...
      Riktig slik det var med å sende inn pekeren. Må rette opp i funksjonen ReplaceEnd

      Før stod det (void)ReplaceEnd( word, step5b_rules );
      Dette har jeg endret til  (void)ReplaceEnd( word, *step5b_rules );
      for alle reglene. Funksjonen forventer nemlig en peker inn, mens
      det som opprinnelig stod i koden var et vanlig arraynavn. */

           /* Part 3: Return an indication of successful stemming */
   return( TRUE );

   } /* Stem */
```