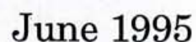
Approximate Content Match of Multimedia Data with Natural Language Queries

Kit-pui Wong

A Thesis Submitted to
The Graduate School of
The Chinese University of Hong Kong
in Partial Fulfillment of the Requirements
for the Degree of
Master of Philosophy in
Systems Engineering and Engineering Management



June 1995

P
98
W66
1995
ult

CONTENTS

ACKNOWLEDGMENT		4
ABSTRACT		6
KEYWORDS		7
Chapter 1	INTRODUCTION	9
Chapter 2	APPROACH	14
2.1	Challenges	15
2.2	Knowledge Representation	16
2.3	Proposed Information Model	17
2.4	Restricted Language Set	20
Chapter 3	THEORY	26
3.1	Features	26
3.1.1	Superficial Details	30
3.1.2	Hidden Details	31
3.2	Matching Process	36
3.2.1	Inexact Match	37
3.2.2	An Illustration	38
3.2.2.1	Stage 1 - Query Parsing	39
3.2.2.2	Stage 2 - Gross Filtering	41
3.2.2.3	Stage 3 - Fine Scoring	42
3.3	Extending Knowledge	46
3.3.1	Attributes with Intermediate Closeness	47
3.3.2	Comparing Different Entities	48
3.4	Putting Concepts to Work	50
Chapter 4	IMPLEMENTATION	52
4.1	Overall Structure	53

4.2	Choosing NL Parser	55
4.3	Ambiguity	56
4.4	Storing Knowledge	59
4.4.1	Type Hierarchy	60
4.4.1.1	Node Name	61
4.4.1.2	Node Identity	61
4.4.1.3	Operations	68
4.4.1.3.1	Direct Edit	68
4.4.1.3.2	Interactive Edit	68
4.4.2	Implicit Features	71
4.4.3	Database of Captions	72
4.4.4	Explicit Features	73
4.4.5	Transformation Map	74
Chapter 5	ILLUSTRATION	78
5.1	Gloss Tags	78
5.2	Parsing	81
5.2.1	Resolving Nouns and Verbs	81
5.2.2	Resolving Adjectives and Adverbs	84
5.2.3	Normalizing Features	89
5.2.4	Resolving Prepositions	90
5.3	Matching	93
5.3.1	Gross Filtering	94
5.3.2	Fine Scoring	96
Chapter 6	DISCUSSION	101
6.1	Performance Measures	101
6.1.1	General Parameters	101
6.1.2	Experiments	103
6.1.2.1	Inexact Matching Behaviour	103
6.1.2.2	Exact Matching Behaviour	106
6.2	Difficulties	108
6.3	Possible Improvement	110
6.4	Conclusion	112

REFERENCES 117

APPENDICES 121

Appendix A	Notation	121
Appendix B	Glossary	123
Appendix C	Proposed Feature Slots and Value	126
Appendix D	Sample Captions and Queries	128
Appendix E	Manual Pages	130
Appendix F	Directory Structure	136
Appendix G	Imported Toolboxes	137
Appendix H	Program Listing	140

ACKNOWLEDGMENT

I owe my immeasurable gratitude to my thesis supervisor, Prof. Vincent Lum. He has been paying his great patience to guide me all the way from the very beginning of the project. He gives me numerous hints in solving theoretical and technical problems. Besides, his vision and experience sharpen my view in the attitude and techniques of doing research. However, for all deficiencies that remained, it is myself who should solely be responsible.

In the project, I made contacts with many leading researchers in the field round the world, including Prof. H. Thompson, Dr. M.L. Mauldin, Dr. K. Dahlgren and Dr. M.T. Tengku. They provided me various degree of advice, and gave me some papers, theses and software tools.

My sincere gratitude goes to Prof. N.C. Rowe and Dr. E.J. Guglielmo of the Navel Postgraduate School, Monterey. They shared with me their experience in doing research in intelligent information retrieval. Their MARIE project was a good reference for me. Prof. Rowe even sent me one of his papers many months before it was published.

A special thanks should be conveyed to Mr. E.L. Antworth of the Summer Institute of Linguistics, Dallas. He gave me a lot of advice and suggestions on morphological parsing from his expertise in computational linguistics. The most important help I received was a copy of his parser PC-KIMMO, which made this project successful.

Many participants at the InfoScience'93 Conference in Korea had given me stimulating comments on the project. Those precious ideas were absorbed in the later refinement of the project. Further, I greatly appreciated the constructive comments from the anonymous referees of our paper [Lum93] as they gave useful ideas for improvements.

I would like to thank Dr. K. F. Wong. In the final stage of my thesis, he gave me many alternative views which I had overlooked.

In the Department of Systems Engineering and Engineering Management at the Chinese University of Hong Kong, the administrative and technical staff gave me various assistance over the years. They helped me very much, directly or indirectly, in the progress of the project.

Some friends of mine outside this field had offered their helpful hands. Candy Leung prepared the excellent hand-drawing of the "media data" examples in Figures 2.3, 3.7, and 3.8. These artworks are good examples and need much artistic talent for their creation. Joanna Cheng frequently raised her amazing ideas how a humanized, natural and friendly user interface and dialogues should appear. I am deeply grateful for their contributions.

Further, I am thankful to all the persons who have helped me in the course of the project, including those whose names I forget to put here.

Last but not least, my greatest debt is to my mother who has been providing me infinite love and care for all the years since my arrival to the world. In finishing the final stage of this thesis, I do not have much time to be with her, even on Mother's Day. Herewith I present my regret and the warmest thankfulness to her.

Kit-pui Wong
June 1995

ABSTRACT

Automatic machine analysis of the contents of multimedia data is not yet realistic. The retrieval of multimedia objects is therefore difficult. In this thesis, we suggest that data contents in a multi-media database system are best described by natural language captions. The captions are restricted to sentences of simple structure containing three semantic groups, namely the agent group, the action group and the patient group. Lower level details in ordinary natural language are less significant and they will be ignored in our assumptions. The restriction rules are discussed in the thesis and we shall show that such a restricted natural language is sufficient for retrieval purpose. The project is given the name ARMON, a partial acronym taken from "Approximate Retrieval of Multimedia Objects by Natural Language Captions".

The heads in the semantic groups play the major role in figuring out the meaning of the whole sentence. Each of the heads can be characterized by a large set of intrinsic properties, named as the implicit features of the group. Entities in the universe of discourse conceptually establish hierarchies with respect to their implicit features and are also depicted by a set called explicit features which are addressed in the context of the captions. Queries, issued by users and parsed into semantic groups in the same way, will be matched against the stored captions. During the matching process, attempts to match with the nearby nodes in the hierarchy are performed when exact match fails.

An algorithm will calculate the matching scores between corresponding groups in caption candidates and the query. An integrated score of matching between a query and a particular caption candidate is determined according to the weighted score of each group. A list of captions will be eventually returned according to a user-given threshold.

The conceptual model has been implemented into a prototype, on which some experiments have been done. The behaviours and results of the working model will be discussed in the final chapter.

Keywords

Computational Linguistics, Information Retrieval, Multimedia, Natural Language Processing

CHAPTER 1

INTRODUCTION

Where is the life we have lost in living ?
Where is the wisdom we have lost in knowledge ?
Where is the knowledge we have lost in information ?

Thomas Stearns Eliot (1888-1965)
The Rock (1934) Pt. 1

Due to the advance in hardware technology, storage and retrieval of multimedia (or simply media) data become available at diminishing cost. Multimedia objects may be stored in various devices, especially those optical media such as CD-ROMs, which are supplied inexpensively at high capacity. The media objects may be digitized and saved in computer storage. A photographic image can be scanned into a standard or proprietary image format and stored as a binary fragment of a master database, in some well-accepted formats like GIF, JPEG, PhotoCD and TIFF [1]. An instance of application is the storage of records including digitized photographs of staff in the personnel record of an organization. A movie segment can be stored in a widely accepted format, such as MPEG [2].

Alternatively, it is possible to store an object with an appropriate analog device, which is under the direct or indirect control of the **Multimedia Database System** (MDBMS). For instance, movies can also be stored in an ordinary video cassette which are played back with VCRs controlled by the MDBMS.

For simplicity but without losing generality, digitized images will serve as examples of "imaginary" media objects in this research. In computerized media data, each object consists of at least, stated or not, two catagories of data elements, namely the registration data and the raw data.

The registration data contains details needed for the right interpretation of what the raw data is, e.g. encoding method, whether it is a digitized audio sound

[1] Several names among the most common bitmap formats across various platforms
[2] A well-accepted digital movie format which contributes to the Video-CD standard

track, image bitmap, or something else. The raw data is the binary bit stream representing the media object itself. Data of this kind is normally compressed or encoded in some way for storage space saving.

Up to the present moment, the software for manipulating multimedia database has not yet kept pace with the hardware technology. For instance, a digitized image can be simply stored in inexpensive mass storage but the retrieval is not yet efficient. An ideal way to retrieve media objects is for the system to automatically analyze the media data themselves against the queries for close match. However, there is not yet a successful image analyzer that can achieve this. We shall discuss further about this in the next chapter.

Another practical solution is the reference to the paths/files which store the media data. In commonly used file systems such as those in MS-DOS or Unix, we can retrieve media data according to their paths, date of creation and even file size. However, this way is usually not satisfactory. In a multimedia database system, users are frequently interested to retrieve a media object not only by the details like the date of creation, the place of origin, and the format of the bitmap that may be given along with the multimedia data, but also by the content of the data itself.

Consider, for example, that a professor in zoology is preparing a seminar in which he is going to illustrate the habitat of whales. He attempts to find some pictures of interests to his audience, and he probably does not care when the pictures are created and how large their file size are. An ordinary multimedia retrieval system is not powerful enough to satisfy the content retrieval of this professor in zoology. Not likely does he remember exactly which files in the system contain these pictures. It is not likely that the professor wants to remember the filespec and path of the media data. Instead, he probably strongly prefers to state the description in a natural language as he does with people.

Our goal is to develop a technique for media data retrieval in a multimedia database system which interacts with its user in a more friendly interface. For instance, we want to let that zoology professor obtain the material by requesting through the interface what he actually wants, as natural as "a huge whale in an

ocean". If there is no such an exact match. The proposed system should recognize something else which is close enough to the request.

We suggest that media data are to be enhanced with natural language descriptions in caption form to depict the content of the data. We then match the media data contents against the user's request, also in natural language form. In this manner the problem of media data content search is transformed into a natural language processing problem and we can apply research done in natural language for our solution.

In addition to two catagories of mandatory data elements described above, i.e. the registration and raw data, the third data element is proposed to be attached to each media object. It is the content descriptor for the media object. The formulation and retrieval of such a kind of content description are the goal of our project. The rationale is going to be discussed in the next chapter. The project is named ARMON to reflect its objective, "Approximate Retrieval of Multimedia Objects by Natural Language Captions".

In the rest of the thesis, we shall start off with a discussion of some background work related to our research. We shall then present our model and approach of breaking up a content description into small groups and derive the essentials of each group for the system's use. Through examples we shall describe how the derivation is achieved. We shall discuss our proposal for matching media data contents against the user query in a way similar to a person attempting to search the contents. Following that we shall present our approach to refine the process with the use of a knowledge base defined by users of the system.

Our work will be presented in the following chapters named Approach, Theory, Implementation, Illustration, and Discussion. The chapters Approach and Theory are hardware and software independent. In these chapters, neither particular software nor hardware will be discussed. Reader need not have knowledge about any specific operating system or specific programming language. For those who are only interested in the rationales and design concepts of ARMON, the chapters Implementation and Illustrations can be skipped.

For those who are interested to know more about the implementation details such as the data structure of the caption database, the parsing mechanism and how they are related to a real life system, the chapters Implementation and Illustration are useful. These are also references provided for those who want to perform further development based on ARMON.

Following those is the last chapter, Discussion, in which some of our findings, difficulties and potential improvements are presented. Finally, there are the Appendices, in which supplementary information such as the directory structure, manual pages for users, glossary, some details of imported modules, and some examples of captions and queries will be given.

CHAPTER 2

APPROACH

Die Grenzen meiner Sprache bedeuten die Grenzen meiner Welt
(The limits of my language mean the limits of my world)

Ludwig Wittgenstein (1889-1951)
Tractatus Logico-Philosophicus (1922) p. 148

In a multimedia database management system, the data probably comes in various forms like static graphical image, audio signal and video signal. In addition to the ordinary alphanumeric data, one may expect that the users of such a system will sometimes request the search of the media data by their contents. As automatic search of contents in media data like image is beyond the capability in today's computer systems, we must find alternative ways to accomplish this task. In some previous works [Holt90, Lum90], a proposal of attaching natural language descriptions to the media data was suggested. When a user query, also given in natural language form, to retrieve the media data by contents is received, the system will match it against all the content descriptions in the database. The search process has thus been transformed into a natural language processing (NLP) domain in which research work has been conducted for a number of years.

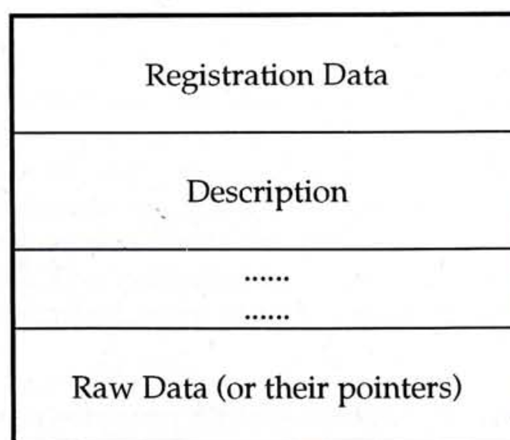


Figure 2.1 A Logical Block of Media Object

Figure 2.1 shows a simple logical block of media object. "Logical" here means that the data elements are logically inter-related but not necessarily located physically together. Description is the element which will be the focus of our NL-based retrieval of media object. Although a lot of NLP integration on

information retrieval has been attempted, today most (if not all) practically used systems are still keyword-based [Smea90, Maul89]. That is, the matching of a piece of text is totally based on the lexical matching of selected keywords within that document, in contrast to the collective semantic meaning in a natural language retrieval system. This approach is oversimplified and is rarely found fully satisfactory.

2.1 Challenges

While it is appealing to apply natural language processing techniques for media data content retrieval, one will quickly discover that there are numerous unsolved problems in natural language processing. For example, while researchers have been able to construct parsers for sentences, understanding their meanings and semantics is still very much in an early stage of its development. Researchers encounter difficulties in finding methods for a computer system in recognizing the different styles of describing the same thing in natural language.

In the past two decades, much research effort had been devoted to natural language information retrieval but result is still far from maturity. Many researchers became disappointed. Smeaton had put his comments as follows [Smea90]:

> In the 1960s, automatic natural language processing (NLP) techniques were seen as quite a desirable computing technique to include into overall information retrieval, but the successful integration of automatic language into information retrieval did not prove to be as easy as was initially hoped, and the idea of integration was soon dropped. This happened because the complexity of natural language texts in general, were grossly underestimated.

Fortunately, unlike the practice in natural language understanding or normal database applications like a bank's account management, a user who requests a retrieval by content in the media data does not really need exact matches, but something that is close to his interests, as depicted by a query. We shall assume that this is the case. Our approach is to define a way that approximates how a person may do in similar circumstances.

Further, the retrieval of captions for multimedia data is somewhat different from the retrieval of information from full texts. The latter usually contains a great variation of syntax and semantics. For example, metaphors like "Time flies like an arrow" and "I saw a man on the hill with a telescope" [Seo89], are typical ambiguous sentences in free text. In restricted structures and domains, similar phrases will be avoided. For the purpose of retrieving multimedia data in our situation, the natural language expressions should be more consistent. We propose to describe the contents with captions in stylized, restricted form of natural language. Although captions frequently occur as phrase, we assume them to be complete sentences, with Subject, Verb and Object (SVO). This structure is the general grammatical structure we allow for captions. Without loss of generality, for illustration process, we shall assume each caption to be a single sentence.

2.2 Knowledge Representation

We aim to build a model that allows us to represent information at different levels of constructs like the explanation of molecules being composed of atoms which are, in turn, composed of electrons, protons and neutrons. Such an approach in knowledge representation is sometimes called the structural primitives approach [Drey81], and is used in other streams of science for explaining complicated and complex phenomena with simpler concepts.

In the world of cognition, one can recognize a complex object without knowing much inside details, or recognize the components without knowing the complex object. In either case, it is essential to find some ways to represent knowledge that allow us to process information in a top down or bottom up manner. At this time, there is neither universal set of primitives nor levels of concepts having been defined for constructing knowledge applicable all over the world. It is more practical to build primitives for a definite domain. We are going to propose one way that is similar to the way people think and that we believe is broad enough to represent complicated concepts in a selected domain of application. In the coming demonstrations in ARMON, the habitat of animals will be used as the domain of application.

Representing knowledge in a logical form is an essential ingredient in the retrieval of information. Semantic nets, frames, and first order predicates are the most common methods. While these methods can be used to represent knowledge, they are at a lower level of representation for our purpose. We want to start at a higher level concept without much concern on the low level details at this time. We use the term "low level" here to point out the involvement of much inside details, and use the term "high level" for the involvement of less in-depth details. The convention is similar to the way used to distinguish low level programming from high level programming.

With examples in our domain, "the horse is eating grass" requires high level representation only. "The large horse is eating green grass" requires a bit lower level details. "The big, brown horse by the tree beside the river is hungrily eating the green long grass" requires much lower level details for representation.

2.3 Proposed Information Model

We attempt to extract fundamental knowledge from higher level concepts. As stated before, expressions in natural language are frequently highly complex. A lot of researches have been conducted in an attempt to decompose complex natural language statements into simpler components. Fillmore [Fill68] generalized the components of NL expressions as thematic roles. Some typical instances of thematic roles are Agent, Object, Beneficiary, Location, Time, Instrument, Manner, etc. [Wend91]. All sorts of complicated NL statements are decomposed into primitives which are subsequently filled in these thematic roles. The size of the set of valid thematic roles is highly dependent on the domain to which they apply. We shall simplify these approaches for the construction of our captions. We borrow similar concepts for our model; thematic roles are restricted to agent, action and patient. They are also collectively known as semantic groups in our model.

As mentioned above, for the purpose of media data description, the captions can be restricted with grammatical constraints without substantial loss of generality. Rules are established to transform each NL caption into three semantic groups,

namely the **agent** group, the **action** group and the **patient** group. Such a sequence is referred to as the AAP in the following context. In ordinary cases, the agent group corresponds to the subject part of the sentence and its associated phrases and descriptions, the action group corresponds to the verb part likewise, and the patient group corresponds to the object part as well. Semantic group information on the captions, along with their roles in the sentence (namely which semantic group) will be stored in the database for searching. For our purpose, discourse relations which are analyzed in a lot of Natural Language Understanding (NLU) systems and Natural Language Generation (NLG) systems, (e.g. [Dali91] and [Mann89]) will not be focused here.

With a set of BNF-like notations, the logical structures representing queries and captions in our approach are simplified as Figure 2.2.

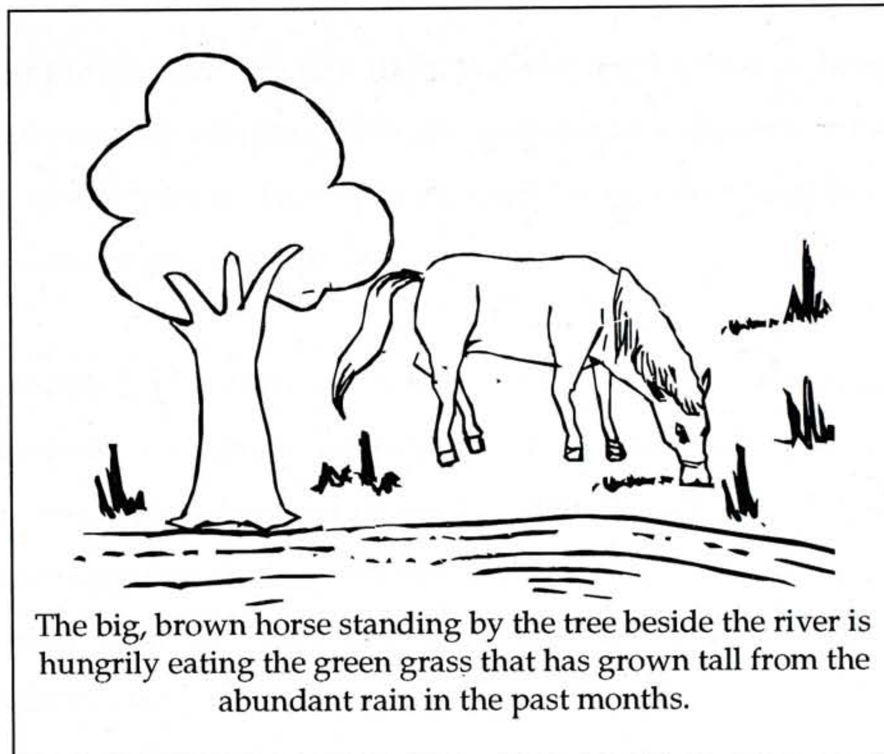
caption	::=	capid · sentence
query	::=	sentence
sentence	::=	{ agent [$expfea_{ag}$], action [$expfea_{ac}$], patient [$expfea_{pa}$] }
$expfea_x$	::=	$attr_{x,1}$, $attr_{x,2}$, ... , $attr_{x,n}$
$attr_y$	::=	$slot_y$: val_y
capid	::=	<numeric_string>
agent	::.=	<string>
action	::=	<string>
patient	::=	<string>
slot	::=	<string>
val	::=	<string>

Figure 2.2 The Semantic Structure of Captions and Queries

Here is a simple example, "a large cow is eating green grass". The logical primitives of the sentence are stated as follows.

SENTENCE = {cow[size:large], eat[], grass[colour:green]}

As mentioned before, it is more realistic to confine the application within a definite domain. In our project ARMON, the domain of knowledge is set to the world of living organisms. For illustration, imagine now that there is a media datum with a corresponding caption as shown in Figure 2.3.



The big, brown horse standing by the tree beside the river is hungrily eating the green grass that has grown tall from the abundant rain in the past months.

Figure 2.3 An Example of Media Data

It is notable that the caption can be described with much more details which are not observable from the media data directly. For instance, nobody can tell from the picture that there have been abundant rain in the past months. Another advantage of this caption description approach is obvious, the attached caption is able to describe "invisible" information for a particular media object. Of course, the description author is encouraged to write only essential information in the captions. Low level details should be avoided.

Such a complex sentence in our example is partitioned into three semantic groups :

i) Agent **the big, brown horse standing by the tree beside the river**

ii) Action **is hungrily eating**

iii) Patient **the green grass that has grown tall from the**

abundant rain in the past months

Each group contains a main item, known as **head** of the group. This is the most essential part the group. In the above example, horse, eat and grass are the heads of the agent group, action group and the patient group respectively. In the rest of the thesis, we sometimes use, in short, the terms "semantic head" or "semantic group" to stand for "head of semantic group".

With this simple picture, we can illustrate the difficulties of image analysis briefed in the previous chapter. Nobody would likely disagree that the visible information of this picture is simple enough for human being to understand, even for a kid just learning to speak.

On the other hand, it is difficult for any existing computer image analysis algorithm to deduce a simple message like "the cow beside the tree is eating grass". There are horse, grass and tree in the picture. Even if an intelligent image analyzer can recognize these components, it has difficulty to associate these things correctly. As in the 2-dimensional picture, a machine analyzer may find a horse which is located on the right side of tree. It cannot say whether the horse is close to the tree in the situation that objects in 3-dimensional space is displayed on a 2-dimensional material. Even if two entities are touching each other on a 2-dimensional display, it is not necessary that these two objects are close in the original 3-dimensional space.

This above arguments serve to re-emphasize that a caption is useful (if not necessary) for the purpose of retrieving multimedia objects.

2.4 Restricted Language Set

As mentioned somewhere before, the "free style" natural language is not encouraged for caption description in ARMON. We would apply some restrictions on the language set for building up the sentences in captions and queries. We borrow a classical example to show the necessity of restrictions for minimizing ambiguity [Hist87].

Nadia saw the man in the park with the telescope.

There are apparently several possible explanations to this short sentence.

(i) Nadia used the telescope to see the man who located in the park.

(ii) Nadia saw a man holding a telescope. The man was in the park.

(iii) Nadia stayed in the park and saw the man through the telescope.

(iv) Nadia saw the man in a park. The park had a telescope installed.

It is quite "natural" for somebody to associate one of the above sentences in daily used English. Hirst argued that some additional rules should be imposed to reduce the unnecessary ambiguity [Hist87]. A prepositional phrase should be attached (placed closely) to the item being described. According to this rule, the sample sentence "Nadia saw the man in the park with the telescope" probably refers to explanation (iv) above. By alternating the position of the questionable prepositional phrases, the sentences get the explanations from (i) to (iii) probably look like :

(i) Nadia saw, with the telescope, the man in the park.

(ii) Nadia saw the man with a telescope in the park.

(iii) Nadia in the park saw with the telescope the man.

Although it is possible that these sentences can still have some other meanings, the ambiguity has been greatly reduced. We agree, for ARMON at least, that the same restrictions should be applied to the grammar used in sentences for describing captions and queries although ambiguities do not always exist. Ambiguity occurs in the above example because the action "see" does not impose and spatial restriction on agent and patient. Other actions may be different from "see"; some of them may impose restriction on the relationship between agent and patient. The action "eat" is an example that have spatial restriction on agent and patient. We have, again, the following example.

A large horse is eating the grass in the farm.

With common sense, the horse which is eating grass in the farm must also be located in the farm since grass is found in the farm. i.e. a cow cannot eat grass remotely. This meaning is implied by the action eat. If we consider again the action "see", ambiguity is observed.

A large horse sees the grass in the farm.

In this sentence, a horse in somewhere else can see the grass in the farm because the action "see" can be a remote action. The horse may or may not locate in the farm when it sees the grass. The example shows that ambiguity does not always occur in one pattern. However, the restriction of keeping attributes close to the host entity should be applied to minimize any potential ambiguity.

In current implementation, the grammar of our restrict language set is represented with Context-Free phrase structure rules[1] shown in Figure 2.4.

S	=	NP · {PPP} · {VP · {NP · {PPP} } }
NP	=	DET · AJP · N I AJP · N I N
AJP	=	AJ · AJP I AJ
VP	=	AVP · V I V
AVP	=	AV · AVP I AV
PPP	=	PP · N · PPP I PP · NP

Figure 2.4 Restricted Grammar Defined for Sentences in ARMON

With these restrictions, we may imagine some good and bad examples immediately.

Proper sentences

i. A horse is eating long grass
ii. A horse is eating long grass in the farm
iii. A large horse is quickly eating long green grass in the farm

[1] The abbreviations used to present the grammar are found in Appendix A

iv. A horse in the farm is quickly eating grass

Improper sentences

v. A horse is eating long grass quickly

vi. A large horse is eating grass which is long

vii. A horse which is large is eating long grass

The examples in sentences (i) to (iv) are proper since they all follow the restricted grammar. Sentences (iv) to (vi) are improper since they disobey the grammar. In (iv) the adverb "quickly" should be placed immediately in front of the verb "eating". In our grammar, relative pronouns should be avoided but sentence (v) and (vi) disobey this rule. It is observed that improper sentences can usually be replaced by alternatives which fit our restricted grammar.

With a grammar defined like this, the caption of the media data shown in Figure 2.3 is better re-written as in Figure 2.5. The phrase "that has grown tall from the abundant rain in the past months" has been omitted since it is regarded as subsidiary information.

The big, brown horse by the tree beside the river is hungrily eating the green long grass.

Figure 2.5 Caption Rewritten to Fulfill our Grammar

We agree that the current grammar (Figure 2.4) is a bit restrictive. In the future, it is possible to expand the grammar according to the requirements of other domains. For instance, to allow descriptive text like (present) participle, we may add a rule like:

$$S = NP \cdot \{PTP\} \cdot \{PPP\} \cdot \{VP \cdot \{NP \cdot \{PTP\} \cdot \{PPP\}\}\}$$

$$PTP = VP \cdot NP$$

where PTP stands for Participle Phrase

With this enhancement, a sentence like "a horse having short legs is eating grass" becomes acceptable. To make this relaxation possible, more intelligent transformation from the lexical form into logical representation is required. Extension like this would be a good issue for future research.

In this chapter, we have already conducted the design basis of ARMON. In the next chapter, we shall have further details in formulating the knowledge model.

CHAPTER 3

THEORY

'What is the use of a book', thought Alice,
'without pictures or conversations ?'

Lewis Carroll (1832-98)
Alice's Adventures in Wonderland (1865) Ch. 1

In this chapter, we shall go through more design aspects of ARMON. Firstly, there will be a bit of general background about Knowledge Representation and Natural Language. Secondly, the concepts how to represent and store the captions will be discussed. Then the algorithm of matching between captions and queries will be explained. The matching score which indicates the closeness of matching will also be illustrated.

3.1 Features

To write a caption, the caption author usually states a collection of properties associated with each semantic group, such as *big, brown, hungrily, green,* etc. This collection of properties is called the set of explicit features of the group. These properties are similar but not identical to the implicit features which will be discussed later in this thesis. The notation of set theory will be widely used here to represent the set of features.

We are going to face the problems of constructing a collection of features, and their permissible values. These two problems are the major difficulties in the model of this kind. Papers on NLP by some researchers (notably Dahlgren [Dahl89] and Gallant [Gall91]) conveyed such an idea, and tried to work out a solution but had not yet come up with a comprehensive and complete proposal that could be broadly used, except in a narrow domain of interests.

Dahlgren [Dahl89] figured out a set of 55 features which was claimed to be sufficient for nouns in general use. Similarly, a set of 10 features was created for verbs. However, details of the rationale to reach those sets of feature slots and

the application of these features had not been mentioned in the reference. Gallant [Gall91] conveyed the idea of context vector which is in fact a matrix of features filled with numerical values instead of narrative values. He indicated more than 85 features as a 5 by 17 matrix for the entity astronomer but he only gave the rationale and usage for a few of these features.

In addition, there were other researchers who wanted to construct finite set of characteristics for conceptual substances defined in a specific and narrow domain. As these did not fit well into our goal, we would not go into them here.

In applying the feature concept of representation, we noted the difficulties risen from the works of others. We feel that the chance of success will be much higher in our task because nearly all works from those like Dahlgren and Gallant were intended for natural language understanding which requires precise semantics as well as complex structures in natural language construction. In our case, we attempt to define a method which approximates roughly the process how people match information beyond keyword matching, not about precise and general natural language understanding system.

We studied the set of features from Dahlgren [Dahl89] and Gallant [Gall91] and eliminated some features that are too vague and unclear to be of use. Many features slots given in their articles were not exemplified with feature value. For some of these features, we could not imagine what type of values would be appropriate, e.g. "*in extension of*", "*legal requirement*", and "*physiology*". Features which were either mystery to us or irrelevant to our application domain had not been employed in the current model. Some poorly defined features were re-defined by us, such as *state* and *status*. No explanation of similarities or differences between these 2 particular features was given in the original article. We redefine these two features and write them with the following representation.

state : solid | liquid | gas | plasma
status : excited | happy | sad | tired

The above representations are not covered here but will be explained later in this section. We define *state* being the physical form of a physical objects while *status*

tells the mental state of an living organism. Features of some real world entities are given in the following examples.

fea (air)	=	{ ....., *state : gas*, ... }
fea (sun)	=	{....., *state : plasma*, ... }
fea (water)	=	{ ....., *state : liquid*, ... }
fea (horse)	=	{ ....., *status : excited*, ... }
fea (cow)	=	{ ....., *status : tired*, ... }

Having eliminated or redefined some useless or unclear features, we put additional features in our domain. Two examples are arbitrarily pointed out here :

life : none | embryo | living | dead
occurrence : natural | artificial

For example, we clearly have:

fea (horse)	=	{ ....., *life : living*, ... }
fea (horse)	=	{ ....., *life : dead*, ... }
fea (egg)	=	{ ....., *life : embryo*, ... }
fea (stone)	=	{ ....., *life : none*, ... }

It is not reasonable to list all eliminated, redefined and added features with full examples here. To a considerable extent, the definition of features are somewhat subjective and domain sensitive. A final set of the features used in ARMON at current stage is given in Appendix B.

Unquestionably, when a larger domain of application or a larger number of application domains are studied, we believe that these sets have to be increased. As our task for the time being is to come up with a model that can be generally applied, we think that it is not a great benefit now to come up with a complete set of features.

With this background on the feature set model, let us now turn back to discuss our feature definition and their assignment of values. Each feature is made up of a slot and a value. The slot describes the identity of the feature and the value tells the property of an instance. For example, a characteristic or property of any object is the colour of the object; thus one feature slot is *colour* and it can have permissible values of *colourless, red, green, blue* and *violet,* etc., represented as follows :

colour : none | red | green | blue | yellow ...

Assume for a particular physical substance that its *colour* is *green.* Assume further that *green* is found as a value in the permissible set, the slot *colour* is instantiated with the value *green.*

colour : green

Like the definition of feature slots, the definition of feature values is somewhat subjective and cannot be comprehensive. To enhance the ability to recognize more feature values, we build into the system an additional mechanism. Suppose that there is a horse of *mud-colour.* This colour is not a permissible value in the set of colours and hence no exact colour is now available. With the use of the knowledge base as transformation rules stored in ARMON, which will be discussed later, the system will recognize a *mud-colour* horse being closely equivalent to a *brown* horse.

colour : brown

The deduction of a meaning like this is generally known as normalization [Lewi89], or **feature transformation** in our thesis. As it is not possible for anybody to ensure that there is always an exact meaning in the value set for the statement, the closest possible value will be attempted as an approximation. This is called the feature level approximation in ARMON. There are other levels of approximation in ARMON.

Consider another example and assume now that there is a phrase, *golden* sun. In the set of permissible colours, we do not have "*golden* ". There should be some sorts of knowledge able to determine a value which is close enough to the original. In this example, a *yellow* sun may be rated as a close approximate as gold is somewhat *yellow* in colour. The feature *colour:yellow* will be the close enough for sun in the context.

3.1.1 Superficial Details

In the rewritten caption, "the big, brown horse by the tree beside the river is hungrily eating the green long grass", in Figure 2.5, horse is the head of the agent group. *Big* in *size, brown* in *colour* and *next to* tree are the secondary properties of horse. These are called the superficial details which are explicitly stated in the sentence. They will be converted into explicit features in a later stage. Together with the restricted grammar, we are also told that the tree is growing by the river from the clause "beside the river". It is directly related to tree but indirectly to horse and is regarded as the tertiary description for horse which will generally be ignored during the process of approximate transformation. A challenging task is the determination of the importance of the information stated in the sentences : essential, secondary, or supplementary. As their names implied, essential information which is core of the a sentence provides a bird-eye view of this sentence. In another extreme, supplementary information which appears as a decoration will be generally ignored. The secondary information helps the mid-level view of a sentence. Essential information is usually delivered by the semantic heads of a sentence. Secondary information is carried by their implicit and explicit features. Decorative phrases provide the supplementary information.

In the above example, secondary descriptions of the agent horse are shown as follows :

size : big
colour : brown
nearby : tree

All those features are explicitly stated in the caption. Besides this set of explicit features, there are other implicit features which are well-known but not stated in the sentence. Some examples are briefly stated below.

life : living

birth : baby

legs : 4

...

This kind of features is hidden or implicit and will be discussed later in this chapter.

For another semantic group, the action group, eat is head of the group. In common sense and in definition contained in the knowledge base, the adverb *hungrily* for eat is interpreted as "eating at a high speed". It is then represented in the form of feature slot-value pair as *speed:high*. This is another example of normalization.

The patient group contains grass as the head, and *colour:green, height:tall* as the explicit features. It is noted that the tertiary description for grass, "**from the abundant rain in the past months**", originally shown in Figure 2.3, has been omitted in the logical transformation since it is regarded only as the supplementary description. The parsed sentence in logical form now looks like:

{horse [*colour:brown, size:large*], eat[*speed:high*], grass[*colour:green, height:high*] }

It should be emphasized again that the ARMON project is neither focusing on machine translation nor creating a precise natural language understanding system. Hence an ultra-fine level of transformation for natural language captions into corresponding logical representations is not necessary. We are concerned only with the most important knowledge of each group. Other subsidiary information are treated with reduced priorities or completely

ignored. For instance, the tenses of verbs and the determiners, "a", "the", etc. are not considered important and could be ignored for the purpose of retrieval.

3.1.2 Hidden Details

In addition to the explicit features already mentioned above, there is another kind of characteristics equally or even more fundamental to the construction of knowledge in the world. As many researchers do, we believe that world knowledge is built up from entities which conceptually form a-kind-of hierarchies. These hierarchies are useful for the determination of the degree of approximation. Figures 3.1 and 3.2 are examples of fragments of these hierarchies. Some researchers prefer to use alternative names for hierarchies like **taxonomy** [Pape86] and **ontology** [Dahl88]. Strictly speaking, there are slight differences between these terms, but we shall not go into depth as this is beyond the scope of this thesis.

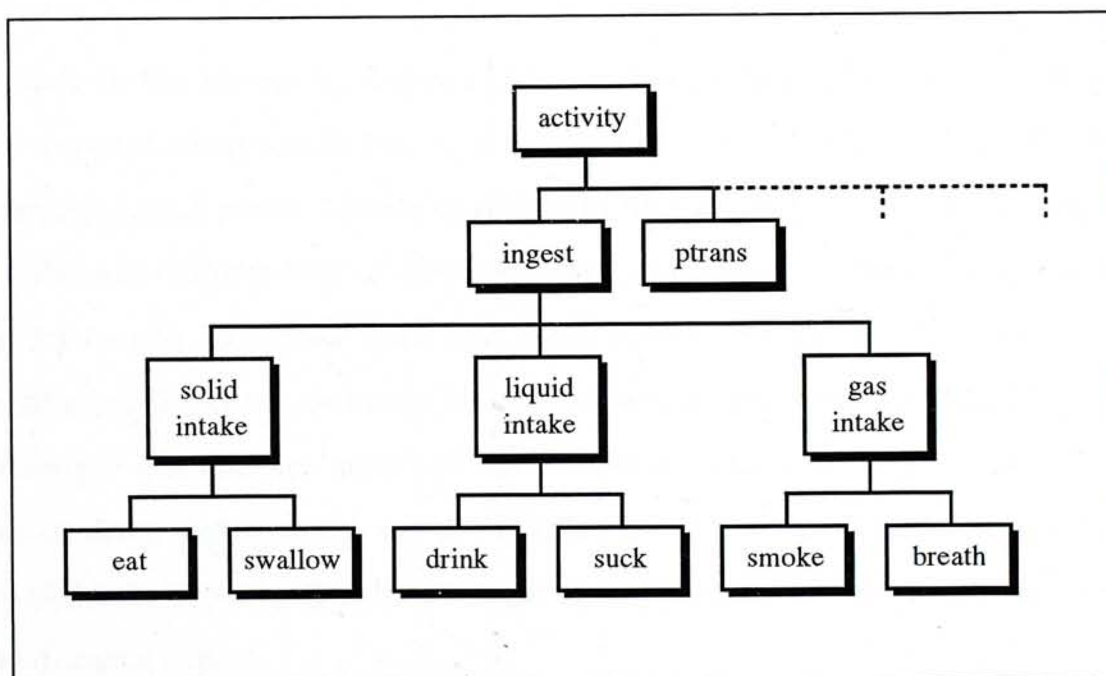


Figure 3.1 A Fragment of the Verb-type Hierarchy

In the model of ARMON, a hierarchy can be either noun-type or verb-type. It consists of entities, which are also called nodes, connected by links. It starts off from a root, and in it the descendants of a node are sub-categories of the node and inherit the properties of their ancestors, unless overruled by explicit specification with the same feature slots coupled with different values. This idea is familiar in object-oriented concepts and is broadly used elsewhere [Woel87, Holt90, Tsud91]. Miller put all his world knowledge including tangible and

intangible entities, including "objetcs", "actions" and "events" into a single hierarchy [Mil86]. Our model of hierarchy is similar to his but not exactly equal.

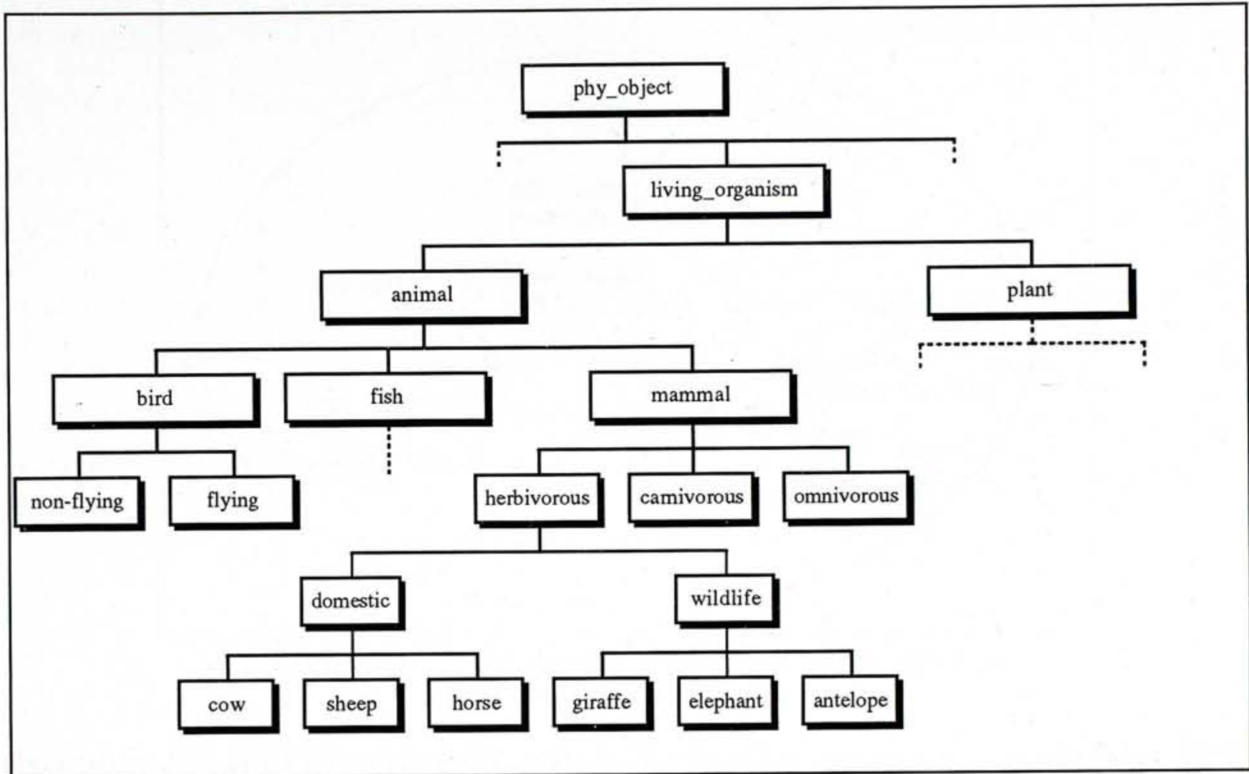


Figure 3.2 A Fragment of a the Noun-type Hierarchy

Each node in the hierarchy is associated with a set of primitive characteristics, i.e. the set of the implicit features of that entity. These features characterize the properties of each node. Nodes in different hierarchies of noun-type and verb-type will have different set of features. For example, some features for the noun hierarchy can be *life, moving_speed, size, colour, location,* and *age.* Implicit features of a node, like explicit ones, can only take on values from a set of permissible values. For example, the characteristics of node horse in the hierarchy intrinsically have values of *living, high* and *large* for the feature (slot) *life, speed* and *size* respectively. These characteristics and values are generally well known but must be defined by the domain expert.

In our particular example, the hierarchy starts off from the root phy_object. Living_organism appears at certain level in the hierarchy. Animal is a child of living_organism. By default, animal carries all the intrinsic characteristics of living_organism, such as *life:living.* At the next lower level, mammal is defined as a child of animal which is the parent of mammal. Herbivorous, carnivorous and omnivorous are the children of mammal. Herbivorous has sub-classes domestic and wildlife. In principle, the properties associated with domestic and wildlife are inherited from all the characteristics of animal.

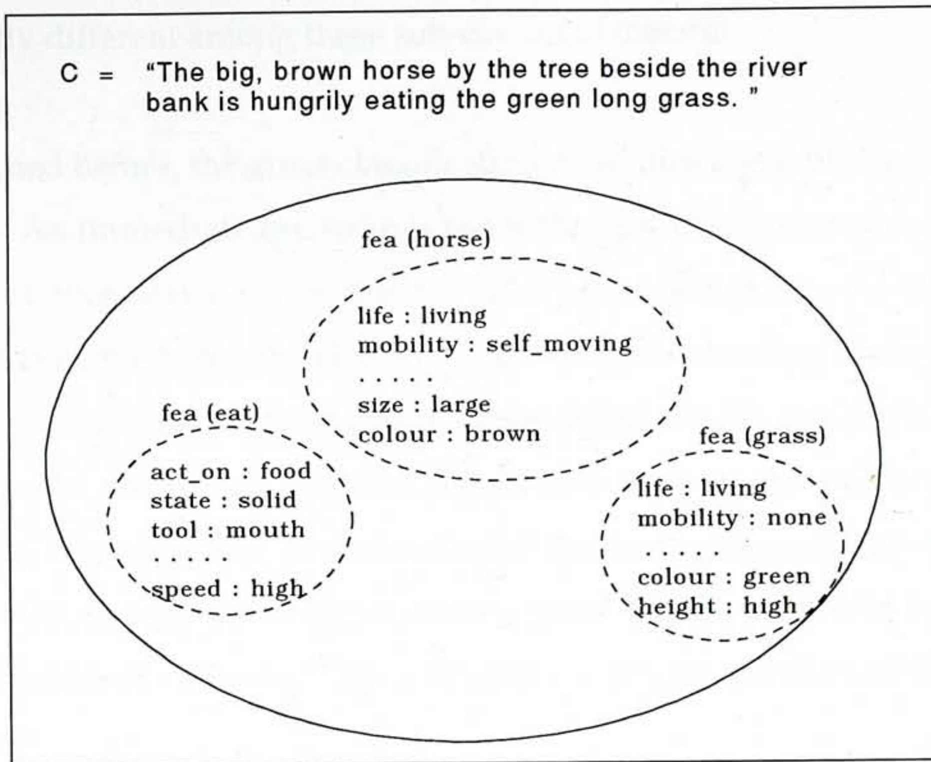


Figure 3.3 Feature Set of a Sample Caption

Exception is a provision through which the implicit features of an ancestor by its children can be overridden. Exceptions frequently occur at nodes of a-kind-of hierarchies like this. For instance, bird is a kind of animal which has the intrinsic property of *mobility:flying*. Chicken does not virtually have this capability although it is a kind of bird. This general implicit feature for bird can be negated for the node chicken.

In this instance, bird has sub-classes flying and non-flying. For example, Chicken and duck are non-flying (birds), Nightingale and pigeon are flying (birds). Dividing bird into flying and non-flying unlikely follows the best sense of classification for biologists, but it sounds good for general purposes. We do not need a biological or any other strict definitions to classify bird. For most people, the capability of flying is the most notable characteristic of bird, it is obviously sensible to classify bird like this. Feature *mobility:flying* is defined as a general characteristic of flying (bird).

For another animal, mammal, the flying capability is a property of only a few species, bat, for example. It is unwise to classify mammal into flying and non-flying because the sub-class non-flying (mammal) will be very small. On the other hand, mammal are highly characterized by another property, the feeding habit, which is broadly classified into herbivorous, carnivorous and omnivorous. Using this property,

mammals are clearly categorized. The limbs, teeth, digesting systems, etc., are significantly different among these sub-classes of mammal.

As mentioned before, the gross classification of entities is established on the type hierarchy. An immediate question is the technique of classification of entities. There is not necessary a single rule to classify the entities into the hierarchy. In this connection, the hierarchy should be constructed according to the need of the application domain, by a so-called domain expert. In the development of this project, we do not have a domain expert to construct the required domain knowledge. We, ourselves, pretended to be the domain expert although we are not. However, in real-world applications, domain experts should be separated from the system developers. This will produce less biased domain knowledge, we believe.

Instead of the top level classification starting from living_organism into sub-classes plant and animal. There may be another alternative classification by another domain expert. He/she may prefer to classify animal into aquatic and terrestrial, or into vertebrate and invertebrate, etc.

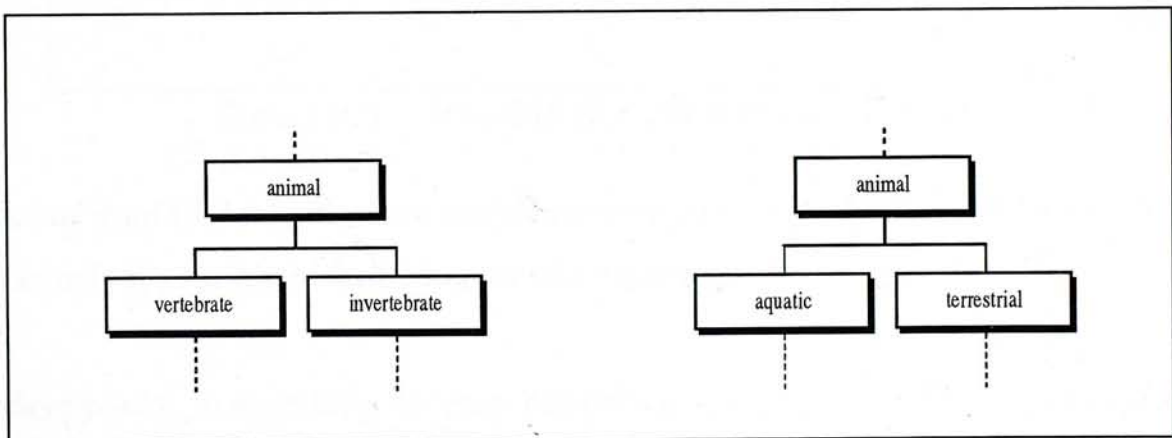


Figure 3.4 Two Alternative Classifications of Animal

The way of classification is unlikely a matter of right or wrong. It is just a matter of appropriateness in a particular domain. It is believed that the domain experts have the expertise to make a sound judgement. The above illustrations show some concepts behind the classification of entities into hierarchy. Other entities are classified on similar foundation.

3.2 Matching Process

A query is given just like a caption and will be partitioned into three semantic groups. The feature values of each group will be derived just like before. Matching is started in a pre-defined algorithm which will be discussed later.

Rowe [Rowe94] suggested six categories of results when a query is attempted to match against a set of natural language captions. Five of them are listed here, added with the (range of) scores. The sixth, "insufficient information given by the user", is treated as an intermediate but not a final result and will not be put here.

Relation between caption & query	Expected match score
Both are exactly identical	1
Entire query matches part of a caption	1
Part of query matches an entire caption	(0 to 1)
Part of query matches part of a caption	(0 to 1)
None of captions matches the query	0

Figure 3.5 Possible Results of Query Search

It is our goal to determine the match score in cases like the third and the fourth, for which approximate matching would demonstrate its role.

Conceptually, in searching for matches between a query and the stored captions in the database, we try to find matches with the main items first before looking at the subsidiary items.

Assuming now that there is already a query parsed into three semantic groups. The system starts the matching process by the sequence already determined with the pre-defined rules. Semantic groups in the query will match against the corresponding group of the captions, i.e. the agent of query is matched against the agents of the captions, and so forth. It locates the node on the hierarchy corresponding to the head of the semantic group of the query which will be matched first, and then attempts to match the feature values of the group

against the different sets of feature values representing the captions in the database with this same head. All the stored captions of the descendants of this node will be treated as if they belong to this node, with their heads replaced by this node. Thus, if there is a caption given exactly as the query, we get a perfect match. Even a caption is not lexically identical, we may still get prefect matching if the main semantic heads and features are equal.

3.2.1 Inexact Match

In case of exact match, the matching score is obviously 1 but it is unlikely that we always have perfect match. We need to define an algorithm to calculate the degree of closeness in those cases. This is done by calculating the matches between the feature values of the query's semantic groups and the values of corresponding groups in the stored captions. The score is then normalized by dividing the individual value matches by the total number of features, producing a value between the boundaries 0 and 1 with 1 being a perfect match. If further search is required, we move up the hierarchy to the parent node and perform searches on that node and its descendants using similar technique.

Searching around the type hierarchy can be viewed as a gross matching while the calculation of match score can be view as a fine matching. In ARMON, the former stage is also called **gross filtering** and the latter stage is put as **fine scoring**. Figure 3.6 simplifies these stages of matching captions against queries.

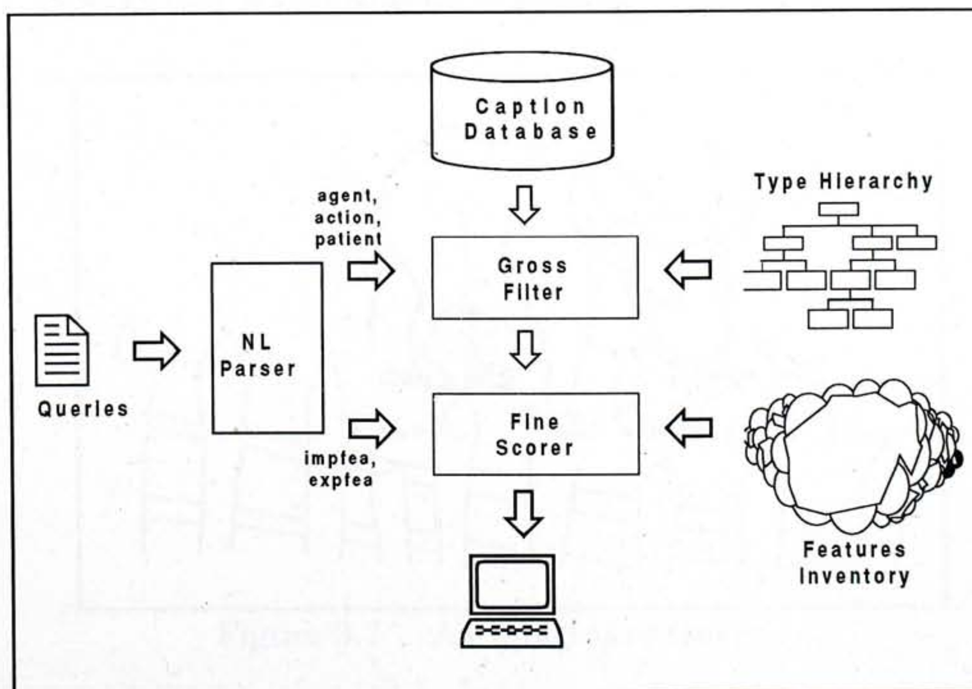


Figure 3.6 Matching of Queries

At the beginning of matching, features of each of the three semantic groups should be extracted. The degree of match between a caption and a query is determined by the relative overlapping between the caption and the query. One simple but reasonable way to quantify the degree of match with respect to the head of one semantic group is to count the number of feature values that are same in both the query and the caption. In this strategy, the features for each group are counted as matched or not, and given a binary 1 or 0 respectively. An integrated score for all three semantic groups will be subsequently calculated to see if the whole caption matches the query well, and will be used to rank the media data. This is done by calculating the integrated values of the three groups, again normalized to lie between 0 and 1. However the system provides a mechanism to state the relative importance of three groups which will be used to weight the groups' importance in the computation of the integrated score. In addition, the user can specify the threshold of match so that only those matches exceeding the threshold will be returned.

3.2.2 An Illustration

The following illustrates an example which demonstrates how to determine the matching score between a query and a candidate caption.

Assume a query Q issued by a user for retrieval of the content :

Q : **A large cow is eating grass in a farm**

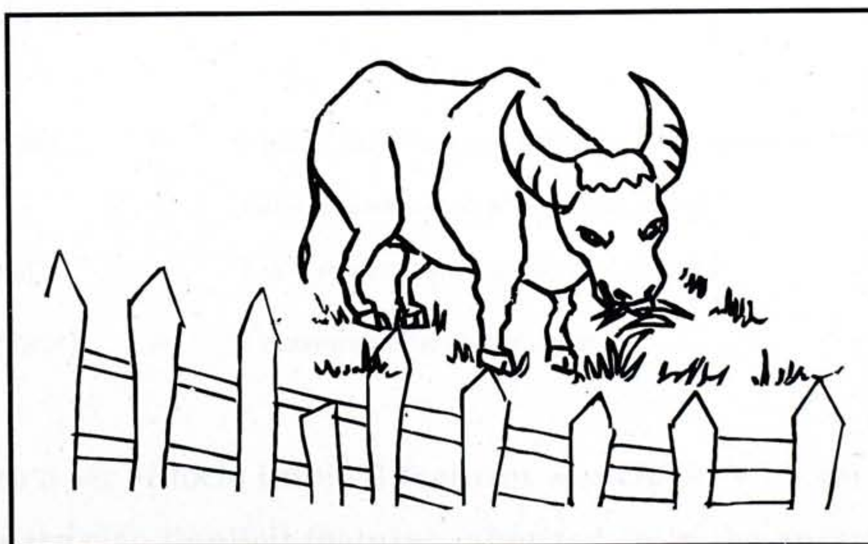


Figure 3.7 An Imaginary Query

The calculation of the matching score will be illustrated with the following stages.

3.2.2.1 Stage 1 - Query Parsing

The three semantic groups are extracted from the sentence.

Agent	**a large cow**
Action	**is eating**
Patient	**grass in a farm**

From the hierarchy in Figure 3.2, cow is a domestic, herbivorous mammal which is animal, animal is a living_organism and so on. Therefore, cow inherits the implicit features (impfea) from all its ancestors, i.e. domestic, herbivorous, mammal, animal, and living_organism. For simplicity of illustrations, we assume that only several features have been defined for each node on the hierarchy. The members of implicit feature set given to each node are defined by domain experts who should know well the properties of every entity in that domain. The process of feature definition and the match algorithm are much similar whether each node contains tens or hundreds of features instead of several in this illustration.

In the implementation, implicit feature slots and values of a particular entity are filled in the logical IMPFEA database. In this example, assume now that we find the following local implicit features for the semantic heads cow, eat and grass in the IMPFEA database.

$impfea_{local}$ (cow) = { *place_at:farm, function:serve_man, speed:low, manner:tame, habitat:domestic, legs:4, power:strong* }

$impfea_{local}$ (eat) = { *act_on:food, state:solid, tool:mouth* }

$impfea_{local}$ (grass) = { *mobility:none, colour:green* }

In addition to a set of local implicit features associated with each semantic group, there are also implicit features inherited from the ancestors on the hierarchy. Taking the entity cow as an example, it inherits all the local implicit features of living_organism, animal, mammal and herbivorous. By moving down the

hierarchy starting at living_organism, all the local implicit feature slot/value pairs of the ancestors of cow are gathered sequentially. The overall implicit feature set of cow will be obtained as a union of local features of itself and all its ancestors :

$impfea_{local}$ (living_organism) = { *life:living, occurrence:natural, state:solid* }

$impfea_{local}$ (animal) = { *mobility:self_moving* }

$impfea_{local}$ (mammal) = { *birth:baby* }

$impfea_{local}$ (herbivorous) = { *diet:plant* }

$impfea_{local}$ (cow) = { *place_at:farm, function:serve_man, speed:low, manner:tame, habitat:domestic, legs:4, power:strong* }

In addition to implicit features, the parser also extracts explicit features for each semantic group. The explicit feature slot/value couples are kept logically in the EXPFEA database with pointers to the corresponding captions. In this example query, only one explicit feature is given with cow.

$expfea$ (cow_Q) = { *size:large* }

Combining all implicit and explicit features, an integrated feature set for cow is obtained :

fea (cow_Q) = IMPFEA ∪ EXPFEA

= $impfea_{local}$ (living_organism) ∪ $impfea_{local}$ (animal) ∪ $impfea_{local}$ (mammal) ∪ $impfea_{local}$ (herbivorous) ∪ $impfea_{local}$ (cow) ∪ $expfea$ (cow_Q)

= { *life:living, occurrence:natural, state:solid, mobility:self_moving, birth:baby, diet:plant, place_at:farm, function:serve_man, speed:low, manner:tame, habitat:domestic, legs:4, power:strong, size:large* }

The cardinal number (no of elements) of the above feature set is 14. Action eat is treated in the same manner, but the details will be omitted here.

$impfea_{local}(eat_Q)$ = { *act_on:food, state:solid, tool:mouth* }

In the query, there is no explicit feature written for eat, i.e. $expfea$ (eat) = {}. Hence the overall feature set is just the implicit feature set.

$fea(eat_Q)$ = { *act_on:food, state:solid, tool:mouth* }

Similarly, we process the patient group and get the following result :

$fea(grass_Q)$ = ...

= { *life:living, occurrence:natural, state:solid, mobility:none, colour:green, shape:narrow, place_at:farm* }

3.2.2.2 Stage 2 - Gross Filtering

Each semantic group of the query, after being decomposed into semantic groups with features, is searched against the corresponding node in the hierarchy. Of course, it is desirable if one or more stored captions in the same node is matched with the query. All the implicit features for that node will be identical except for the explicit feature values. If sufficient result is obtained, searching for this semantic group will not go across to other nodes on the hierarchy, unless commanded by the user otherwise.

In case that no exact match or insufficient matches are produced from the captions for a particular semantic group, search will then begin from the parent and the sibling nodes and their descendants in the hierarchy. The system will search for approximate matches from the stored captions in these nodes. The process to calculate match scores will be shown shortly. If still insufficient result is produced, the system will move up the hierarchy one node further and search for matches in it and its descendants. The process will be repeated until sufficient result is obtained.

Let us consider the agent group, cow, first. Assume now that no exact match of cow can be found in any one of the captions. However, two sibling nodes, horse

and sheep are found in two candidate captions (Figure 3.8). Let them be arbitrarily labeled C_1 and C_2.

C_1 : **A white horse by the tree is drinking clean water**

C_2 : **A young sheep is eating short, brown grass in the forest**

Implementation level details are skipped here and will be re-visited in the next chapter. This stage, as it is named, works like a filter to collect captions which are significantly close to the query and pass it to the next stage.

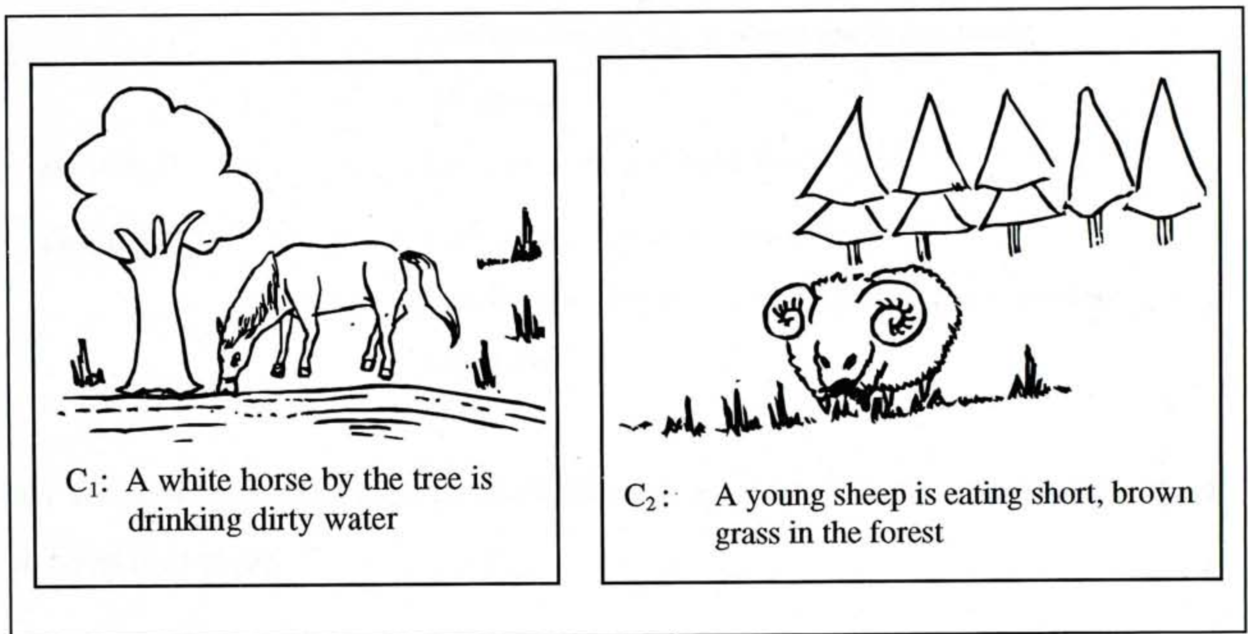


Figure 3.8 Two Pieces of Media Data Approximately Retrieved

3.2.2.3 Stage 3 - Fine Scoring

First, we have to decompose C_1 and C_2 into semantic groups with feature sets, in the same way done for the query, Q. The matching score will be determined according to the agreement between their corresponding feature sets.

C_1: **A white horse by the tree is drinking clean water**

fea (horse$_{C1}$) = { *life:living, occurrence:natural, state:solid, mobility:self_moving, birth:baby, diet:plant, function:serve_man, speed:high, manner:tame, habitat:domestic, legs:4, power:strong, size:medium, colour:white, nearby:tree* }

fea ($drink_{C1}$) = { *act_on:food, state:liquid, tool:mouth* }

fea ($water_{C1}$) = { *living:none, occurrence:natural, state:liquid, mobility:gravity, colour:none, shape:none, taste:none, appearance:clean* }

C_2: **A young sheep is eating short, brown grass in the forest**

fea ($sheep_{C2}$) = { *life:living, occurrence:natural, state:solid, mobility:self_moving, birth:baby, diet:plant, function:serve_man, speed:low, manner:tame, habitat:domestic, legs:4, power:weak, size:small, age:young* }

fea (eat_{C2}) = { *act_on:food, state:solid, tool:mouth* }

fea ($grass_{C2}$) = { *life:living, occurrence:natural, state:solid, mobility:none, colour:brown, shape:thin, place_at:field, height:low* }

Now we determine the degree of similarity, *sim*, between the agent groups, i.e. cow, horse and sheep

$$sim\,(cow_Q, horse_{C1}) = \frac{card\,(fea\,(cow_Q) \cap fea\,(horse_{C1}))}{card\,(fea\,(cow_Q))} = 11 / 14 = 0.79$$

$$sim\,(cow_Q, sheep_{C2}) = \frac{card\,(fea\,(cow_Q) \cap fea\,(sheep_{C2}))}{card\,(fea\,(cow_Q))} = 10 / 14 = 0.71$$

We see that the similarity value between cow in the query and horse in C_1 is higher than the value between cow and sheep in C_2. Based on this result, it can be said that cow in the query is closer to horse in C_1 than sheep in C_2. But this information is insufficient to indicate the overall degree of match between Q and

C_1, as well as Q and C_2. Other corresponding semantic groups between the query and a caption must be evaluated in the same way and the integrated scores will be computed.

For the action group,

$$sim(\text{eat}_Q, \text{drink}_{C1}) = \frac{card(fea(\text{eat}_Q) \cap fea(\text{drink}_{C1}))}{card(fea(\text{eat}_Q))}$$

$$= 2/3$$

$$= 0.67$$

Intuitively, $sim(\text{eat}_Q, \text{eat}_{C2}) = 1$

Similarly, for the patient group,

$$sim(\text{grass}_Q, \text{water}_{C1}) = \frac{card(fea(\text{grass}) \cap fea(\text{water}))}{card(fea(\text{grass}_Q))}$$

$$= 1/7$$

$$= 0.14$$

$$sim(\text{grass}_Q, \text{grass}_{C2}) = 5/7$$

$$= 0.71$$

The overall matching scores, or the overall similarity value between Q and C_j, $sim(Q, C_j)$, is computed as the weighted sum of scores of individual semantic groups. The relative importance of each semantic group is represented by a weight, w_i, as follows :

$$sim(Q, C_1) = sim(\text{cow}_Q, \text{horse}_{C1}) \times w_{ag} + sim(\text{eat}_Q, \text{drink}_{C1}) \times w_{ac} + sim(\text{grass}_Q, \text{water}_{C1}) \times w_{pa}$$

$$sim(Q, C_2) = sim(\text{cow}_Q, \text{sheep}_{C2}) \times w_{ag} + sim(\text{eat}_Q, \text{eat}_{C2}) \times w_{ac} + sim(\text{grass}_Q, \text{grass}_{C2}) \times w_{pa}$$

w_{ag}, w_{ac} and w_{pa} are the relative weights of the roles agent, action and patient respectively. For the simplicity of comparison in different retrievals, w_i's are normalized. The necessary conditions are :

$$w_i > 0, \quad \text{and}$$
$$\Sigma w_i = 1$$

Obviously, the value of function *sim*() must then lie between 0 and 1 and the degree of similarity can be easily observed. We believe that nobody other than the query issuer knows well the relative importance of the three semantic groups in his/her own query. The weights are preferably given by him/her. However, if no weight is given, they are equally weighted by default.

For illustration purpose, all weights w_i, are also assumed equal,

$$\text{i.e.} \quad w_{ag} = w_{ac} = w_{pa} = 1/3 \approx 0.33$$

then,

$$\mathit{sim}(Q, C_1) = 0.79\times0.33 + 0.67\times0.33 + 0.14\times0.33 \approx 0.53$$

$$\mathit{sim}(Q, C_2) = 0.71\times0.33 + 1\times0.33 + 0.71\times0.33 \approx 0.81$$

Thus in this example, the second caption C_2 is found to be a closer match with the query, Q.

An user-given **threshold** reflects the requirement of strictness of the retrieval (see Figure 3.9). It lies between 0.0 and 1.0. If the user gives a low threshold value, the retrieval will be loose, i.e. less close captions can be returned. In contrast, when a higher threshold value is given, e.g. 0.95, only the closest captions will be returned. In one extreme, the threshold value of 1.0 returns only those captions which exactly match the query.

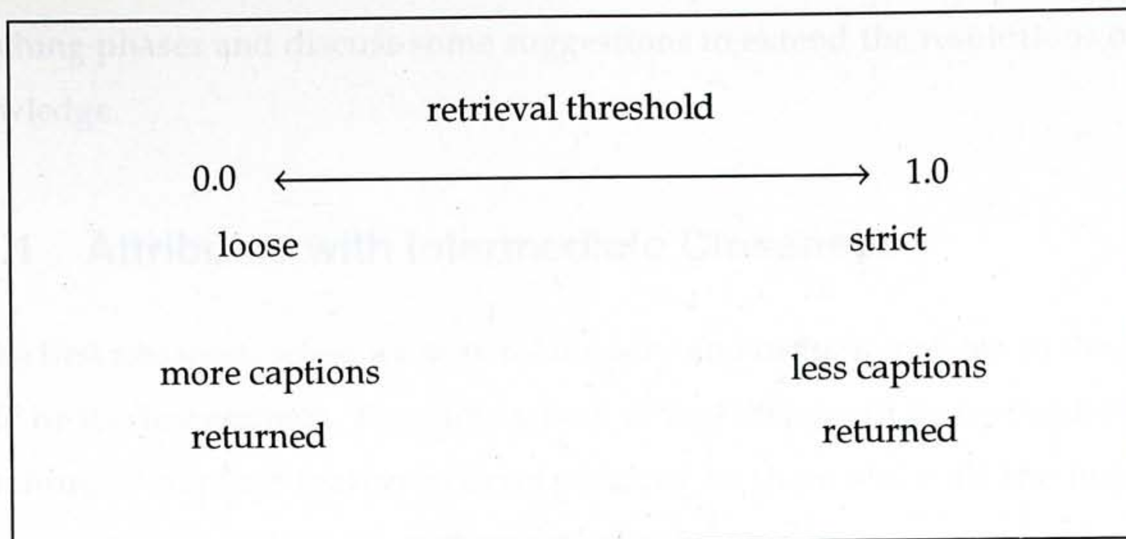


Figure 3.9 Effect of Retrieval Threshold on Returned Captions

In current example, either both, one or none of the captions will be returned, according to the user-given threshold, such as 0.5, 0.7 or 0.9.

Although the current stage of ARMON handles only the captions but not the media data themselves, it is imagined that two bitmap data associated with the corresponding captions are returned. It is hoped that these pictures may, to certain extent, beautify the boring text in this thesis.

To summarize this model, retrieval mainly goes through 2 stages, i.e. gross filtering and fine scoring. The former makes use of the knowledge embedded in the type hierarchies to sieve away the "irrelevant" captions. In other words, it collects only those captions that match closely the query. It then passes the "filtrate" to the next stage. The next step calculates the matching with accounts to the complete feature sets in captions and queries. This is one characteristic of ARMON. In contrast, other models using multiple stages retrieval employ different approaches. For instance, MARIE [Guli92] used the keywords approach for the front stage and hierarchy search for the back stage. In another project ADRENAL [Lewi89], the keyword sub-system was designed as the front end and other NL processor was used in the back end.

3.3 Extending Knowledge

In the calculations of matching scores between queries and captions addressed in previous section, the matching of individual features are binary, i.e. either 1 or 0. We now probe more deeply into two particular situations during the

matching phases and discuss some suggestions to extend the resolutions of the knowledge.

3.3.1 Attributes with Intermediate Closeness

In the first situation, when we search for query and caption matches in the same node or its descendants, the comparison of matching will be focused on the matching of explicit features corresponding to them since all the implicit features match except for the explicit feature values.

At this feature level approximation, sometimes the binary comparison values, 1 or 0, are too rough. For example, colour is a frequently used explicit feature for many entities. Assume now that different colours exist for one entity, say flowers. *Colour:blue* and *colour:purple* will be considered as either matched or not, and there is nothing in-between. But *colour:purple* is closer to *colour:blue* than *colour:white* for most people. To improve the degree of match for different features, we introduce some intermediate levels between the two extremes.

With another illustration, although *red* colour is somewhat different from *pink*, there are certain similarities between the two colours. Subjectively speaking, the feelings for most people are similar. Scientifically speaking, the prime colours components [1] are also significantly close. It is "unfair" to say that *colour:pink* and *colour:white* have the same degree of closeness (both 0) with respect to *colour:red*.

To solve this problem, we may assign a feature match score of 0.5 to the features *colour:red* and *colour:pink*, for example. In contrast, the features *colour:green* and *colour:red* would be given a match value of 0, as before. For most people, a *pink* apple is undoubtedly closer to a *red* apple than a *green* one. For illustration, some typical colour pairs are suggested as follows.

Colour Pair	Feature Match Score
yellow - golden	1
brown - mud-coloured	1
red - orange	0.5

[1] Red, green and blue, or RGB in short

red - pink	0.5
pink - violet	0.5
blue - purple	0.5
white - grey	0.5
white - black	0
red - blue	0
red - green	0
....	....

Figure 3.10 Examples of Attribute Matches with Intermediate Value

This kind of information can be stored in a mini-knowledge base, and we can assign a value of matching for individual features somewhere between 0 and 1. We can add any number of "steps" between 0 and 1, from very fine to very rough. For illustration purpose, it is unnecessary to give too fine grading levels. A step of 0.5 is believed to bring significant improvement over the "0 and 1" scoring. A domain expert is free to assign a very fine value if he has sufficient confidence to distinguish such a small difference, for example, something between 0.42 and 0.43. Actually, it is usually not necessary or useful to have such super-fine resolution for a single feature in the approximation at this stage.

3.3.2 Comparing Different Entities

In the second situation, if the head of a particular semantic group cannot be exactly matched (or insufficient matches) from any of the captions, nearby nodes on the same hierarchy have to be searched. When some of the nodes are found, the degree of matching will be determined with the steps mentioned in the previous section. But more information must be known before a fair judgement of matching can be determined. We find that the same feature slot and value pair can have different meaning for different entities. The difference is related to some intrinsic properties which have been ignored.

For instance, the feature *size:big* has been considered fully matched (match value of 1) for a *big* cat and a *big* horse. A *big* cat probably weighs about 50 kg while a *big* horse probably weighs over 500 kg. The absolute meaning is different for the two separate nodes on the animal hierarchy. Also, a *fast* jet plane may fly at a speed

above 2000 kmh^{-1} but a *fast* train probably runs no more than 200 kmh^{-1}. Therefore the common feature *speed:high* has different absolute meanings for different nodes on the vehicle hierarchy, such as aeroplane and train in this example. These inherited facts illustrate that a particular feature does not necessarily convey the same meaning for different entities, i.e. different nodes on a type hierarchy. It seems that more information is needed if we aim for a higher resolution in feature comparison. This is the purpose for extending the knowledge beyond ordinary cases.

Consider again the example of *big* cat and *big* horse. When we compare the feature *size:big* of cat and horse, we do not assign the match score of 1 for that feature, we only assign a value of 0.5 for that common feature which are "partially" matched. Similarly, we assign a value of 0.5 to the feature *speed:high* for a *fast* aeroplane and a *fast* train. It can be easily determined if the knowledge for aeroplane and *fast* train contains the information of typical speeds of both aeroplane and train.

This extra mini-knowledge base also keeps various knowledge for specific instances. For example, the feature *speed:high* for action eat in the phrase "eat *hungrily*" is also derived from the extended knowledge. It can be observed that the extended knowledge stored in the mini-knowledge base is helpful for determining a score some way between 0 and 1, and distinguishing finer details which are subject-sensitive.

We have already illustrated that the system can now use implicit, explicit and this recently derived information to assign the feature values, and use them for approximately finding matches between the query specification and the stored captions. The usage of the mini-knowledge base is dependent on how far we want to distinguish the details of individual knowledge. Of course, it improves the final scoring to certain extent. But in ordinary cases, it can be ignored in scoring as a first approximation. A balance between simplicity and accuracy of scoring should be considered. If the user finds that the matches from the system are not satisfactory, then he/she can instruct the system to use a better refinement from the mini-knowledge base.

3.4 Putting Concepts to Work

To implement our model, two modes of operation have been considered, the **unattended mode** and the **interactive mode**. The former will be designed to involve as few user interactions as possible. In case of any ambiguity, the system will try to determine on its own knowledge.

In contrast, the interactive mode assumes that the user is sitting in front of the inquiring terminal. He/she is ready to give response on the terminal on which ambiguities are displayed. Papegaaij in [Pape86] called operations like this **Disambiguation Dialogue**. In his book, a dialogue between computer and user was illustrated to disambiguate a phrase "capital development". The interactive mode of ARMON is similar to the dialogue like that.

Each mode has its own advantages. The interactive mode produces an anticipatory higher precision since it tries to resolve any uncertainty by "human intelligence" in case that "artificial intelligence" is not able to solve. However, this causes an inconvenience to the user since the user must be occupied for interactive response. It may take several minutes if the retrieval takes such a time duration.

The unattended mode saves user's time from sitting in front of terminal, waiting and standing by to answer questions from the computer. After the user has issued a query on the inquiry terminal, he/she may walk away to perform other tasks and return some time later to receive the results.

In this and the last chapter, we have already exhibited the concepts behind ARMON. All of the functionality of the model has been implemented except the extended mini-knowledge base discussed in Section 3.3. We are going to present more implementation details in the coming chapters.

CHAPTER 4

IMPLEMENTATION

We must use words as they are used or stand aside from life.

Dame Ivy Compton-Burnett (1884-1969)
Mother and Son (1955) Ch. 9

In this chapter, we shall discuss the overall structure of ARMON, the implementation of the NL recognition sub-system and the knowledge model which we have already proposed.

Unix is a widely accepted operating system on various hardware. It has been selected as our major development platform. Several Unix variants are available to us, namely the HPUX on the HP 9000 workstation, Solaris on the Sun SparcStation and Linux on an ordinary Intel x86-based PC. We intend to implement a portable system usable in these various Unix variations.

The C Language has been selected as the major programming language and the GNU C Compiler as the language tool in the project ARMON. This compiler has the advantages of portability, economy, adequate documentation and is operational on various Unix platforms. The software itself, as well as its documentation, is freely available on most public FTP servers on the Internet. Moreover, much help is available on the Usenet News on the Internet. The other two supporting toolboxes, the PC-KIMMO [Antw90, Antw92] and Metalbase [Jern92], have been chosen as the natural language parser and the database function library respectively. They are also functional and operational on the available Unix platforms.

4.1 Overall Structure

The overall NL retrieval system is constructed from several modules, known as the User Interface, the Parser/Matcher Shell, Database Handler, and associated supporting modules, described as follows :

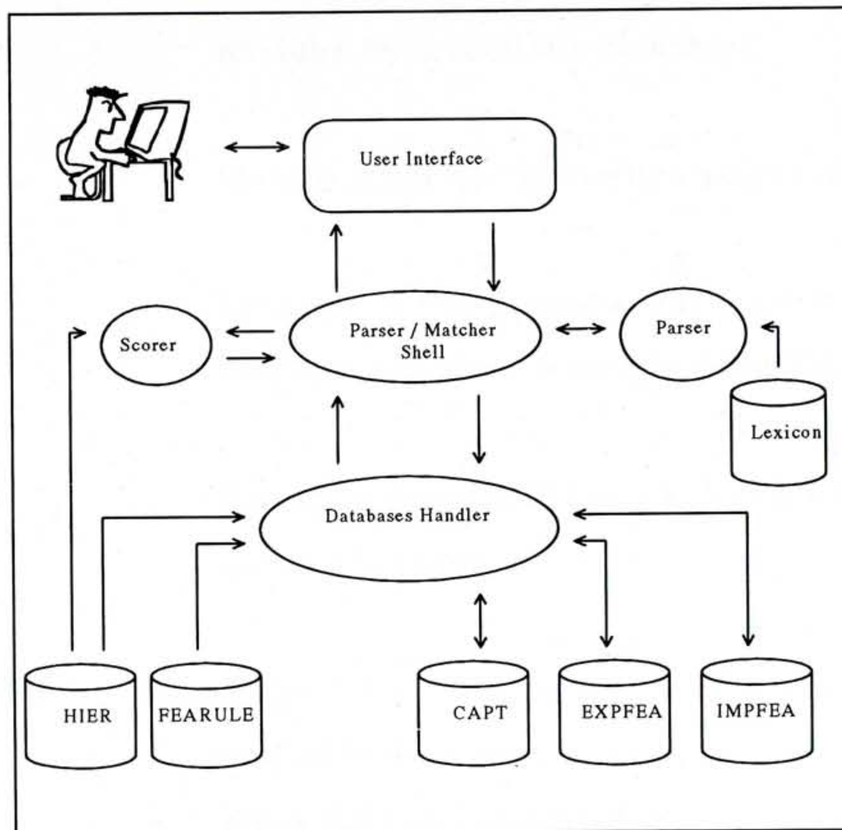


Figure 4.1 Building Blocks of ARMON

User Interface	Routines to handle the commands and data given by the users, format the results and display to the users. In this project, we developed only an ASCII interface, GUI is not our aim.
Parser/Matcher Shell	Routines to understand the user commands, coordinate the operations of the parsers, database handler and scorer
Parser	Routines to transform the natural language sentences into their logical representations. PC-KIMMO is used in ARMON.

Scorer	Routines with algorithms to determine the matching score of a particular stored caption and the sentence in query
Database Handler	Programs to maintain the lower level data base manipulations. It includes several different modules for several sub-databases.
HIER	Storage of the type hierarchy in logical form
CAPT	Storage of the most recent records of the captions. Metalbase is used in ARMON.
EXPFEA	Storage of the explicit features for each semantic group of captions in CAPT.
FEARULE	Table to keep the map for transforming adverbs and adjectives into explicit features, with the knowledge for normalization
IMPFEA	Storage for keeping the implicit details of entities on the type hierarchy
Lexicon	Dictionary used to determine the meaning of each word. ENGLEX is used in ARMON

In the above diagram, every block is a logical module. In this sense, one logical block may be composed from several separate physical fragments in the system. For instance, IMPFEA is composed from a number of data files instead of a single one. Another example is EXPFEA which represents scattered data files around the captions holding them, not a single data structure. All these individual data structures will be addressed later in this chapter.

Now let us probe a bit deeper into the implementation details. Since we shall not develop the whole system from scratch, we first try to look for a natural language parser with adequate capabilities and documentation for our purpose.

4.2 Choosing NL Parser

Instead of developing a NL parser from scratch, we intended to take advantage of an existing NL parser for our implementation. There were several NL parser prototypes published in various papers, such as TRUMP [Jaco92], DBG [Mont89], MCHART [Thom83], FRUMP [DeJo79] and its derivative, McFRUMP [Maul89]. Initially, it appeared to us that there should not be great difficulty to get an adequate NL parser.

At the beginning, we tried to find a full semantic parser. A lot of attempts had been tried and none was successful for some reasons. The selection of a parser was restricted by the various factors which essentially gave us few choices. We tried to look into a parser DBG [Mont89], originally written in Prolog. To fit well with the parser, we had actually considered using Prolog as the main development tool. DBG and Prolog were eventually excluded because a lot of unexpected problems were encountered. The difficulties in choosing a parser would be further discussed in Chapter 6.

Finally we chose a morphological parser PC-Kimmo [Antw90]. Compared to a full semantic parser, a morphological parser does not have the capability of semantic parsing on NL sentences. This kind of parser has a number of shortcomings compared to an ideal semantic parser. For instance, no semantic tree is generated by the parser. Under this limitation, the morphological parser does not distinguish "saw" being "a cutting tool" or "the past tense of a kind of perception with eyes".

Fortunately, different from machine translation systems, we did not need to get an accurate translation of each token in a sentence. In designing this system, we had considered how to make the translation as accurate as necessary for the retrieval purpose. To overcome part of the limitation of this parser, ARMON was designed to provide to the users two modes of operations, the interactive

and the unattended modes. These options will be illustrated below with examples.

4.3 Ambiguity

Assume now that the user issues a simple query, "The cat saw a mouse". The parser processes all tokens in the sentence successfully except the word saw. In the interactive mode, the system is designed to ask the user for ambiguity like the following example.

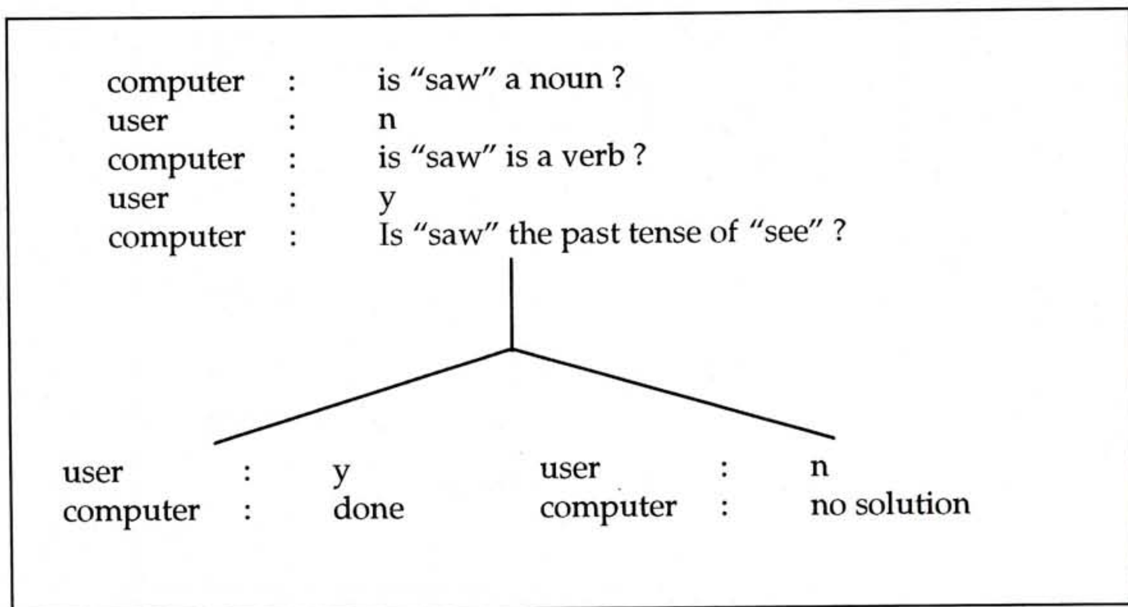


Figure 4.2 User Involvement in Interactive Mode

If the alternative mode, i.e. the unattended mode, has been chosen, the parser will try to make her own decision without bothering the user who issues the query. In this example, the parser will resolve the ambiguity by a simple rule of elimination. With our assumptions, each sentence contains the semantic group in the sequence agent, action and patient, i.e. with the lexical Noun-Verb-Noun, or N-V-N, structure. It can be sorted out without ambiguity that cat and mouse are nouns. The parser can hence predict by elimination that saw is a verb. Saw will be interpreted with the meaning "the past tense of a kind of perception using eyes" instead of "a kind of cutting tools".

Recalling that ARMON is developed from two sub-systems, the front-end parsing system and the back-end database system. The front end accepts a natural language description while the back-end stores the description in its

corresponding logical form. There are some matching and scoring mechanism in between.

PC-KIMMO is the core of our parser. The following is the original untouched output of a NL string "a cow is eating grass" using this parser.

```
a
                a           DT
cow
                `cow        N
is
                be          AUX.3SG
eating
                `eat+ing    V+PRG
grass
                `grass      N
```

Figure 4.3 Results of Parsing with PC-KIMMO

The output of the original parser is encoded with a set of notation which is defined as "gloss tag" in ENGLEX. The set is listed below but full output syntax will not be discussed here. ENGLEX is an English lexicon designed to work with PC-KIMMO. It contains approximately 11000 nouns, 4000 verbs and 3400 adjectives, and a total of 20000 lexical entities [Antw92b]. A brief description of ENGLEX will be provided in Appendix G; more details can be referred to ENGLEX User's Guide [Antw92b]. The gloss tags defined by ENGLEX are given as follows :

Gloss Tag	Meaning
N	noun
PN	proper noun
V	verb
AUX	auxiliary
AJ	adjective

AV	adverb
PP	preposition
DT	determiner
CJ	conjunction
QN	quantifier
DEM	demonstrative
PR	pronoun
IJ	interjection
FN	foreign
CD	cardinal
OD	ordinal
1	first person
2	second person
3	third person
SG	singular
PL	plural
GEN	genitive
CMP	comparative
SPR	superlative
PST	past
PTC	participle
PRG	progressive
NR	nominalizer
VR	verbalizer
AJR	adjectivizer
AVR	adverbizer
NEG	negative
PEJ	pejorative
DEG	degree
ORI	orientation
LOC	location
NUM	number

REV	reversive
ORD	time and order
NEO	neo-classical

Figure 4.4 List of All Gloss Tags Defined in ENGLEX

The raw output of PC-KIMMO is not directly useful for us. This immediate output passes through a simple transformation before they are used in subsequent processes. Part of the parser has been re-written to give a formatted result for other components of ARMON. The algorithm will be discussed later in this chapter.

Cow, eat and grass in the sample are resolved as noun, verb and noun respectively.

N : cow
V : eat
N : grass

Algorithms will transform the part of speech information here into the semantic heads, namely agent, action and patient.

4.4 Storing Knowledge

For the parser to do its job, a knowledge model has to be formulated to store the real-world knowledge as well as the parser output. Two "AI programming languages", namely Prolog and Lisp, have been used in the implementations of many IR models, such as FERRET [Maul89] and MARIE [Guli92]. Due to several limitations, some of which have been mentioned in previous section (4.2), we gave up the use of Lisp and Prolog as the implementation tool.

We then look for alternative implementation tools for knowledge storage. We have to design another way to keep the real world knowledge as a particular information model. After a number of iterations of drafts and refinements, we decide to store all the knowledge as tables for the ease of integration of several

modules. Tree-like structures are transformed into relational-like databases. In later sections we shall explain how this is accomplished. It is finally demonstrated to be a good model for our implementation of this knowledge model. Further, a database tool kit, Metalbase [Jern92], was found on FTP servers in the Internet. As it was also written in C-language, it was straight forward to interface this "database engine" with the selected NL parser, PC-KIMMO, the details of which will be discussed in the next section.

Models of data structures have to be designed to store the captions themselves and other associate "knowledge" output from parsing. Several of the major knowledge components are given as follows :

1. Hierarchies of real world entities, including noun-type and verb-type
2. Implicit features of the above entities
3. Captions
4. Explicit features of semantic groups in captions
5. Transformation maps which relate attributes in lexical form with logical explicit features

Generally speaking, the first two are static knowledge while the last three are dynamic knowledge by nature. In the implementation, captions are stored in a simple relational(-like) database called Metalbase which will be discussed later. Other knowledge are kept as plain text files in the Unix file system.

Metalbase [Jern92] is a toolbox suitable for development here. It is a freeware found in the Internet, is written in C language, and is compatible with the various C compilers running in the Unix systems, such as Linux, Solaris and HPUX, after little revision.

4.4.1 Type Hierarchy

The type hierarchies, as described in previous chapter, are working as conceptual representations of real world entities, both noun-type and verb-type. For illustrations here, examples are mainly taken from noun-type entities. Recalling that there are conceptual links between entities in the hierarchies, links which establish the intra-hierarchy "spatial" relation of individual nodes are

usually implemented as pointers. In current table-like implementation of the type hierarchy, an alternative method is adopted. The actual links do not exist. Instead, another mechanism is built to identify the relative location of individual entities in the hierarchy. Actually it proves working well in our implementation of this model.

Each entity is represented as a record in the type hierarchy database, HIER. For each entity or node on the type hierarchy, there are two fields which are meaningful to the users, the node name (nodename) and node identity (nodid). Among these two items, the nodid has higher significance. This id is not randomly assigned; it is computed according to its relative "position" in the hierarchy.

4.4.1.1 Node Name

The node name is the first field of each node. It is simply an ASCII string which tells users which entity it is. There are two restrictions in creating this field. Each node name should be in a single string and limited within a particular length. If the full name is too long, acronym or abbreviations are preferred.

4.4.1.2 Node Identity

Each node is identified by an identity, called nodid, which is a string of numeric characters. The topmost entity is the root of a hierarchy. Each nodid embeds essential information about the node itself - the relative location of the node in the hierarchy. The length of the nodid tells the vertical level of the node. For instance, a nodid of "11131" (quotes excluded) represents the information that the node is located on the fifth level counted down from the root, its ancestors are "1113", "111", "11" and "1", and its siblings are "11132", "11133" and "11134", etc. Figure 4.5 shows the entities and their identities on the type hierarchy. It is now observed that it is easy to determine the relation between two particular nodes. The algorithm of computation will be illustrated soon.

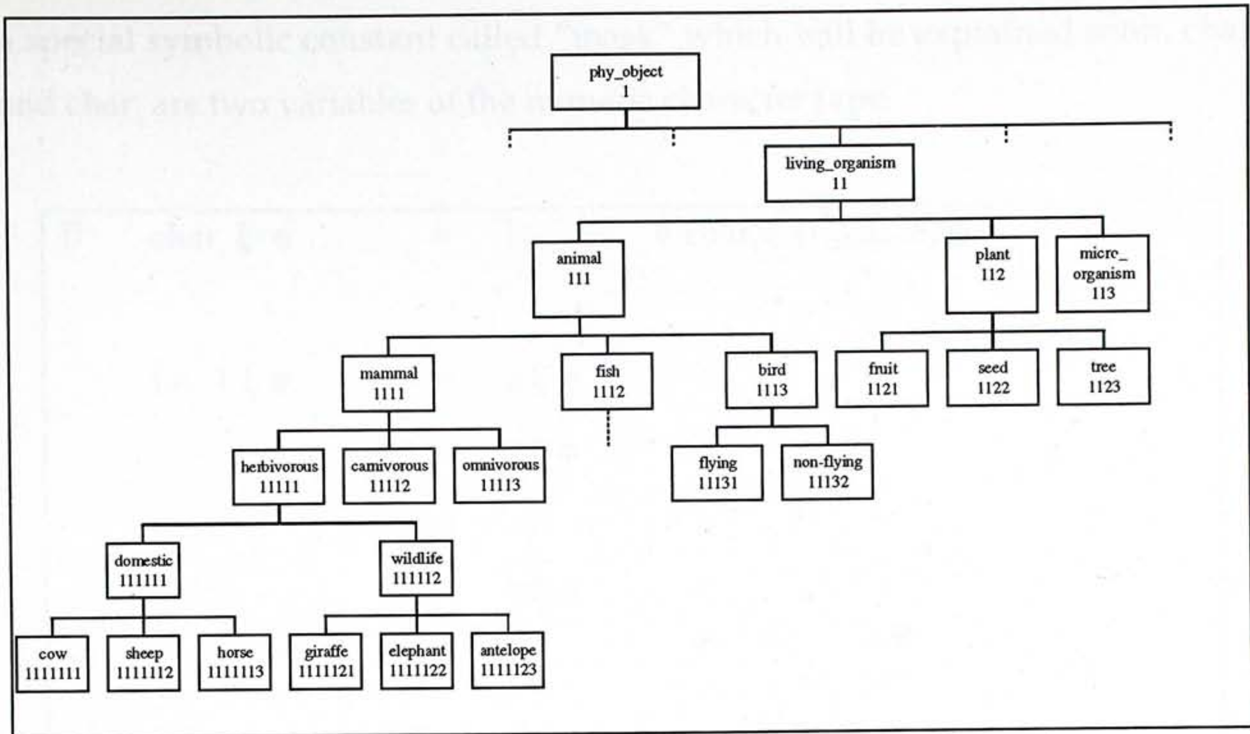


Figure 4.5 Entities with nodid on the Type Hierarchy

As mentioned before, each nodid is made up of a string of numeric characters. It means a string of arbitrary length containing the characters "1", "2", ... "9". There is a total of 9 possibilities for each characters. "0" is reserved and not used here. Remember that a particular node differs from its immediate sibling (if any) by the last character. There are at most 8 siblings for a particular node. In an alternative view, there should not be more than 9 immediate children for a single parent. It is one assumption of this model. This restriction, however, can be easily relaxed by allowing the nodid to take the alphanumeric values from "a" to "z". This extension allows a maximum of 34 siblings and 35 children for a particular node.

As a further relaxation, each level of the nodids can be represented by 2 numeric digits. In this case, the node plant with nodid 112 will be represented by a notation like (01, 01, 02). This extension allows a maximum of 99 siblings at each level. However, it is obvious that each nodid will not be as "brief" as the current representation. These extensions have not yet been implemented since the current model fulfills well our current knowledge domain.

A function is now defined to describe the relative distance between two nodes in the hierarchy. It is called *dist*(x,y) here, where x and y are two arbitrary nodes on the type hierarchy. Before we define the function *dist*(x,y), we firstly define a meta-operator ξ which is used to define the function *dist*(). In the following, φ is

a special symbolic constant called "mask" which will be explained soon, $char_1$ and $char_2$ are two variables of the numeric character type.

i) $char_1 \xi \varphi = 1 \quad \forall char_1 \in \{1,2,3,...8,9\}$

$$\text{i.e. } 1 \xi \varphi = 2 \xi \varphi = 3 \xi \varphi = ... = 9 \xi \varphi = 1$$

ii) $\varphi \xi char_1 = 0 \quad \forall char_1 \in \{1,2,3,...8,9\}$

$$\text{i.e. } \varphi \xi 1 = \varphi \xi 2 = \varphi \xi 3 = ... = \varphi \xi 9 = 0$$

iii) $char_1 \xi char_2 = 0 \quad \forall char_1, char_2$ where $char_1 = char_2$

$$\text{i.e. } 1 \xi 1 = 2 \xi 2 = 3 \xi 3 = ... = 9 \xi 9 = 0$$

iv) $char_1 \xi char_2 = 1 \quad \forall char_1, char_2$ where $char_1 \neq char_2$

$$\text{i.e. } 1 \xi 2 = 2 \xi 1 = 1 \xi 3 = 1 \xi 4$$

= ......

= 1

Figure 4.6 Definition of the Meta-operator ξ

Part (iii) and (iv) of the operator ξ is somewhat similar to the well-known operator exclusive-or, ⊕, on binary digits. The physical meaning and the usage of the defined operator ξ will be described immediately.

The function *dist*(x,y) is loosely defined as the distance of y from x. Assume now that x and y are the nodids of two distinct nodes on the type hierarchy. The steps of evaluating *dist*(x,y) is stated below.

i) get the nodids of x and y,

ii) fill the shorter nodid with the mask, φ, on the right to make them the same length,

iii) operate the corresponding digits of two nodids digit by digit from left to right.

e.g. 1

dist(1111, 1111111) = *dist*(1111φφφ, 1111111)

= 1111φφφ

ξ)1111111

0000000

= 0

e.g. 2

dist(1111111, 1111) = *dist*(1111111, 1111φφφ)

= 1111111

ξ)1111φφφ

0000111

= 111

v) the relative distance is obtained as the resultant string being viewed as a binary number, i.e. 0 or 111 in above examples.

A short form of the ξ operation has been exhibited in the above illustration. To explain in more details, the operation is expanded as follows.

```
  1111φφφ
ξ)1111111
  0000000
```

= (1ξ1), (1ξ1), (1ξ1), (1ξ1), (φξ1), (φξ1), (φξ1), (φξ1)

= 0, 0, 0, 0, 0, 0, 0

When the individual digits are concatenated together, we get 0000000, or simply 0 after the leading zeros are trimmed. The operation of ξ in the second example is similar and will not be expanded again.

We now see several examples showing how the meta-operator ξ and the function *dist*() work. With the definition of ξ and *dist*() in terms of ξ, the relative distance between two nodes on the hierarchy is demonstrated. In the example, 1111 is the nodid of mammal and 1111111 is the nodid of cow. It is shown that *dist*(mammal, cow) or *dist*(1111, 1111111) has a value of 0; *dist*(cow, mammal) or *dist*(1111111, 1111) has a value of 111. As expected, *dist*() is not communicative. It is true that a query which asks for a node on mammal will equally satisfy the captions about any sub-class of mammal, including cow. The opposite is not true. A query asking for cow will not necessarily satisfy on captions about its super-classes. This general property in IR is the origination of the formulation of the function *dist*(). More examples of the function *dist*() follows.

i)	*dist* (cow, sheep)	=	*dist* (1111111, 1111112)
		=	1
ii)	*dist* (cow, elephant)	=	*dist* (1111111, 1111122)
		=	11
iii)	*dist* (cow, domestic)	=	*dist* (1111111, 111111φ)
		=	1
iv)	*dist* (cow, wildlife)	=	*dist* (1111111, 111112φ)

		=	11
v)	*dist* (cow, mammal)	=	*dist* (1111111, 1111φφφ)
		=	1
vi)	*dist* (cow, fish)	=	*dist* (1111111, 1112φφφ)
		=	1111
vii)	*dist* (cow, plant)	=	*dist* (1111111, 112φφφφ)
		=	11111
viii)	*dist* (cow, herbivorous)	=	*dist* (1111111, 11111φφ)
		=	11
ix)	*dist* (herbivorous, cow)	=	*dist* (11111φφ, 1111111)
		=	0

Figure 4.7 Examples of some inter-entity distances

If each result of the function *dist*(x,y) is viewed as a binary number, it is observed from these examples, without rigorous proof, that the function *dist*(x,y) is coherently decreasing with the closeness between its arguments, x and y. It is a simple check of the closeness of two arguments. Observing from (iii) and (iv), one may conclude that cow is closer to domestic than to wildlife. This is clearly true in the real world under common sense.

It should be emphasized that the function *dist*(x,y) is not communicative, i.e., *dist*(x,y) ≠ *dist*(y,x) in general. The practical examples are (viii) and (ix). Case (ix) happens when a caption with cow is tried to be matched from query containing herbivorous. In contrast, case (viii) happens when a caption containing herbivorous is attempted to be matched with a query containing cow. In reality, when a query asks for an entity whose position is high in the hierarchy, e.g. herbivorous, it is equally acceptable if some captions contain more specific entity, e.g. cow is found. Hence, the function *dist*() should also reflect this similarity. However, the reverse is not true. i.e. when a query asks for a specific entity, e.g. cow, it is usually unsatisfactory to get a more general entity, e.g. herbivorous, from the caption. The result obtained from this algorithm, *dist*(cow, herbivorous) > *dist*(herbivorous, cow), reflects this assumption. The behaviour of *dist*() looks closer to a vector function than a scalar function.

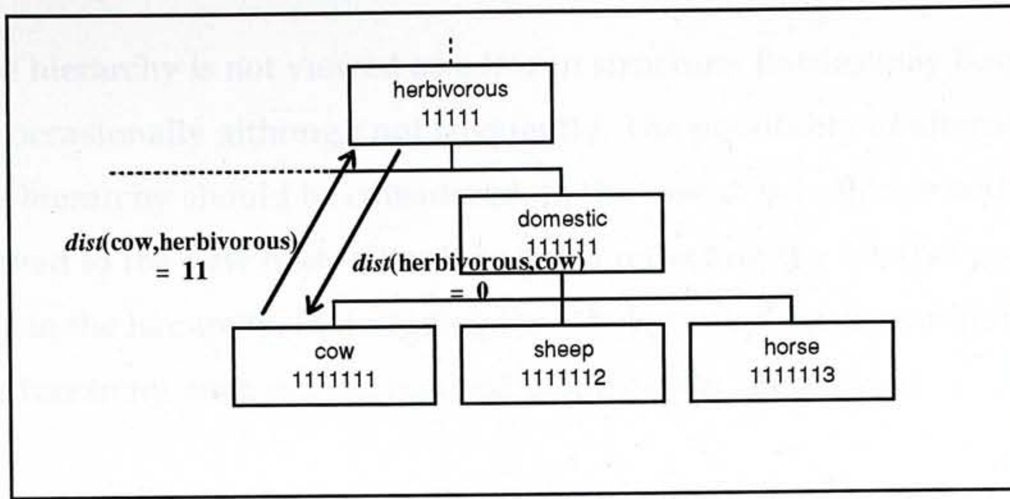


Figure 4.8 Vector Properties of the *dist*() Function

Another point is that all inter-sibling distances [1] are assumed to be the same, i.e. the values of *dist*(1111000,1111001), *dist*(1111000,1111002), *dist*(1111000,1111004), etc. are all equal. To determine the closeness among these entities, features of individual entities would solely be taken into account. That is the computation in the fine scorer. In this design, the way how the type hierarchy is drawn imposes no direct interference on the magnitude of the matching score. This is actually one of our intentions.

In implementation, the type hierarchy is also formatted as a text file under the Unix file system. It is currently located in the path $PRJHOME / data / hier.tab. Nodes are put in the file line by line, each of which contains the nodename and the nodid, as the figure shown below.

living_organism	11
animal	111
fish	1112
cow	1111111
sheep	1111112
elephant	1111122
wildlife	111112

Figure 4.9 Fragment of a Type Hierarchy

The rows need not be sorted; they are inserted in arbitrary order. In the above fragment, the entity living_organism has a nodid of 11, the entity animal has a nodid of 111, etc.

[1] Distances between two entities under the same parent

4.4.1.3 Operations

The type hierarchy is not viewed as a frozen structure. Entries may be added or deleted occasionally although not frequently. The possibility of alternations to the type hierarchy should be considered. In the case of insertion, a nodid has to be assigned to the new node since a nodid is reflecting the relative position of the node in the hierarchy. In designing the whole system, basic manipulations of the type hierarchy, such as insertion and modification, are allowed.

The type hierarchy should allow at least several basic operations, including insertion, deletion and printing (listing). Other modifications can be treated as a combination of these basic operations. Since the type hierarchy is formatted in a text file, the hierarchy file can be edited directly with an editor, such as `vi`, for operations like insertion, deletion or substitution. Another way will make use of a small utility named `hierop` which is included as a part of the entire ARMON system. This utility was written to operate the type hierarchy in a more user-friendly interactive manner.

4.4.1.3.1 Direct Edit

This method is straight forward. The file holding the type hierarchy is recalled and edited by any common editor such as `vi` or `pico` on Unix. With this method, the hierarchy can be modified instantaneously. This advantage is notable when there are a lot of modifications on one hierarchy. All modifications can be completed in a single edit. On the other hand, the shortcomings are also obvious. The user has to manually check the syntax and the consistence of the table. For example, it is the responsibility of the user to make sure that there is no duplicate nodid on one type hierarchy.

4.4.1.3.2 Interactive Edit

With this method, the hierarchy is modified by a small self-developed software utility. This utility accepts operations and operands from the users, and asks the user for any unknown detail. It eventually locates a "hole" for the new entity. In this system, the interactive editing features were written as a utility named hierop which is located in $PRJHOME/bin.

Let's consider that an entity reptile is going to be inserted interactively by an user. This will be accomplished by the command *hierop i reptile*.

The utility *hierop* communicates with the user in a typical dialogue as shown below in Figure 4.10.

computer	:	is reptile a member of living_organism ?
user	:	y
computer	:	is reptile a member of animal ?
user	:	y
computer	:	is reptile a member of mammal ?
user	:	n
computer	:	is reptile a member of fish ?
user	:	n
computer	:	is reptile a member of bird ?
user	:	n
computer	:	immediate parent of "reptile" is "animal", id:111
		Entity animal has 3 immediate child(ren) :
		1111 mammal
		1112 fish
		1113 bird
		new entity reptile should have a new id of the form
		"111_"
		please enter the last digit of the new id
user	:	2
computer	:	invalid, please enter another digit :
user	:	4
computer	:	the id of reptile is 1114

Figure 4.10 Adding an Entity under Interactive Mode

This is a simple dialogue between the computer and the user during the insertion of a new node. Having gotten the new node information, the system will show the occupied nodids to the user. The user should enter a nodid which is not yet occupied and the system rejects any violating nodid. The nodid

assignment is not yet fully automatic because of the possible existence of "holes" and our intention of keeping the algorithm of `hierop` simple.

In this data model, adding a node on the type hierarchy is just a simple insertion of a record on the database. A new nodid is assigned to the new record. In this example, reptile is a member of living_organism but none of others, therefore the node reptile must be an immediate child of living_organism. Its nodid must be "111?".

living_organism	11
animal	111
fish	1112
reptile	1114

Figure 4.11 Hierarchy Table Inserted with Node Reptile

The nodids operations are communicated to the users through a human-friendly interface. In this operation, `hierop` helps user search for a "hole" to place the new entity and tells the user that 1111, 1112 and 1113 have been occupied. With common sense, the user may choose any unoccupied nodid, 1114, 1115....., etc., and, say in this case, selects 1114 for the new entity reptile. The new fragment in the hierarchy is then adjusted as follows.

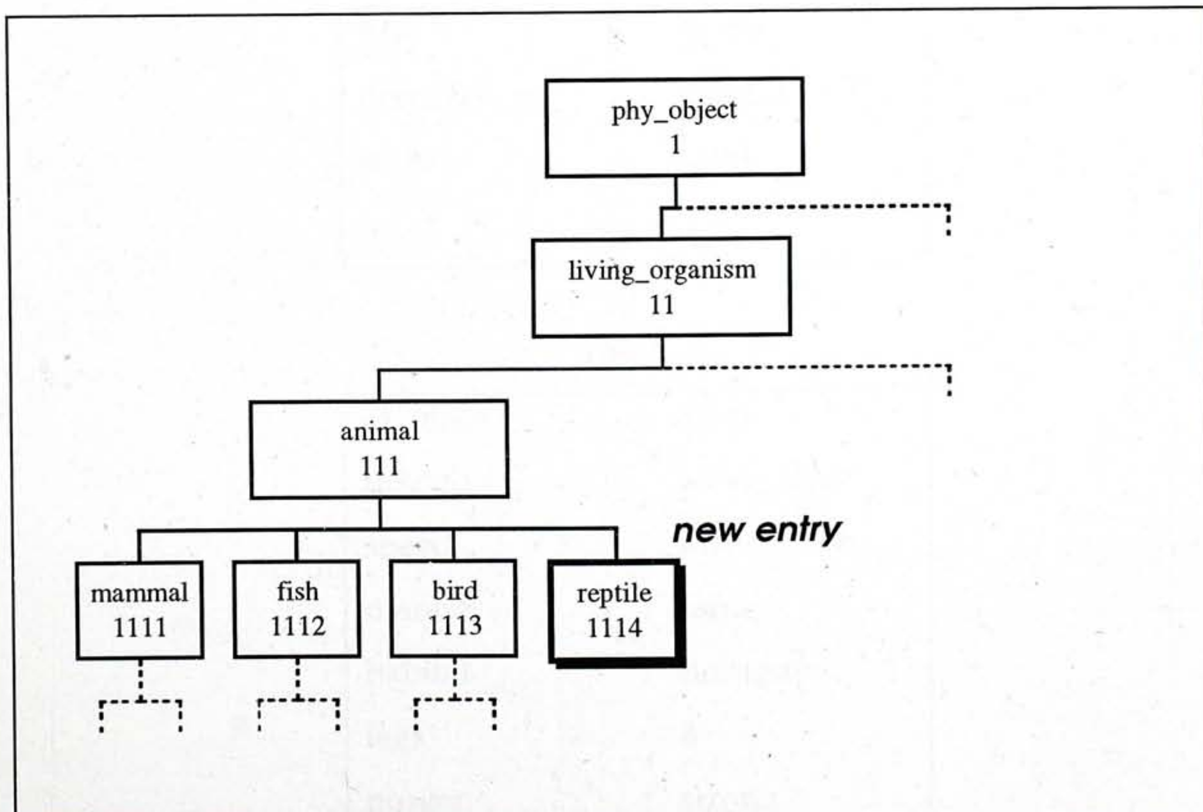


Figure 4.12 Hierarchy Fragment after Entity Inserted

Two more operations, deletion and print (display), can be done similarly by *hierop*. The following shows the examples of their usage.

```
hierop d reptile
hierop p mammal
```

The first operation attempts to remove the node reptile and its entire sub-tree. The second operation will print the sub-tree under the node mammal. More details of the usage of the command *hierop* will be described in the manual pages of Appendix E.

4.4.2 Implicit Features

As mentioned in the previous chapter, each entity has a set of default intrinsic features. This is the secondary level knowledge in addition to the primary level knowledge (type hierarchy). The intrinsic features are being stored in the text files under the hierarchical Unix file system. Each entry is stored as a file with a name equal to the entity itself. Each entity file contains at least two fields: the implicit feature slot and the implicit feature value. Two examples are shown below.

living_organism

life	living
occurrence	natural
state	solid

cow

at_place	farm
function	serve_man
speed	low
manner	tame
habitat	domestic
legs	4
power	strong

The above tables show the implicit features of the entities of living_organism and cow, respectively. They are found in the paths $PRJHOME / data / impfea / living_organism and $PRJHOME / data / impfea / living_organism / cow, respectively.

4.4.3 Database of Captions

There is another important data structure which stores the parsed captions in logical form. A fragment of the caption database is shown in the form of a table. As explained before, AGENT-ACTION-PATIENT gives rise to the primary semantic structure of an ordinary sentence. Each caption is stored as a basic unit of aggregation in this manner.

As this data structure can be large in size compared to other data structure already mentioned, it is not stored as plain text file. Instead it is stored in a simple relational-like database toolkit. We actually tried to look into several possible solutions and finally found MetalBase [Jern92] in the Internet. It was available as freeware together with source codes and documentation. It was thus acquired for integration into our application at the source code level.

It is anticipated that a relational database exhibits higher efficiency than a plain ASCII file in retrieval when the size of data is growing large. The CAPT database is prepared for any potential growth of the caption population in the system. MetalBase includes a proper indexing mechanism [Jern92] to provide better performance when CAPT grows large. However, the advantage of this structure has not been practically shown in this prototype because CAPT is not yet significantly large in our experiments.

A simple sample of CAPT database is shown here for illustration and will be used later in this chapter. Firstly it is displayed as an usual table-like form directly related to the relational-like database. The database file itself is stored as $PRJHOME / data / capt / capt.rel in the hierarchy of the Unix file system.

capid	agent	action	patient
1232	cow	eat	grass
2678	horse	drink	water
4511	sheep	eat	grass
7622	monkey	walk	

Figure 4.13 Fragment of the Caption Database

As mentioned in the previous chapter, a short form is used to represent the sentences in captions and queries in a more convenient manner. In this form, sentences in Figure 4.13 are written as 1232 {cow[], eat[], grass[]}, 2678 {horse[], drink[], water[]}, 4511 {sheep[], eat[], grass[]} and 7622 {monkey[], walk[], - }, respectively. The explicit features are then simply filled in the square brackets when they are extracted from the original NL sentence. These handling of explicit and implicit features will be discussed in the coming sections.

4.4.4 Explicit Features

The set of explicit features is another important data structure. As mentioned in the previous chapter, the explicit features are usually collected from the descriptive text in the sentences. They are usually adjectives or adverbs which are collectively called the attributes of the semantic groups agent, action or patient. The attributes will undergo a transformation to give the corresponding explicit features. The rules of transformation will be discussed in the next section.

Consider again the example "the large cow is quickly eating the green grass". The attributes are *large* for cow, *quickly* for eat and *green* for grass. We label these attributes attrag, attrac and attrpa respectively. Applying the transformation map discussed later in next section, the explicit features for the entities are obtained :

AJ:large → *size:large*
AV:quickly → *speed:high*
AJ:green → *colour:green*

In this implementation, we take advantage of hierarchical Unix file system to store the explicit features of each caption, as shown in Figure 4.14.

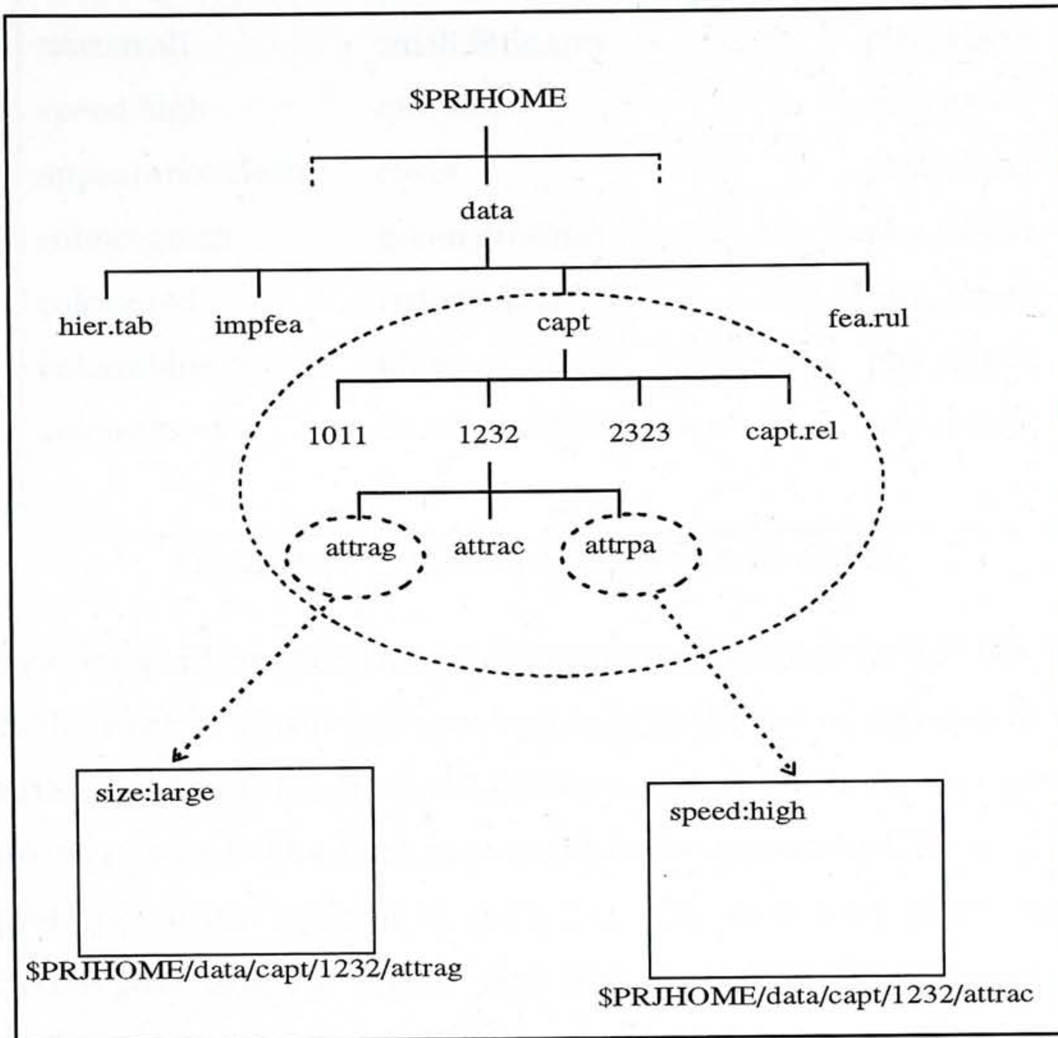


Figure 4.14 Explicit Features Stored in a Unix File System

In this sense, the explicit features of the agent group would be stored in $PRJHOME / data / expfea / 1232 / attrag. The explicit features of the action group would be stored in $PRJHOME / data / expfea / 1232 / attrac and the explicit features of the patient group would be stored in $PRJHOME / data / expfea / 1232 / attrpa. Figure 4.14 just shows the several items directly related to the feature set. Other major components constructed in the file system can be found in Appendix F.

4.4.5 Transformation Map

Transformation map is another ASCII file which holds the mapping needed for transforming attributes into explicit features. It is named as fea.rul and kept in

the path $PRJHOME / data / fea.rul. An example of its file contents is shown as follows :

size:large	great,large,big,huge,giant	phy_object
size:small	small,little,tiny	phy_object
speed:high	quick,fast	activity
appearance:clean	clean	phy_object
colour:green	green,greenish	phy_object
colour:red	red,reddish	phy_object
colour:blue	blue	phy_object
colour:brown	brown,reddish-brown	phy_object

Figure 4.15 A Sample Transformation Map

Each row or record contains three fields separated by TABs (ASCII 09). The first field is the explicit feature, the second field is the list of equivalent written attributes and the third field is the higher-most entity, under which the transform is valid. The first row is taken as an example to explain the interpretation of this table. If an entity is a sub-class of phy_object and it was depicted as *great, large, big, huge* or *giant*. It is equivalent to having an explicit feature of *size:large*.

In symbolic terms, the relationship is represented as follows :

$$has_attr\ (X, ATTR) \rightarrow has_efea\ (X, size{:}large)$$

$$\forall ATTR \in \{ great, large, big, huge, giant \} \cap isa\ (X, phy_object)$$

Following this rule, we may map the token "*large*" in "*large* cow" into the explicit feature "*size:large*" because :

cow is a phy_object

AND

large ∈ { *great, large, big, huge, giant* }

In this sense,

has_attr (COW, *large*) → *has_efea* (COW, *size:large*)

With the same mapping rule, other expfeas of **"a large cow is quickly eating green grass"** are obtained. The results are then kept in the data structure mentioned in the last section.

In this chapter we have presented the details on how the main data structures are implemented. In the next chapter, we shall discuss the parsing and matching mechanisms based on these data structures.

CHAPTER 5

ILLUSTRATION

That is the best part of beauty, which a picture cannot express.

Francis Bacon (1561-1626)
Essays (1625) Of Beauty

In ARMON, parsing is the process which transforms natural language descriptive text into logical representation stored in the back-end data structures. For information retrieval purpose, some grammatical details can be and will be ignored without significant loss of meanings of the whole sentence. It is a basic assumption of our model and has been explained in Chapter 2. For instance, determiner "a", auxiliary "to be" and the progressive tag, PRG, will all be ignored in this stage. Some extra information, e.g. tense, etc., are dropped in this stage. "Eating" in the example in Figure 2.5 is given in a continuos tense. During parsing, it passes through the decomposition into gloss tags V and PRG.

eating → eat + ing → V + PRG

The parsing result tells that "eating" is a verb (V) in progressive state (PRG). V is a major information for the token "eating" and PRG is just a subsidiary information of the token "eating". With ARMON's assumption of ignoring subsidiary information, only the key information of a token is kept, i.e. only the tag V is meaningful here. Ignoring supplementary details as shown in this example is sometimes called **text skimming** [Lewi89].

5.1 Tagged Items

In the simplest examples, there are verbs and nouns in a sentence. In most cases, there are often adjectives and adverbs. The adjectives and adverbs are always attributes of noun-type and verb-type entities, respectively. The PC-KIMMO

parses any sentence word by word into a list of **gloss tag**[1]. The possible syntactic classes of each token will be given as the parsing result.

The sample sentence "the large cow is quickly eating the green grass" is parsed by the PC-KIMMO as follows:

the		
	the	DT
large		
	`large	AJ
cow		
	`cow	N
is		
	be	AUX.3SG
quickly		
	`quick+ly	AJ/AV+AVR1
eating		
	`eat+ing	V+PRG
the		
	the	DT
green		
	`green	AJ.NR0
	`green	AJ
grass		
	`grass	N

Figure 5.1 The Raw Output of the PC-KIMMO Recognizer

The above figure shows a table of raw output of the PC-KIMMO. The raw output does not provide any direct information of semantic groups that is required in our model. Some processing will soon be addressed to cover the necessary transformation of semantic details as needed.

[1] Please refer to Figure 4.4 for a complete list of gloss tags and their symbols

The symbols which are shown in the above presentation have been widely presented in previous section without much explanation. Perhaps it is a suitable place to give some details here. These tags were defined by the lexicon, ENGLEX, [Antw92b] and re-tabulated in Figure 4.4 in the previous chapter. The raw output of PC-KIMMO is simple. It is a list of records, each of which contains the token itself, the decomposed morphemes and the related gloss tags. Gloss tags are abbreviations for the morpheme components in tokens. The concepts are fundamental to understand further details of the model, we shall explain the meaning of this tag here.

The implications of DT of "**the**", AJ of "**large**" and N of "**cow**", are straightforward because there is only one grammatical interpretation for each token. They represent Determiner, Adjective and Noun respectively. On the other hand, there may be tokens which have multiple tags. For example, "**quick**" is tagged with "AJ/AV" because "**quick**" can be either an adjective or an adverb as defined in ENGLEX.

AVR stands for adverbizer which converts a word into a noun. "**Quickly** " here is recognized as "'**quick+ly**". It is tagged with the notation "AJ/AV + AVR1". It means that the root "**quick**" can be either an adjective or adverb. However, the suffix tag AVR means that the suffix "**ly**" converts the POS of "**quick**" into adverb. We now call "quick" adverb-equivalent. The purpose is to distinguish the recognized part-of-speech from the written form of the token. The "1" in AVR1 is simply an index of "AVR" for distinguishing one adverbizer from other adverbizers, with labels like AVR2, AVR3, etc. Those commonly used adverbizers have been completely defined in ENGLEX [Antw92b].

AJ·NR0 for "**green**" here means that the root of "**green**" is **green** itself and it can be nominalized into noun without any suffix (or regarded as null suffix). V+PRG for "**eating**" have similar meaning. The "**ing**" suffix to "**eat**" gives the root "**eat**" an additional progressive (PRG) meaning in addition to the default properties of "**eat**".

Adverbizer (AVR) and nominalizer (NR) have been explained so far. There are two more important gloss tags defined by ENGLEX, namely the verbalizer (VR)

and adjectivizer (AJR). We collectively call them POS converters. VR and AJR have similar meaning as AVR and NR, they make words verb-equivalent and adjective-equivalent, respectively. We do not repeat the explanation here.

In addition to these gloss tags, there are totally 40 gloss tags defined, as shown in Figure 4.4. There are too many gloss tags to be fully explained here; we have omitted much more useful details. The on-line manual of ENGLEX [Antw92b] provides excellent explanations and illustrations of many single and combined gloss tags. Interested readers may obtain more background information from this manual.

5.2 Parsing

In ARMON, parsing has the meaning of the transformation of lexical forms [1] into logical representation [2]. All the ambiguity will be solved in this stage. In this section, the parsing mechanism will be illustrated through examples.

5.2.1 Resolving Nouns and Verbs

In this section, examples are used to illustrate how to resolve the noun-type and verb-type tokens into the corresponding semantic groups. Since PC-KIMMO is not a semantic parser, it does not automatically distinguish individual semantic heads, namely agent, action and patient which are needed in the retrieval system. Some rules have to be formulated to distinguish these semantic groups. In this section and the next few sections, tables are used to illustrate the stages in resolving NL sentences into corresponding logical forms and are structurally written as arrays and linked lists in our program.

Consider again the sample sentence, "the large cow is quickly eating the green grass", the nouns and verbs are extracted as follows :

---

[1] As summarized in Figure 2.4
[2] As summarized in Figure 2.2

Token	Gloss Tag	Meaning
cow	N	Noun
eat	V	Verb
grass	N	Noun

Figure 5.2 Output of Nouns and Verbs from Parsing

As mentioned in previous chapter, the natural language used to write caption sentences is based on a restricted subset of common natural language. These properties of such a subset of natural language help a lot in resolving semantic ambiguity. In our grammar, passive voice is avoided and hence, the agent always precedes the patient in a sentence according to the agent-action-patient sequence.

The grammar of ARMON defines that nouns and verbs occur in the sequence N-V-N in a sentence. If there is absolute certainty on one (or two) entity (entities) in the sentence, i.e.

N - ? - N
N - V - ?
? - V - N
? - V - ?
N - ?
? - V
:
etc.,

the possibility of ambiguous forms will be greatly reduced or even totally eliminated. Lower level details, such as the second verb and the third noun will be treated with lower significance. They can be transformed into explicit features or simply ignored without critical effects.

The simplest case is that only one (or two) noun(s) occurs (occur) in a sentence. The first noun is assumed to be the agent and the latter the patient. A more complicated situation happens when there are more than two nouns in a sentence. The parser will assume that the first noun to be the agent and the

second one the patient. For nouns beyond the second one, ARMON will simply ignore them or treat them as subsidiary information.

Similarly, if there is only one verb in the sentence, it is simply that the verb is the action of the sentence and thus fills in the corresponding semantic group. Similarly, any "extra" verb will be ignored or processed as subsidiary information of the sentence. The verb eat in the above example is determined as the action of the sentence. Consider again the sample sentence "the large cow is quickly eating the green grass", the analysis is briefly stated as follows :

- The first noun is cow, which is identified as the agent.
- The second noun is grass, which is determined to be the patient.
- The first and only verb is eat, which is determined to be the action in the sentence.

The effective mapping between tokens in sentence and the semantic groups therefore looks like Figure 5.3.

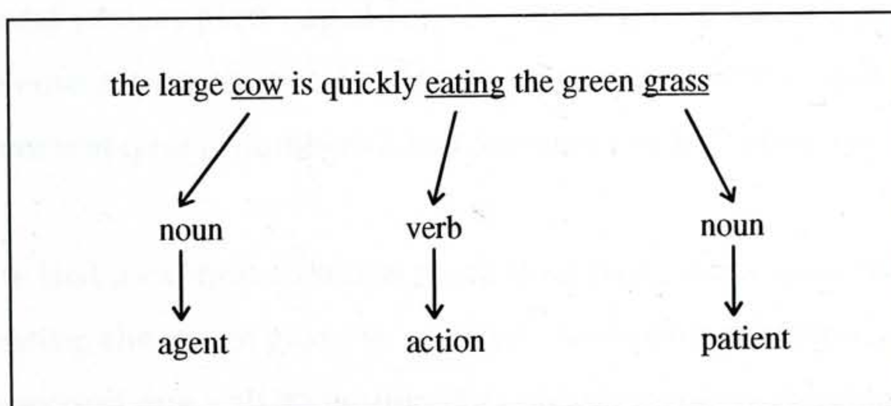


Figure 5.3 Mapping Nouns and Verbs

With the above simple algorithm, the lexical tokens, cow, eat and grass are designated to the semantic groups agent, action and patient respectively.

Recalling that there is a database CAPT which is implemented with Metalbase. The resolved agent, action and patient are now entered in the corresponding fields in the caption database CAPT.

CAPT

capid	agent	action	patient
?	cow	eat	grass

Figure 5.4 Agent, Action and Patient are Filled as a Record in CAPT

During parsing, each caption is assigned an unique identifier, capid, which is also a useful reference to this caption anywhere in the system. It is emphasized again that capid and nodid are distinct items, and their roles should not be confused.

In ARMON, a capid can be specified by the user for a particular NL caption. In this case, the system will add the NL caption as a new one if the capid is not existing. If the given capid is existing, the new caption will replace the existing one with the same capid. Please refer to the relevant Manual Pages in the Appendix E for further details.

When there is no preferred capid given by the operator, ARMON will generate an unique and unoccupied capid for the new caption to be inserted. In the current implementation, the valid range of capid is 10-19999. Capid 1 is used for identifying current query, numbers 2 to 9 are reserved for future expansion.

Assume now that a caption contains more than two nouns as in "the large cow is quickly eating the green grass in a farm". As in our grammar, other nouns beyond the second one will be assumed as supplementary descriptions which are less important in the whole sentence. In this example, the third noun is farm, which is then transformed into an explicit feature *place_at:farm*. This kind of transformation will be discussed in the next section. At the current stage, other non-essential tokens, e.g. DET, AUX....., etc., will be ignored.

5.2.2 Resolving Adjectives and Adverbs

In this section, adjectives and adverbs are analyzed to find out that which semantic groups they belong to. In the previous section, it has been explained

how the parser processes the simplest sentence into two nouns, cow and grass, and one verb, eat. Other classes of tokens, adjectives and adverbs, are going to be illustrated now. Firstly, the parse results of the tokens *large*, *quickly* and *green* are summarized in Figure 5.5.

Token	Tag	Meaning
large	AJ	Adjective
quickly	AJ/AV+AVR1	Adverb-equivalent
green	AJ · NR0	Noun-equivalent
	AJ	Adjective

Figure 5.5 Output of Adjectives and Adverbs from Parsing

It is observed that each of the tokens **large** and **quickly** is given one meaning only. They are the attributes of agent and action, named **attrag** and **attrac**, respectively. There is no ambiguity to distinguish the POS of these two tokens. They are adjective and adverb-equivalent, respectively. In contrast, the token **green** is given two possible POS, noun and adjective. Some heuristics will soon be given to resolve ambiguities like this.

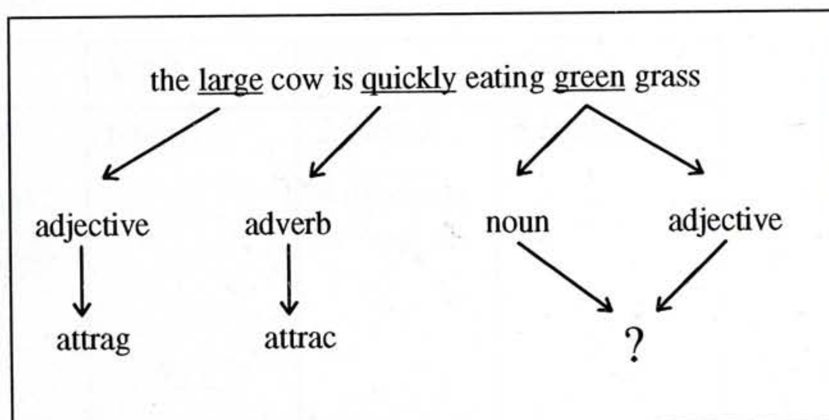


Figure 5.6 Ambiguity during Mapping Adjectives and Adverbs

In the following paragraphs, part of the resolving mechanism will be shown with the above example. As previously mentioned, tables will be used for a clearer explanation. At first, a label is created for each lexical token used for lexical reference. Arbitrarily, the labels are taken as the sequence of occurrence of the token in the sentence. They are tabulated as shown below.

Sentence Table

Label	Lexical Token
1	the

2	large
3	cow
4	is
5	quickly
6	eating
7	the
8	green
9	grass

A POS table is used as working storage for the possible POS of relevant tokens. The POS table consists of 4 columns, namely noun, verb, adjective and adverb. In each column, token labels are filled as to indicate possible POS of the tokens belonging to. In this table, each lexical token can be filled as more than one instance. In theory, each token can have one and only one POS. The incorrect POS of each token (if any) will be eliminated by an algorithm to be described soon.

POS Table

Noun	Verb	Adj	Adv
3 (cow)	6 (eat)	2 (large)	5 (quickly)
9 (grass)		8 (green)	
8 (green)			

Recall that the relevant production rules from our grammar of the restricted language set in Figure 2.4.

$$NP = DET \cdot AJP \cdot N \mid AJP \cdot N \mid N$$

$$AJP = AJ \cdot AJP \mid AJ$$

As an example , noun phrase (NP) can be expanded as the following pattern :

$$NP = \{DET\} \cdot AJ_1 \cdot AJ_2 \cdot \ldots \cdot AJ_k \cdot N$$

In this sense, adjectives always precede nouns in a particular phrase. In this particular example, if "**green**" is assumed a POS of adjective, "**green grass**" with the sequence AJ-N which obeys the ARMON grammar. But it is not true if "**green**" assumes another POS, noun, because N-N violates our grammar. As a result, the second possibility should be eliminated and only the first remains.

The POS table looks like the follows after the token "**green**" after noun has been eliminated.

Noun	Verb	Adj	Adv
3 (cow) 9 (grass)	6 (eat)	2 (large) 8 (green)	5 (quickly)

The resolved meanings are then written into the sentence table.

Label	Lexical Token	POS
1	the	-
2	large	AJ
3	cow	N
4	is	-
5	quickly	AV
6	eating	V
7	the	-
8	green	AJ
9	grass	N

Now there are two nouns and one verb as a result of resolving the ambiguity. According to the rules of resolving nouns and verbs addressed in previous section, it is easy to place each noun and verb in the positions of agent, action and patient.

Agent	Action	Patient	attrag	attrac	attrpa
3 (cow)	6 (eat)	9 (grass)	?	?	?

Adjectives and adverbs have been extracted by the parser. If there are only one noun and one verb in a caption, the adjectives and the adverbs must "belong" to the noun and the verb, respectively. If there are more than one noun and one verb, predefined rules will be referred. In ARMON, the patterns AJ-N and AV-V are valid in its grammar. Some other patterns, say AJ-V and AV-N, etc., are invalid. Complete lexical grammar can be referred to Figure 2.4 as in the previous chapter.

With the collected information, we can now deduce that "green" is likely an attribute of the patient, **attrpa**. It is written as the following table in short.

Agent	Action	Patient	attrag	attrac	attrpa
3 (cow)	6 (eat)	9 (grass)	2 (large)	5 (quickly)	8 (green)

According to the information obtained so far, the inter-relationship between individual lexical tokens becomes clear now.

Token	Semantic Role	Implied Relation
the	-	
large	attrag	⤵
cow	ag	←
is	-	
quickly	attrac	⤵
eating	ac	←
the	-	
green	attrpa	⤵
grass	pa	←

Figure 5.7 The Relations among Lexical Tokens

However, the attributes obtained here do not yet reach the requirements of our knowledge model. As the logical model of sentences shows, we have to map all attributes, including attrag, attrac or attrpa, into feature slot and value pairs. The transformation will be addressed in the next section but the result is shown here first.

attrag:large	→	size:large
attrac:quickly	→	speed:high
attrpa:green	→	colour:green

Figure 5.8 Transformation of Attributes into Feature Slots and Values

5.2.3 Normalizing Features

At this stage, attributes are mapped into pairs of feature slot and value. In addition, different attributes of similar meaning are "normalized" into a single expfea in the data structure. For example, consider the attributes *quickly, fast* for the action eat. The root forms of these attributes are firstly obtained, i.e. *quick* and *fast*, respectively.

Recalling the file fea.rul in Figure 4.15, the line about *quick* and *fast* is marked with ① as follows. (Figure 5.9)

size:large	great,large,big,huge,giant	phy_object
size:small	small,little,tiny	phy_object
speed:high	quick,fast	activity
speed:high	hungrily	eat
appearance:clean	clean	phy_object
colour:green	green,greenish	phy_object
colour:red	red,reddish	phy_object
colour:blue	blue	phy_object
colour:brown	brown,reddish-brown	phy_object
colour:yellow	yellow,golden	phy_object

① (arrow pointing to the size:small line) ② (arrow pointing to the appearance:clean line)

Figure 5.9 The Transformation Map for Normalization

The enclosed line ① implies mapping both "quick" and "fast" into one expfea *speed:high* for all sub-actions under activity, such as run.

AV:quickly → *speed:high*

AV:fast → *speed:high*

On the other hand, the mapping of **hungrily** is only valid for the entity eat.

i.e. AV: hungrily → *speed:high* $\forall$ action $\in$ eat

In other words, the meaning of "hungrily" in "the tiger is running hungrily" is unknown to the system. For instance, it may be running very fast to catch a rabbit as dinner, or it may be moving slowly because it has no energy at all. There may be other possible inferences, as well. It is not known because the inference has not been defined in the knowledge of ARMON. It is therefore important to define scope of validity of mapping rules for attributes.

Similarly, the adjectives for cow, AJ:great, AJ:large, AJ:big, AJ:huge and AJ:giant are also transformed into *size:large*. ($\because$ cow $\in$ phy_object)

AJ:great → *size:large*

AJ:large → *size:large*

AJ:big → *size:large*

AJ:huge → *size:large*

AJ:giant → *size:large*

With this mapping, attributes (adjectives and adverbs) are mapped to features for particular scope of entities.

5.2.4 Resolving Prepositions

Prepositions play a distinct role in natural language sentences. Some prepositions in a sentence are essential to the idea of the whole sentence. In contrast, some prepositions are merely metaphors which carry little significance in the sentence. Perhaps proposition gets the highest variety of patterns and

meanings. For example, "by" in the phrase "standing by the tree" has a meaning of position "beside". "By" in the phrase "touch by hand" means the use of an instrument (of hand). "By next week" has an implication of time sequence. The parser PC-KIMMO is able to distinguish the grammatical part-of-speech of "by" but semantic resolution of "by" is beyond its capability.

Considering the phrase, "standing by the tree", the raw output of the PC-KIMMO gives "by" in the above phrase possible two possible POSs, preposition (PP) or adverb (AV).

```
standing
                `standing        V(`stand)+NR24
standing
                `stand+ing       V+PRG
by
                by               PP/AV
the
                the              DT
tree
                `tree            N
```

Referring to the ARMON's grammar in Figure 2.4, there is no rule for AV·NP but the rule making up PP·NP exists. It can be deduced that "by" is a preposition (PP). The story has not yet ended. Its semantic meaning is still unknown to us. Further heuristic has to be found to solve "by" more thoroughly.

Remember that the primary information in a natural sentence is the focus of processing. As an assumption of ARMON, low level details bear less degree of importance than primary details. Preposition is such a low level detail, loss of which does not destroy the essence of the whole sentence.

Fortunately, prepositions are commonly bounded within a finite set of meanings. This assumption reduces the probability of error occurrence during resolving prepositions. In the interactive mode of operation, the system can take the advantage of consulting the on-line user in case of non-solvable prepositional phrase.

General purpose positions can be broadly grouped into limited finite classes. In this example, "by the tree" is a preposition phrase, PPP. The set of common meanings of "by" can be found in an common dictionary. In the CD-ROM version of the Random House Unabridged Dictionary, 2nd Edition [Rand93], a total of 24 meanings for the preposition "by" is defined. Some close meanings are grouped and they are finally reduced into 10 groups.

Assume now that the system is operated in the interactive mode. The user will be asked what the actual meaning of "by" is. A sample dialogue is shown as follows.

computer	:	is "by" a preposition ?
user	:	y
computer	:	select the closest meaning of the preposition "by". 1. location 2. time 3. tool 4. rule 5. reason/consequence 6. medium x. irrelevant
user	:	1
computer	:	select the closest meaning of the preposition "by" 1. above 2. below 3. inside

4. outside

5. near to

6. far away

7. central

8. surround

9. approaching

10. separating

user : 5

computer : done

Figure 5.10 Resolving Preposition

Figure 5.10 illustrates how the preposition "by" is being solved. The query user is firstly asked to give a rough meaning. After "position" has been selected by the user, the position-related sub-meanings are listed for the user to narrow down. The user chooses "near to" from the list. ARMON eventually infers a feature of *nearby:tree* for the preposition "by".

Preposition is one of the most challenging items for natural language parsers. Like many parsers which do not give adequate resolution for prepositions, the unattended mode of ARMON system does not provide a good result in this matter either. To certain extent, the interactive mode of this system is a way to overcome this difficulty. A better automatic parsing mechanism for preposition is needed in future enhancement.

5.3 Matching

The matching process involves gross filtering which searches the semantic groups roles in the query from the caption database. Fine scores are then calculated from the explicit and implicit features in the query and the captions obtained in the gross matching. For simplicity but without loss of generality, it is assumed here that the sample caption database shown in Figure 4.13 is used here for illustration of matching. It is repeated here in the short form.

1232 {cow[], eat[], grass[]}

2678 {horse[], drink[], water[]}

4511 {sheep[], eat[], grass[]}

7622 {monkey[], walk[], - }

5.3.1 Gross Filtering

As its name implies, gross filtering extracts the roughly matched captions from the caption database. It passes the "filtrate" to the next stage for fine scoring. For better understanding and comprehension, the steps of gross filtering are illustrated with several examples instead of straight narration.

Case 1 -

Consider now the query, "a horse drinks water".

i) The query is first parsed into the semantic heads, namely agent, action and patient which are horse, drink, and water respectively. The query is now represented in its short form :

{horse[], drink[], water[]}

Since there is no explicit feature being stated in the query, the positions for the explicit features, i.e. the space within the square brackets, are left blank.

ii) Each role is then searched in the caption database. In more details, agent horse in query is searched from the agent fields in the caption database. Action eat in the query is searched from the action fields in caption database. Patient grass in the query is searched against the patient fields in the caption database.

Obviously, all the three roles can be matched in the caption database at the caption with capid 2678. The matching score is computed as in the previously defined formula in Chapter 2.

The matching score is simply 1. This is an example of exact match. The next example will show how an inexact match is processed. The gross filter has to take more effort to get it done.

Case 2 -

Consider another query, "a horse is eating grass".

i) As before, the query is firstly parsed into the agent, action and patient roles which are horse, eat, and grass respectively, i.e. {horse[], eat[], grass[]}.

ii) For the role action, exact matching is found in captions 1232 { cow[], eat[], grass[] } and 4511 { sheep[], eat[], grass[] }. For the role agent, exact matching is found in caption 2678 { horse[], drink[], water[] }. For the role patient, exact matching is found, again, in caption 1232 { horse[], eat[], grass[] }. It is observed that there is no complete matching for the whole query, i.e. no caption in the CAPT database has all three thematic roles matching the query. Up to now, the closest caption is 1232 { cow[], eat[], grass[] } in which two semantic heads eat and grass match the query.

iii) Since the agent in the query cannot be exactly found in CAPT, the next step is to search for an agent which happens to be the closest to the agent in query { horse[], eat[], grass[] }. The knowledge stored in the type hierarchy is now recalled. Firstly, we check to see whether there is any node horse in the HIER database. There is one whose nodid is 1111113.

By masking the rightmost character, the nodid of its immediate ancestor is obtained, i.e. 111111 in this case. Note that it is not necessary to know the node name of that ancestor.

The resultant string (i.e. the nodid of its ancestor, 111111) will then be matched against the left portion of nodid in the HIER database. Several nodes are matched on this level of approximation, i.e. cow

(1111111), sheep (1111112) and horse itself (1111113). The entities cow, sheep and horse are all in a sibling relationship. The searching of siblings on the same level as horse is hence illustrated. These steps are written in the program called the sibling searching routine.

Next the process tries to find these nodes in the existing CAPT database. Now horse, sheep and cow are matched in the captions 2678{horse[ ], drink[ ], water[]}, 1232{cow[ ], eat[ ], grass[]} and 4511{sheep[], eat[], grass[]} respectively. With this method, the searching of the nodes in a tree is transformed into the searching of the entries in the database. Combining with the result obtained in the previous step, a gross result is obtained. Caption 2678 seems to be the closest caption in this example.

In the examples above, the process of gross filtering has been shown. One gives an exact match and the other gave an inexact match. All the results obtained here will be fed to the next step. Since the gross filter does not account for the attributes associated with the semantic heads, even an exact match in the above example will be passed to the fine scorer in next step to have more precise matching.

5.3.2 Fine Scoring

The next step following the gross filtering process is to determine the fine matching scores in the set of captions extracted so far. The implicit features of each entity are searched from IMPFEA and the explicit features are extracted from the EXPFEA in the path $PRJHOME / data / capt / <capid> / <sem_group>.

Taking cow as an example, all its local implicit features are collected in $PRJHOME / data / capt / impfea / cow.

place_at	farm
function	serve_man
speed	low
manner	tame
habitat	domestic
legs	4
power	strong

Figure 5.11 Local Implicit Features of Cow

In short, the set of local implicit features of cow is:

$impfea_{local}$(cow) = { *place_at:farm, function:serve_man, speed:low, manner:tame, habitat:domestic, legs:4, power:strong* }

Remember that cow also inherits impfeas from all its ancestors in addition to these implicit features defined at the node itself. As mentioned in the previous section, there is a simple way to find the ancestors of a particular node in this implementation. Recall that the nodid of cow is "1111111" and its immediate ancestor is domestic, "111111". From its data file $PRJHOME / data / capt / impfea / domestic, the set of its local implicit features is found.

$impfea_{local}$(domestic) = { *belongto:human* }

Other ancestors up the hierarchy are 11111, 1111, 111, 11 and 1. They stand for herbivorous, mammal, animal, living_organism and phy_object respectively but it is not necessary to know their node names if only their implicit features are to be found. With similar operations, the impfeas of these entities are simply retrieved. In summary, we have:

$impfea_{local}$(cow) = { *place_at:farm, function:serve_man, speed:low, manner:tame, habitat:domestic, legs:4, power:strong* }

$impfea_{local}$(domestic) = { *belongto:human* }

$impfea_{local}$ (herbivorous) = { *diet:plant* }

$impfea_{local}$ (mammal) = { *birth:baby* }

$impfea_{local}$ (animal) = { *mobility:self_moving* }

$impfea_{local}$ (living_ organism) = { *life:living, occurrence:natural, state:solid* }

Here the implicit features are retrieved from cow to living_organism to demonstrate the sequence of our operations.

The explicit features are extracted in a simpler way. For each semantic group, its set of explicit features is extracted form $PRJHOME / data / capt / <capid> / <sem_group> which has been stored in logical form. For example, the explicit feature set of cow in caption 1232 is kept in $PRJHOME / data / capt / 1232 / cow. In this case, cow has only one explicit feature pair.

$expfea$ (cow_{1232}) = { *size:large* }

The algorithm for the remaining scoring procedures has been clearly illustrated. No further breakdown will be repeated here.

This is how the query in Chapter 3, "A large cow is eating grass in a farm", is processed. In that Chapter, most of the gross and fine matching procedures have been demonstrated on high level view. With the supplement of low level details just delivered, the whole picture how ARMON matches NL queries with NL captions should become clear. As a result of the query, two close captions C_1 and C_2 are returned.

sim (Q, C_1) = ...

= ....

= 0.53

sim (Q, C_2) = ...

= ....

= 0.81

In this example, the corresponding degrees of similarity are returned together with the captions. The calculation which has been shown in previous chapter will not be repeated here. As mentioned before, the user may vary the tightness of retrieval by adjusting the retrieval threshold. If the user finally rejects the results obtained so far, ARMON can be instructed to search for more captions.

In this chapter, some examples have been employed to illustrate the algorithm about how the gross and fine matching algorithms work. Although this model is still far from an ideal retrieval system, it is a good illustration of applying linguistic information in information retrieval. In the next chapter, we are going to discuss some experiments with our prototype. Some difficulties will also be addressed there, along with some ways for enhancement.

CHAPTER 6

DISCUSSION

As long as
truly intelligent and multi-purpose machines are beyond our reach,
research and development has to concentrate on smaller tasks
that do seem to be feasible.

B.C. Papegaaij *[Pape86]* p. 53

We shall now give some results and general discussions of the whole project. We shall firstly present some general measurements of ordinary Information Retrieval (IR) systems, and then walk through the procedures and results of some of our experiments with the ARMON prototype. Next the difficulties encountered in the project will be addressed. Certain soundness and weakness of ARMON will be brought out, and improvements of the model will be raised. Finally a brief conclusion of the whole project will be given.

6.1 Performance Measures

In order to evaluate our prototype , we have to introduce some means for measuring the performance of general IR models. However, the general measurements do not sufficiently represent well our model. Thus we shall discuss the behaviour of the prototype in terms of other factors, using the results of some experiments as illustrations.

6.1.1 General Parameters

Recall and precision are two major numeric dimensions for measuring the performance of a general IR system which basically focuses on matching retrieval results with a given query. These retrievals are widely used as a scale of performance measurement [Tesk82, Jaco92, etc.].

Recall measures the coverage of a retrieval. We abbreviate it as RECL in the rest of this thesis. It is given as the ratio of relevant captions which have been retrieved and the total number of "really" relevant captions stored in the caption

database. For an ideal IR system, it equals to one. i.e. all relevant captions have been retrieved without being overlooked. In reality, it is usually less than one since some relevant captions have been missed, i.e. not retrieved.

Precision measures the accuracy of a retrieval, which is hereafter symbolized as PREC. It is calculated as the ratio of relevant captions retrieved to the total number of retrieved documents. For an ideal IR system, the precision also equals one, i.e. all retrieved captions are relevant without exception. For real-world systems, the precision is less than one since some retrieved documents are not relevant.

The meanings of these two parameters are best described in Figure 6.1.

	Relevant	Irrelevant
Retrieved	w	y
Not Retrieved	x	z

Figure 6.1 Several Parameters for Measuring Performance

We have, N = $w + x + y + z$ (population of candidate captions)

PREC = $w / (w+y)$

RECL = $w / (w+x)$

For ideal systems, PREC = 1 and RECL = 1

In addition to the above definition of RECL and PREC, there is a special condition which has not been considered in most articles, the condition of zero divided by zero. These cases occur when $w+y=0$ and/or $w+x=0$. In these circumstances, we simply label them as **undefined**.

The parameters PREC and RECL were mainly, at their best, formulated for measuring "traditional" IR systems which virtually focus on exact matching. These systems make only "all-or-none" decisions. That is, they only grade

candidates being only relevant (1) or irrelevant (0). They do not differentiate how great they are relevant or irrelevant.

Distinguished from those systems, ARMON was designed for approximate or inexact matching as well. In principle, ARMON is able to rank every candidate caption and gives it a value of numerical score of relevance between 0 and 1. However, in practice, only sufficiently close captions which have been extracted from gross filtering will be ranked.

For inexact retrieval, some less relevant items are intentionally returned by lowering the threshold. The parameters "relevant" and "irrelevant" referred in Figure 6.1 are less meaningful and hence PREC and RECL are less representative for measuring the performance of such a system. We shall show other measures for systems like this. On the other hand, ARMON can function like an "exact" IR system when the threshold is set to a value close to 1, say 0.9. In this case, RECL and PREC are more meaningful because the retrieval are now aimed to be exact. This will be addressed soon in our experiments.

6.1.2 Experiments

To examine the behaviour of retrieval with ARMON, some experiments were done in matching a set of captions against several queries. In the experiments, a collection of 50 captions were built. Those captions were written according to our grammar defined in Figure 2.4. The sample queries were also created with these rules. It was believed that 50 captions would be sufficient to illustrate some major behavior of approximate retrieval with the prototype.

Q1 : a large sheep is eating the long grass
Q2 : a horse is drinking water
Q3 : a white horse is eating grass in a large farm

6.1.2.1 Inexact Match Behaviour

We firstly examine this inexact retrieval behaviour of ARMON. The queries Q1, Q2 and Q3 are fed one by one to retrieve the set of our captions. By varying the

threshold and recording the number of captions being retrieved, we study the behaviour on the plotted graph, Figure 6.2.

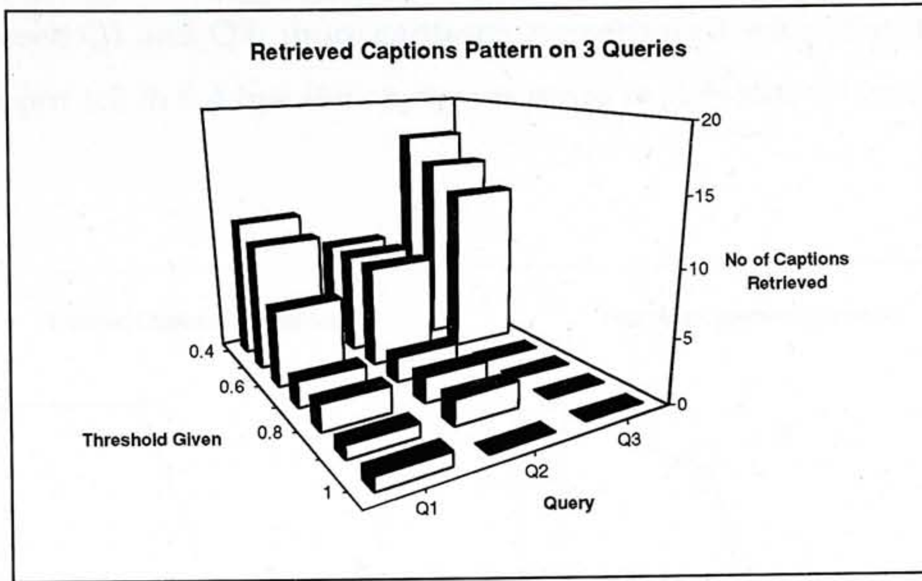


Figure 6.2 Retrieved Captions with Varying Threshold

In the experiment, several properties of the graph are noticed. We shall get into them point by point. For traditional IR system, each candidate caption is either retrieved or not. In ARMON, captions are not only retrieved but also ranked with match score.

In the illustration, we observe that the retrieval patterns of Q1, Q2 and Q3 are different. For Q1, the number of retrieved captions are gradually dropping when threshold is changing from 0.4 to 1.0.

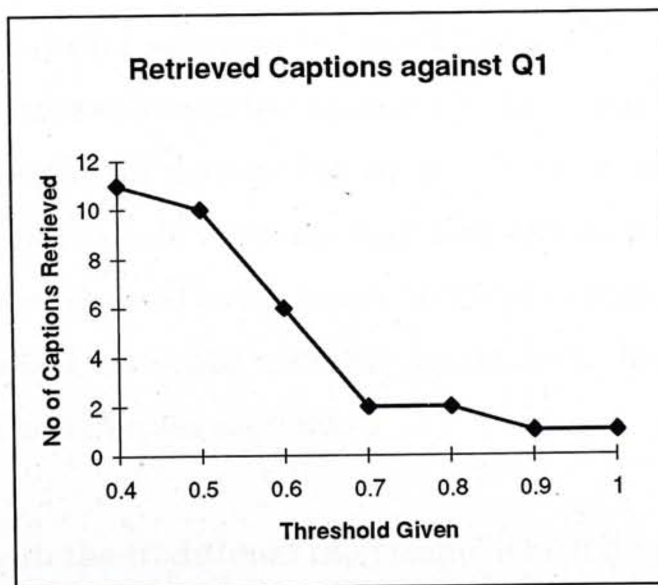


Figure 6.3 Retrieved Captions against Q1

This is the most typical behavior in our experiments, most of which is not presented in this article. On the other hand, the behaviour for Q3 is considerably

different. No caption is retrieved when the threshold is set at 1.0, 0.9, 0.8 and 0.7 but there is a dramatically jump when the threshold is set to 0.6. Q3 produces a rate of change significantly steeper than what Q1 does. For Q2, the pattern is lying between Q1 and Q3; more captions are retrieved when the threshold is adjusted from 1.0 to 0.4 but the change is more rapid than Q1 and less rapid than Q3.

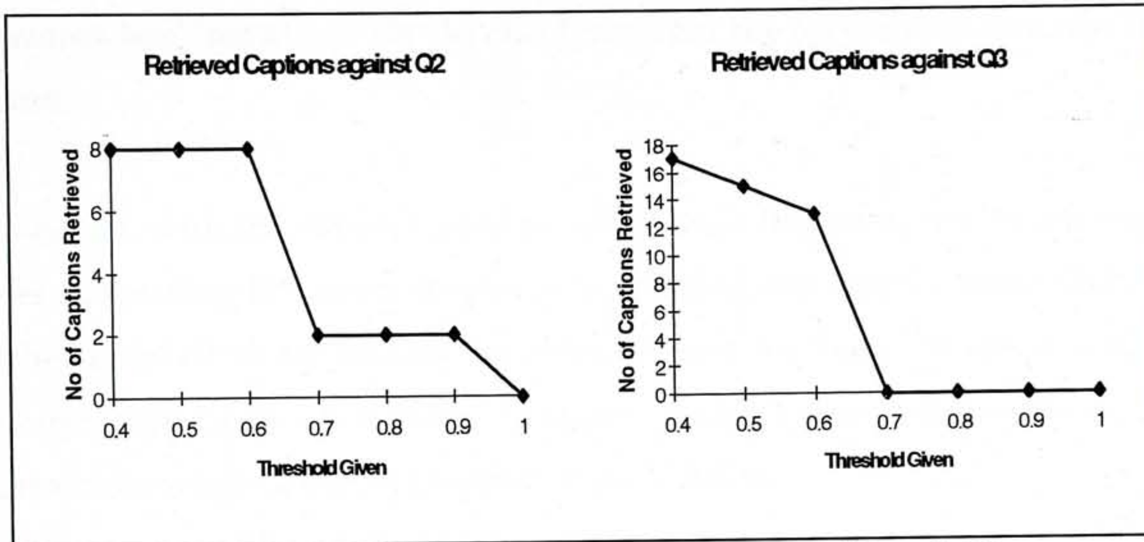


Figure 6.4 Retrieved Captions against Q2 and Q3

It is noticed that the retrieval behaviour may vary greatly. The advantage of approximate retrieval is clearly shown. For most existing traditional keyword-based retrieval systems, such as the one currently used in the libraries in our university, the retrieved items are not ranked. They do not tell users the degree of relevance of the retrieved (or non-retrieved) items. Many close items will probably be overlooked and, hence important or useful information may be lost. For Q3, there is no exact solution (score=1.0) for this query. In this case, ARMON user will have to loosen the tightness by lowering the threshold to 0.8, say. Even that there is no match. The user will have to further lower the threshold to 0.6 before some captions matching the user's query are found. Thirteen captions are returned as the result. If thirteen captions are too many for the user, he/she may fractionally increase the retrieval threshold or manually select among this set of thirteen captions.

Comparatively, with the traditional IR systems which do not support ranked output, if the user wants to get some close items, he/she probably has to search around all candidate captions, a total of 50 in our case but much more in real-world applications. A commonly experienced example is the searching of books

in a library. Most, if not all, of the on-line catalogue systems used in libraries provide only keyword-based exact matching which returns only and all exact matches of keywords even though the retrieved documents are semantically incorrect.

Imagine that a user wants to search for titles with content of the keyword "gun" but none is found. He may have to find something else by alternating the keywords like "pistol" or "rifle" to check whether the caption set contains these entities.

In contrast, with the concept used in ARMON, a hierarchy and some feature tables concerning the above items can be created and approximate searching can be achieved based on this information base. Systems designed with our concept reduce much human concern. This shows another benefit of approximate retrieval concept employed in ARMON.

6.1.2.2 Exact Matching Behaviour

Referring to the principles discussed in the previous section, ARMON is a superset of the traditional exact matching systems. It can work as an exact IR system as well by setting the retrieval threshold equal to 1. In the rest of this section, we switch to examine the exact matching properties of ARMON. Firstly we choose a retrieval threshold at 0.9 which is believed to indicate a caption being close enough to the query.

According to the principle addressed in the previous section, especially the table indicated in Figure 6.1, we have to justify whether each retrieved caption being relevant to the query or not. The values of w and y are obtained accordingly. Then we have to look into each candidate caption to check how many captions are actually relevant. Another parameter, x, standing for number of non-retrieved relevant captions, is obtained. Since the involved captions are not large in number, we can list all the involved captions here. The retrieved captions are listed in Figure 6.5.

Query	Caption retrieved with threshold = 0.9	Relevance
1	a large sheep is eating the long grass	Yes
2	a cow is drinking water	Yes
	a cow drinks water	Yes
3	(none)	-

Figure 6.5 Retrieved Captions with Threshold set to 0.9

A second problem is how we determine whether a caption is relevant or not. There is no fixed rule to determine how two NL sentences are relevant to each other. It depends on the requirement of the user and application. We have to manually justify the relevance of the retrieved and non-retrieved captions. We define a rule saying that the caption is relevant to the query if all three semantic heads in the caption are equal to the corresponding semantic heads in the query. Reading from the examination result in Figure 6.5, all the retrieved captions are relevant to the captions. With the extended definition in the previous section, the precision of retrievals concerning Q1 and Q2 are simply 1 and the precision of retrieval with Q3 is undefined.

$PREC_{Q1} = 1/1 = 1$, and

$PREC_{Q2} = 2/2 = 1$

We then scan the set of all candidate captions and manually examine which captions are actually relevant to the queries. For Q1, two more non-retrieved captions have semantic heads equal to those in the query. These captions are

a sheep is eating grass

a sheep eats grass

We regard these captions "relevant" according to our previous assumptions

Now, $w = 1$ and $x = 2$

then $RECL_{Q1} = 1 / (1+2)$

$= 0.33$

For Q2, there is no observable non-retrieved caption which matches the query.

Now, $w = 2$ and the highest $x = 0$

then $RECL_{Q2} = 2 / (2+0)$

$= 1.0$

For Q3, there is no caption that matches the query and hence both recall and precision are undefined as previously addressed.

i.e. $PREC_{Q3}$ = undefined

$RECL_{Q3}$ = undefined

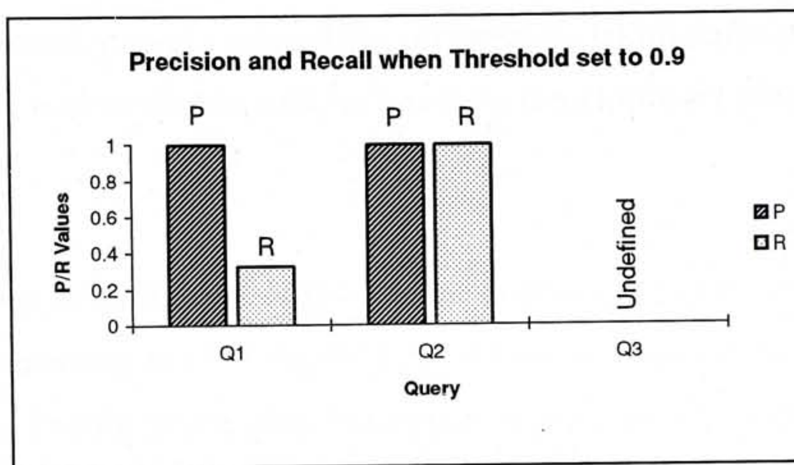


Figure 6.6 Precision and Recall Results

Figure 6.6 is a graphical representation of the results of PREC and RECL for the sample captions against the queries Q1, Q2 and Q3, with the retrieval threshold set at 0.9. With this example, it is seen that the precision is pretty good but the recall is a bit fluctuating. It is expected that the precision will drop and the recall will rise if the tightness of retrieval is reduced by lowering the retrieval threshold, say.

6.2 Difficulties

We had encountered a lot of difficulties in the course of project development. The first difficulty, as addressed previously, lay on the seeking for an adequate NL parser. Initially we tried to find a syntactic or semantic parser but it eventually happened to be unsuccessful. The morphological parser, PC-KIMMO, which we eventually found, had not been known to be used in any IR applications elsewhere. The problems in making use of a parser like this soon

became evident. There are many other difficult challenges in this project. We shall immediately look into a few of the highest concern to us.

Having read a number of articles, e.g. [Lewi89], [Guli92] and [Haln90], done by researchers involved in the area, we tried to gather more practical details by getting in touch with the authors. Some of the authors could not be reached by postage or email, e.g. Montgomery [Mont89]. Some of them had already discontinued from their work, e.g. Frump [DeJo79], in this field. Some wanted to keep exclusive use of their parsers, e.g. TRUMP [Crof90, Jaco92, Lewi89], and Naive Semantics [Dahl89]. Some authors were eager to share their parsers, e.g. McFrump [Maul89] and MCHART [Thom83] but further development based on those experimental parsers were difficult because (i) no sufficient developer's documentation was available, and/or (ii) their development platforms were not available to us.

At the first stage of designing ARMON, there was a plan of using Prolog as the major programming tool. Knowledge of the real world model would be represented as Prolog predicates. However, this proposal encountered a lot of difficulties. At the very beginning, a parser DBG written in Quintus Prolog [Mont89] was selected as the parser for ARMON. DBG is a large software consisting tens of thousand lines of source code. The documentation available to us, however, is only a journal article of several pages . We were unsuccessful to contact the authors [Mont89] regarding for further documentation. Guglielmo [Guli92] had developed a prototype of retrieval system, MARIE, based on the DBG parsers. We tried to ask him for more documentation about the DBG parser but were told that no more written documentation was available from the original authors. Consequently we gave up using DBG as the NL parser in the project and Prolog was dropped as well.

Even when we were willing to pay for a commercial NL parsing toolbox, there was no appropriate product available together with source codes. Fortunately a morphological parser, PC-KIMMO, was eventually found usable in our research. The associated lexicon, ENGLEX, was a rather complete English lexicon. There were excellent documentation and illustrations with this parser. Further, the software and hardware platforms are common and available to us.

The inter-relation among PC-KIMMO, ENGLEX and ARMON will be covered in Appendix G. Although they are not a pair of ideal parser and lexicon, they have given good performance after incorporation of our own context analysing rules as mentioned previously.

6.3 Possible Improvement

Many IR projects done by researchers worldwide, e.g. ADRENAL [Lewi89], MARIE [Guli92], had already accumulated many years of experience. Here in our university, we just started our investigation without any previous history in this area of study. We had neither a mature NL parser and lexicon, ncr related knowledge base. We did not even have any existing captions and queries for testing. All these had to be done by ourselves. It was really a great challenge for us. Significant experience has been already accumulated to get to the current stage of the ARMON project. It is a great advantage to continue research in this area in the future.

There is no doubt that the current ARMON model is not yet a perfect system. There are many enhancements which should be done to increase the performance and usability. For instance, appropriate modern programming techniques should raise the modularity, expandability and reusability of the final product. Prolog had been tried as a major development tool but was finally given up because of many reasons already mentioned. Some object oriented tools such as C++ had also been attempted but it also failed because of the problems in integrating external toolboxes which were coded with incompatible programming techniques. To achieve these enhancements, much effort had to be spent to re-structure the imported components. This plan was eventually given up because of the high uncertainty.

Finally the C Language was selected with reasons also mentioned before. To overcome some problems known to exist with the use of this language, we tried our best effort to structure our programs to keep ARMON's maintainability and reusability. For example, all coding was intentionally written to conform to the ANSI-C standard. Most operating parameters were dynamically configurable instead of "hard-coded" in the sources. With these provisions, the code was

believed to be more portable across platforms. This was finally proved as the code happened to be portable across several different Unix platforms available to us. Some discussions had been addressed in the previous chapters.

In the development of ARMON, version 1 of PC-KIMMO, which was then the latest version available, is employed as the core parser. As previously mentioned, it is just a morphological parser. This parser was chosen simply because we did not have any better alternative. As a morphological parser, its major pitfall is the lack of capability of analyzing context information. To cope with this limitation, we developed certain amount of necessary additional routines to analyse the context information. Together with the restricted grammar, it improved the resolution power of the parser to a great extent.

We still believe that a syntactic or semantic parser is much superior to a morphological parser. Recently, a beta release of PC-KIMMO Version 2 was announced. As in Version 1, it is freely distributed on the Internet. It is a good news for us although it is a bit late. This version is now equipped with a unification-base word grammar [Antw95a]. With this component, syntactic analysis can be done with much less difficulty. Together with PC-KIMMO 2, Version 2 of ENGLEX [Antw95b] is also in its beta release. Although these two products are not yet mature, they are believed to be a good couple of linguistic tools for the further development of ARMON.

In the Summer Institute of Linguistics, Dallas, PC-KIMMO is being used as a major component in the development of another NLP tool, PC-PATR, which is a syntactic parser based on the PATR-II formalism [Shie84]. The alpha release of PC-PATR is now available [McCo95]. This is a good illustration for the application of the new release of PC-KIMMO and ENGLEX.

Because this new version was available only in March this year (1995), we had not had the time to rewrite most of ARMON with this version of PC-KIMMO. Many modules would have to be rewritten to certain extent in order to integrate the syntactic components of the parser. However, it can be done without great difficulty. The new versions of PC-KIMMO and ENGLEX can provide much improvement to our model. The most appreciated enhancement, as mentioned

before, is that less effort is required for context analysis. When this pair of NL tools grows mature, it is reasonable to integrate this couple of improved parser and lexicon later in the development of ARMON.

Execution speed is an issue which we have not addressed. Although there is not yet any formal measurement, it is observed that the current implementation of ARMON executes at a moderate speed. Some further work can be done to improve the execution speed. In current implementation, there is no well designed cache or buffer to keep the most recently retrieved knowledge. In some cases, the speed is unnecessarily retarded because the same piece of information, such as a particular implicit feature, is retrieved again and again form the raw data files. Without great alternation to the algorithm of current model, an appropriate caching mechanism could improve the execution speed.

Another potential improvement should be done on the restrictive grammar. As mentioned in Chapter 2, the grammar can be relaxed by adding several rules to make decorative participle phrase become acceptable. With the enhanced parser, PC-KIMMO 2, such an extension will involve little effort. Further relaxation can be done depending on requirements of application domains.

6.4 Conclusion

Information retrieval of natural language text is obvious and natural to human beings but hard for machines to emulate. The main difficulties happen in the area of computerizing natural language understanding and processing. Researches in this area attempt to integrate artificial intelligence, computational linguistic and database management techniques. Up to the present moment, none of the known works has empowered the language processing capability remotely comparable to the human brain. There is no commercially available multimedia system which takes full advantage of the linguistic approach. Most of the practical systems are still existing as "directories and files" system. Lack of user-oriented retrieval method is a major pitfall of those systems. All the linguistic-aided IR systems are still experimental in research institutes. Much advance and much work have to be done before those research items can be put to real-world systems.

At the very beginning of this thesis, we have proposed the application of linguistic tools in fetching desired media objects from a large population of media data. ARMON is based on this concept and has been designed as such a model for "Approximate Retrieval of Multimedia Objects by Natural Language Captions".

Several ideas have been put forth to reduce the complexity of natural language processing for the purpose of this kind. We have argued that accurate logical representation of captions and queries are not necessary. Certain degree of approximation is reasonable because an all-purpose NL understanding system is not the goal of this project. According to the nature of the media objects, several ways to approach approximation have been suggested.

ARMON starts matching by recognizing only the root form of words, followed by going into low level details, then by normalizing the features, and searching entities on the type hierarchies, and finally by ranking the grossly selected captions. These are briefly the main steps of such a multi-level approximation with ARMON.

The resultant prototype of ARMON has shown, to certain degree, that this concept is a good approach that improves the traditional keyword retrieval. This is actually what we have expected before starting the project. Below is a summary to state what and how ARMON achieves functions which have been planned.

i. Each token in a sentence will be mapped into its root form before further processing. Some minor information, such as tense or plurality, etc. will be ignored at this level. This simple operation minimizes any word undetected due to the variety of its lexical form. This will, hopefully, increase the recall of the system.

   If the sentence is a new caption, the decomposed sentence will be placed in various segments over the ARMON databases. Otherwise, if the sentence is a query, retrieval will get into the next step.

ii. The collection of candidate captions will pass through a gross filtering process, in which only roughly matched captions will be extracted. This screening process considers only the semantic heads and makes use of knowledge of classification with a type hierarchy. This level of matching compares only the gross closeness between a query and the candidate captions.

iii. Each roughly matched caption is individually scored against the query. A numerical matching score between 0 and 1 is calculated by an algorithm which considers the implicit and explicit properties (features) of the semantic entities in the captions and the query. This finer level matching ranks the candidate captions according to their numeric scores.

iv. The tightness of the retrieval is adjustable by a user-given threshold. The user can optimize the Recall and Precision according to preferences case by case. The sufficiently close captions are ranked and presented to the user.

At the current stage of the project, we had created the necessary knowledge bases for searching. For example, we created a hierarchy for noun-type entities, another hierarchy for verb-type entities, a table holding feature slots and values, and a table for transforming and normalizing attributes from lexical form into features. The imported parser and lexicon, PC-KIMMO and ENGLEX have been rewritten and integrated. Code has been written to integrate all these knowledge bases and modules into an operational system.

There is no doubt that all these knowledge and programs in ARMON are still at a prototype stage. ARMON is, however, working quite satisfactorily according to various measurements. It can be expanded to adopt to a realistic application on many domains of the real world with some further work. The expansion should not take a lot of effort. For instances, if ARMON is switched to the domain of sports, a hierarchy of sports events or equipment, etc should be built.

Implicit features of individual entities should also be defined. These tasks are better done by experts in sports.

Different from other IR systems, here we make use of two separate knowledge bases for the gross and fine levels of matching. ARMON calculates the matching scores using the features of semantic groups after filtering by the type hierarchy. Working with two independent knowledge bases at different level is anticipated to reduce any one-sided bias that may exist. In other words, the fine level scoring determines the matching score of a candidate caption independent of the geometry of the type hierarchy once it has been grossly filtered.

The benefits and difficulties of linguistic approach in multimedia data retrieval have been discussed. We should realize that there are so many barriers and immaturity in the use of natural language processing techniques to depict and retrieve multimedia data. Although great effort is still required to refine the performance of the ARMON model, it is yet far more realistic than the content analysis of the media data themselves in their original forms. We strongly believe that our proposed approach is the most practical to develop content retrieval means for multimedia data.

REFERENCES

REFERENCES

[Antw90] Antworth, E.L.
PC-KIMMO: A Two-level Processor For Morphological Analysis. Occasional Publications in Academic Computing No. 16., Summer Institute of Linguistics, Dallas, 1990.

[Antw91] Antworth, E.L.
Introduction to two-level phonology. *Notes on Linguistics*, 53:4-18, Summer Institute of Linguistics, Dallas, 1991.

[Antw92a] Antworth, E.L.
PC-KIMMO 1.0.8 User's Guide, On-line documentation, package available on FTP: // ftp.sil.org/software/unix/pckim108.tar.Z

[Antw92b] Antworth, E.L.
ENGLEX 1.0 User's Guide, On-line documentation, package available on FTP: // ftp.sil.org/data/unix/englex10.tar.Z

[Antw92c] Antworth, E.L. and McConnel, S.R.
KTEXT 1.0.3 User's Guide, On-line documentation, package available on FTP: //ftp.sil.org/software/unix/ktext103.tar.Z

[Antw92d] Antworth, E.L.
KGEN 0.3 User's Guide, On-line documentation, package available on FTP:// ftp.sil.org/software/unix/kgen03.tar.Z

[Antw95a] Antworth, E.L.
PC-KIMMO 2.0β27 User's Guide, On-line documentation, package available on FTP: //ftp.sil.org/software/unix/pckimmo020b27.tar.Z

[Antw95b] Antworth, E.L.
ENGLEX 20β3 User's Guide, On-line documentation, package available on FTP: //ftp.sil.org/data/unix/englex20b3.tar.Z

[Crof90] Croft, W.B.
Towards Intelligent Information Retrieval : An Overview of IR Research at U. Mass. *Data Engineering*, IEEE, Vol. 13 (1990), No 1, 17-24.

[Dahl88] Dahlgren, K.
Naive Semantics for Natural Language Understanding, Kluwer Academic Publishers, 1988.

[Dahl89] Dahlgren, K., McDowell, J. and Stabler, E.P.
Knowledge Representation for Common sense Reasoning with Text, *Computational Linguistics*, Vol. 15 (1989), No 3, 149-170.

[DeJo79] DeJong, G.
Prediction and Substantiation: A New Approach to Natural Language Processing, *Cognitive Science*, Vol. 3 (1979), 251-273.

[Deli91] Dalianis, H.
A Method for Validating a Conceptual Model by Natural Language Discourse Generation, SYSLAB Working Paper No 190, Department of Computer and Systems Sciences, Stockholm University, Sweden, 1991.

[Drey81] Dreyfus, H.L.
From Micro-Worlds to Knowledge Representation, in J. Haugeland (ed.), *Mind Design: Philosophy, Psychology, Artificial Intelligence*, MIT Press, 1981.

[Fill68] Fillmore, C.
The Case for Case, Universal in Linguistic Theory, Holt, Holden-Day, New York, 1964.

[Gall91] Gallant, S.I.
Context Vector Representation for Document Retrieval, *AAAI-91 Natural Language Text Retrieval Workshop*, Anaheim, CA. July 15, 1991.

[Gibr82] Gibran, K. and Sayings, S.
The Next Generation of Text Processing Systems, in Teskey, F.N. (ed),

Principles of Text Processing, Ellis Horwood Publisher, 1982.

[Guli92] Guglielmo. E.J.
Intelligent Information Retrieval for a Multimedia Database Using captions , Ph.D. Thesis, Department of Computer Science, Naval Postgraduate School, 1992.

[Hahn90] Hahn, U.
Topic Parsing : Accounting for Text Macro Structure in Full-Text Analysis, *Information Processing & Management*, Vol. 26 (1990), No 1, 135-170.

[Hirs87] Hirst, G.
Semantic Interpretation and the Resolution of Ambiguity, Cambridge University Press, 1987.

[Holt90] Holtkamp, B., Lum, V.Y. and Rowe, N.C.
DEMON - A Description Based Media Object Model , *Proceedings of International Computer Software and Applications Conference (COMPSAC)*, Chicago, Oct 31 - Nov 2, 1990.

[Jaco92] Jacobs, P.S.
TRUMPT: A Transportable Language Understanding Program, *International Journal of Intelligent Systems*, Vol 7 (1992), 245-276.

[Jern92] Jernigan, Richid
MetalBase User's Guide, On-line documentation, package available on FTP://sunsite.ust.hk/pub/Linux/apps/database/mbase.tar

[Kart83] Karttunen, L.
KIMMO: A General Morphological Processor. *Texas Linguistic Forum*, 22: 163-186, 1983.

[Kosk83] Koskenniemi, Kimmo
Two-level Morphology: A General Computational Model for Word-form Recognition and Production. Publication No. 11, Department of General Linguistics, University of Helsinki, 1983.

[Lewi89] Lewis D.D., Croft, W.B. and Bhandaru, N.
Language-Oriented Information Retrieval, *International Journal of Intelligent Systems*, Vol 4 (1989), 285-318.

[Lum90] Lum, V.Y. and Meyer-Wegener, K.A.
An Architecture for a Multimedia Database Management System Supporting Contents Search, Advances in Computing and Information, *Proceedings of the International Conference on Computing and Information*, Niagara Falls, Canada, May 23-26, 1990.

[Lum93] Lum, V.Y. and Wong, K.P.
A Model and Technique for Approximate Match of Natural Language Queries. *Proceedings of InfoScience'93, the International Conference in Commemoration of 20th KISS Anniversary*, Seoul, Korea, 1993, 525-534.

[Mann89] Mann, W.C, Matthiessen, C.M.I.M. and Thompson, S.A.
Rhetorical Structure Theory and Text Analysis, Research Report ISI/RR-89-242, Information Science Institute, University of South California., 1989.

[Maul89] Mauldin, M.L.
Information Retrieval by Text Skimming, Ph.D. Thesis, Carnegie Mellon University, 1989.

[McCo95] McConnel, S.
PC-PATR 0.96α13 User's Guide, On-line documentation, package available on FTP: // ftp.sil.org/software/unix/pcpatr096a13.tar.Z

[Mill86] Miller, J.R.
A Knowledge-Based Model of Prose Comprehension: Applications to Expository Texts, in Britton, B.K. (ed), *Understanding Expository Text*, Lawerence Erlbvaum Associates, New Jersey, 1986.

[Mont89] Montgomery, C.A. et. al.
The DBG Message Understanding System, *Proceedings of the Annual AI Systems in Government Conference*, Washington, D.C., March 27-31, 1989.

[Nils80] Nilsson, N.L.

Principles of Artificial Intelligence, Morgan Kaufmann Publishers, Inc., Los Altos, 1980.

[Nire87] Nirenburg, S. and Raskin, V.
The Sub world Concept Lexicon and the Lexicon Management, *Computational Linguistics*, Vol 13 (1987), No 3-4, 276-289.

[Pape86] Papegaaij, B.C., Sadler, V. and Witkam, A.P.M.
Word Expert Semantics - an Interlingual Knowledge-Based Approach, Foris Publications, Dordrecht, Holland, 1986.

[Quir72] Quirk, R., Greenbaum, S., Leech, G. and Svartvik, J.
A Grammar of Contemporary English, Longman, 1972.

[Rand93] Random House Inc., Word Perfect Corp.
Random House Unabridged Electronic Dictionary, Utah,1993

[Rowe94] Rowe, N.C.
Inferring Depiction in Natural Language Captions for Efficient Access to Picture Data, *Information Processing & Management*, Vol 30 (1994), No 3, 379-388.

[Seo89] Seo, J. and Simmons R.F.
Syntactic Graphs : A Representation for the Union of All Ambiguous Parse Trees, *Computational Linguistics*, Vol. 15 (1989), No 1, 19-32.

[Sgal86] Sgall, P., Hajicova, E. and Panevova, J.
The Meaning of the Sentence in its Semantic and Pragmatic Aspects, D. Reidel Publishing Company, Czechoslovakia, 1986.

[Shie84] Shieber, S. M.
The Design of a Computer Language for Linguistic Information, *Proceedings of Coling84, 10th International Conference on Computational Linguistics*, Stanford University, California, July 2-7, 1984, 362-366.

[Simo91] Simons, G.F.
A Two-level Process for Morphological Analysis. *Notes on Linguistics*, 53:19-27, Summer Institute of Linguistics, Dallas, 1991.

[Smea90] Smeaton, A.F.
Natural Language Processing and Information Retrieval, *Information Processing & Management*, Vol. 26 (1990), No 1, 111-134.

[Teng90] Tengku, M.T. and van Rijsbergen, C.J.
SILOL : A Simple Logical-Linguistic Document Retrieval System, *Information Processing & Management*, Vol. 26 (1990), No 1, 111-134.

[Thom83] Thompson, H.
MCHART: A Flexible, Modular Chart Parsing System, *AAAI-83*, 1983, 408-410.

[Tsud91] Tsuda, K., Yamamoto, K., Hirakawa, M., Tanaka, M. and Ichikawa, T.
MORE: An Object-Oriented Data Model with a Facility for Changing Object Structures, *IEEE Transactions on Knowledge and Data Engineering*, Vol. 3 (1991), No. 4, 444-460.

[Vann90] Vanni, M.
Abstract of "PC-KIMMO: a two-level processor for morphological analysis.", *Journal of Languages & Linguistics* 4:498-500, Georgetown,1990.

[Webe88] Weber, D. J., Black, H.A. and McConnel, S.R.
AMPLE: A Tool for Exploring Morphology, Occasional Publications in Academic Computing No. 12, Summer Institute of Linguistics, Dallas, 1988.

[Wend91] Wendlandt, E.B. and Driscoll, J.R.
Semantic Extensions to Text Retrieval, *6th Symposium on Methodologies for Intelligent Systems - ISMIS'91*, Charlotte, Carolina, October 16-19, 1991.

[Woel87] Woelk, D. and Kim, W.
Multimedia Information Management in an Object-Oriented Database System, *Proceedings of the 13th International Conference on VLDB*, Brighton, England, Sept., 1987.

APPENDICES

Appendix A Notation

A.1 Abbreviations

$PRJHOME	Home directory of the ARMON source tree
AJ	Adjective
AJP	Adjective Phrase
attrac	Set of attributes on Action
attrag	Set of attributes on Agent
attrpa	Set of attributes on Patient
AV	Adverb
AVP	Adverb Phrase
capid	Caption Identity
CAPT	Logical captions database
DET	Determiner
expfea	Explicit Feature(s)
EXPFEA	Logical Database of expfea
GUI	Graphical User Interface
HIER	Logical Database of Type Hierarchy
impfea	Implicit Feature(s)
IMPFEA	Logical Database of impfea
MDB	Multimedia Database
MDBMS	Multimedia Database System(s)
N	Noun
NL	Natural Language
NLG	Natural Language Generation
NLP	Natural Language Processing
NLU	Natural Language Understanding
nodid	Node Identity
nodename	Node Name
NP	Noun Phrase
PP	Preposition
PPP	Preposition Phrase

PREC	Precision
RECL	Recall
S	Sentence
V	Verb
VP	Verb Phrase

A.2 Fonts with Special Meanings

Meaning	Examples
attribute or feature	*size : large*
direct quote	a piece of quoted text
entity	animal
function	*impfea* ( )
path or filename	$PRJHOME/data/capt/impfea/animal
shell command	`hierop p animal`
Title of book or journal	*Information Processing & Management*

Appendix B Glossary

action	In a logical representation of NL sentence, action is the main process being exhibited. It is commonly a mental or physical process and is usually the verb part of sentence. It is also called activity in some articles.
agent	In a logical representation of NL sentence, agent is the entity which starts to carry out an action. It is usually the subject part in the sentence and is also called actor in some writing.
application domain	A finite real-life application area to which the media data belong.
ARMON	The name of our project, a partial acronym taken from "Approximate Retrieval of Multimedia Objects by Natural Language".
attributes	The lexical descriptive text for semantic groups. It will be transformed into features in ARMON.
capid	Caption identity, a numeric string to uniquely labeled a caption in the database CAPT.
caption	A description of media data written in natural language. See also natural language.
domain expert	A person who is familiar with the application domain and responsible for creating hierarchies and features in ARMON model.
ENGLEX	An English lexicon used as a part of ARMON.
entity	An object on the type hierarchy, used here exchangeably with node.
explicit features	The properties of a entity written in a natural language sentence.
feature	An attribute that depicts a property of the semantics of a natural language sentence. Each feature is paired up from a slot and a value. In ARMON, it is symbolized as *slot:value.*

fine scoring	A final stage of caption matching which takes account of individual feature sets of semantic groups.
gross filtering	An early stage of caption matching which takes into account of knowledge kept on the type hierarchy.
implicit features	The hidden properties of an entity unwritten in the natural language sentence. See also features and explicit features.
lexical form	The written form of a token (word) appeared in a sentence.
media data	Used exchangeably with multimedia data.
morphemes	A set of minimally meaningful units that compose a word. See also morphology.
morphology	The study of word structure.
multimedia data	Data items which contain non-textual contents, usually relating to audio or visual information.
natural language	A language used in communications in the communities of human being. In this article, it means the English language under a restricted grammar.
node	An object on the type hierarchy, used here exchangeably with entity.
nodid	Node identity, a numeric string to uniquely label a node on the type hierarchy.
nodename	Node name, a human readable name of a node on the hierarchy, can be written in abbreviated forms.
patient	In a logical representation of NL sentence, patient is the entity which is undergone an action or its result. It is usually the object part in the sentence.
PC-KIMMO	A morphological parser used as a part of ARMON.
precision	A performance measurement of general IR systems. See Chapter 6 for details.
query	A sentence which is written in restricted natural language issued by user to match close captions used in MDBMS.
recall	A performance measurement of general IR systems. See Chapter 6 for details.

semantic groups	In ARMON, any sentence is assumed to be composed of three semantic groups, namely agent group, action group and patient group, any one but not all of which may be null. Each group contains a head and a set of features. For simplicity, the term "semantic group" used in this article can stand for the head of the associated semantic group. It is used exchangeably with thematic roles. See also agent, action, patient and features.
semantic head	Short form of "head of semantic group", used exchangeably with thematic role.
sentence	Sentence used in ARMON is a statement written in English with restrictions.
syntactic class	Part-of-speech.
thematic roles	Heads of semantic groups. See also semantic groups.
type hierarchy	The tree-like representation of the sub-class and super-class relationship among world entities.

Appendix C Proposed Features

Feature	Examples of Value
act_on	food I animal I plant I environment I people
age	<numeral> + <unit> I old I young I infant I adult
appearance	dirty I bright I dark I clear
birth	egg I baby
colour	none I white I yellow I orange I brown I red I blue I green I purple
content	<phyobj>
diet	plant I animal I microorganism I inorganic I mixed
duration	<numeral> + <unit> I short I med I long
frequency	none I low I med I high
function	serve_man I computation I transport I none
gender	masculine I feminine I neuter
goal	live I transport I leisure
habitat	land I forest I desert I water I river I ocean I lake I home
haspart	<phyobj>
height	high I med I low
life	none I embryo I living I dead
manner	fierce I tamed I polite
material	<phyobj>
mobility	none I self_moving I fuel_driven I gravity_driven I flying I aquatic
name	<string>
nearby	<phyobj>
occurrence	natural I artificial I past I future
odour	none I fragrant I sour I stinking I choking
operation	<action>
opponent	gravity I obstacle I human
orientation	horizontal I vertical I declined
partof	<phyobj>
place_at	<place>
place_from	<place>

<table>
<tr><td>place_thru</td><td><place></td></tr>
<tr><td>place_to</td><td><place></td></tr>
<tr><td>power</td><td>weak | med | strong</td></tr>
<tr><td>processing</td><td>dried | cooked | preserved | fermented</td></tr>
<tr><td>relative_loc</td><td>near | far | above | below</td></tr>
<tr><td>shape</td><td>needle | round | rectangular | irregular | narrow</td></tr>
<tr><td>size</td><td><numeral> + <unit> | atomic | fine | small | med | large</td></tr>
<tr><td>speed</td><td><numeral> + <unit> | 0 | low | med | high | SOS | SOL (SOS = Speed of Sound, SOL = Speed of Light)</td></tr>
<tr><td>state</td><td>solid | liquid | gas | plasma</td></tr>
<tr><td>status</td><td>angry | happy | sad | tired</td></tr>
<tr><td>strength</td><td>hard | soft | brittle | tough</td></tr>
<tr><td>structure</td><td>simple | complicated | assembled</td></tr>
<tr><td>taste</td><td>tasteless | salty | sweet | bitter | sour</td></tr>
<tr><td>temperature</td><td><numeral> + <unit> | frozen | room_temp | hot | boiling | red_hot</td></tr>
<tr><td>texture</td><td>rough | smooth</td></tr>
<tr><td>time_at</td><td><numeral> + <unit> | past | now | future</td></tr>
<tr><td>time_from</td><td><numeral> + <unit> | past | now | future</td></tr>
<tr><td>time_to</td><td><numeral> + <unit> | past | now | future</td></tr>
<tr><td>tool</td><td>none | mouth | limbs | nose | ear | optical | electrical | mechanical</td></tr>
<tr><td>weight</td><td><numeral> + unit | low | med | height</td></tr>
</table>

Appendix D Sample Captions and Queries

Captions in Experiments

a cat saw a mouse

a cow drinks water

a cow eat corn

a cow is drinking water

a cow is eating bearing

a cow is eating corn

a giraffe is walking in the forest

a horse drinks cola

a horse eats grass

a horse eats long grass

a horse is drinking cola

a horse is eat grass

a horse is eating grass

a large cow is quickly eating green grass

a large lion is chasing a small dog

a large tiger is eating a big deer

a large tiger is hungrily eating a small deer

a large tiger is quickly eating a small deer

a lion is eating meat

a lion is running

a lizard is lying on the rock under sunshine

a rhinoceros is wondering on the grassland

a saw breaks

a sheep eats grass

a sheep eats vegetable

a sheep is eating grass

a sleeping lion

a small cow is eating brown grass

a small goat is eating brown grass

a turtle is laying eggs

many fishes are swimming in the river

some large lions are chasing a small dog

some small wolves are drinking dirty water

the bear catches a fish from the river

the cow is eating corn

the dog is barking

the dog jumps into the water

the giant whale is closing its big mouth

the large sheep is eating the long grass

the lazy lion is waiting for its food

the long grass is green

the monkey is asking for food

the monkey is breaking the coconut on the tree

the monkey is eating coconut

the monkey jumps between the trees

the shark opens its month

the storm killed many animals in one day

the swallow is building a net

the young birds are learning to fly

Queries in Experiments

a large sheep is eating the long grass

a horse is drinking water

a white horse is eating grass in a large farm

Appendix E Manual Pages

In this appendix, the usage of several major user commands in using ARMON is listed as manual pages. These commands include *capop*, *hierop* and *purgeCAPT*. As usual in Unix convention, all commands are case sensitive, i.e. mis-typing *purgeCAPT* as *purgecapt* will not be recognized by the shell. This example imposes an advantage to minimize the chance of CAPT being purged carelessly. The format of these manual pages should be familiar to those Unix users. These operation guides have been entered as the common Unix manual page format. It can be simply re-formatted into ordinary manual pages to be kept in the Unix file system and be read with the *man* command.

NAME

capop - caption operations in the ARMON system

SYNOPSIS

capop a [text file] [i|u]

capop d [text file]

capop s [text file] [i|u] [threshold]

DESCRIPTION

Command *capop* enables the user of ARMON to delete, add or search for caption(s) described in a text file.

The first argument is either *a*, *d* or *s* which stands for add, delete or search respectively. The second argument gives a filename which contains the material to be processed.

When the first argument is *a*, *capop* will look into the file specified in the second argument. The file should contain the to-be-added captions on a line.

If a line starts with the form of " nnnn > " where nnnn is an integer, it implies that the user prefers to add the following caption with a capid nnnn. If nnnn does not exist in the caption database, this NL caption will be inserted as a new caption. If nnnn is currently existing, the NL caption will overwrite the existing one with the same capid. If the third argument is *i*, it will run in the interactive mode. Otherwise, if it is *u* or left blank, it will run in unattended mode by default.

When the first argument is *d*, *capop* will look into the file specified in the second argument. This file should contain the caption identifier(s), capid. If there are multiple captions to be deleted. The file should contain the caption identifiers line by line, one on each line. If there is no caption matching the wanted capids, an error message will be reported on standard output.

When the first argument is *s*, *capop* will look into the file specified in the second argument. The file should contain lines of natural language sentences. These are the sentences to be searched according to our retrieval algorithm. If the third argument is *i*, it will run in the interactive mode. Otherwise, if it is *u* or left blank, it will run in the unattended mode by default. The fourth argument, threshold, is meaningful in this case. It should be given in a decimal value between 0.0 and 1.0, both inclusive. It indicates how "close" the captions is to be matched with the query in order to be retrieved. If this argument is missing, ARMON will automatically assume a default value of 0.75.

EXAMPLES

```
% capop a capfile
```

Assume that the file capfile now looks like the following :

```
a sheep is drinking water in the farm
13231 > a horse is eating grass
```

The first caption is going to be inserted into the CAPT database. A unique random number will be generated to be subsequently used as the capid for that caption. The second caption is going to be inserted with a capid 13231. If this capid does not exist, it will be inserted as a new one. Otherwise, if the same capid is already existing, it will replace the old one in the caption database.

```
% capop d capfile
```

Assume that the file capfile looks like the following :

```
1232
23421
```

This command attempts to delete captions numbered 1232 and 23421. It reports any error to the user.

```
% capop s capfile i 0.6
```

Assume that the file capfile now looks like the following :

```
a sheep is drinking water in the farm
a horse
```

The first and the second captions will be matched against the existing caption database. The value 0.6 is the threshold value; captions exceeding this

threshold will be retrieved. The forth argument, *i*, suggests that ARMON runs in the interactive mode, and the user will be consulted in case of any ambiguity.

——— ———

NAME

hierop - hierarchy operations in the ARMON system

SYNOPSIS

hierop a [entity]

hierop d [entity] [i/u]

hierop p [entity]

DESCRIPTION

Command *hierop* enables the user of ARMON to delete and add entities or print (list) the main hierarchy.

The first argument can be either *a*, *d* or *p*, which stands for an operation of add, delete or print, respectively. The second argument is an entity name. The third argument is optionally *i* or *u* which stands for interactive or unattended mode, respectively, for the operation.

When the first argument is given as *a*, *hierop* takes the second argument as the new entity to be added to the type hierarchy. The user will be asked a sequence of questions and the location of the new entity will be finally determined. The user will be given a hint on what the nodid can be and the user should then accordingly enter an unique nodid for the entity to be added.

When the first argument is *d*, *hierop* will examine the second argument *entity* as the top-most node of the sub-tree to be removed. If the third argument is *u*, the whole sub-tree headed by the node *entity* will be removed silently. This option should be used with great care because the operation is irreversible. Otherwise, if the third argument is given as *i* or not given at all, the interactive mode will be used as default. User will be

prompted for confirmation from the standard input. If the specified entity is not found on the hierarchy, an error message will be reported.

When the first argument is *p*, *hierop* will print in the output a fragment of type hierarchy starting with the second argument, *entity*. If *entity* is not given, the whole type hierarchy will be printed. If *entity* is not found, an error message will be reported.

EXAMPLES

% hierop a reptile

The operation tries to insert an entity named *reptile* into the type hierarchy. The user is ready to specify the entity interactively.

% hierop d bird i

The operation attempts to delete the sub-tree starting with *bird*. If the classes flying and non-flying are the children of *bird*, for example, the whole sub-tree will be deleted after getting the confirmation from the user.

% hierop p animal

This operation will print the sub-tree starting at the entity *animal*.

———✂———

NAME

purgeCAPT- purge the whole CAPT

SYNOPSIS

purgeCAPT [i|u]

DESCRIPTION

Command *purgeCAPT* enables the user of ARMON to remove all caption records in the CAPT database and leaves CAPT blank. The command takes an argument of either *i* or *u*. If *u* is entered as the argument, the hierarchy will be immediately removed. Otherwise, the user will be prompted for confirmation before the commitment of deletion.

EXAMPLE

```
% purgeCAPT   u
```

This command will attempt to clear the caption database after getting the confirmation interactively from the user.

Appendix F Directory Structure

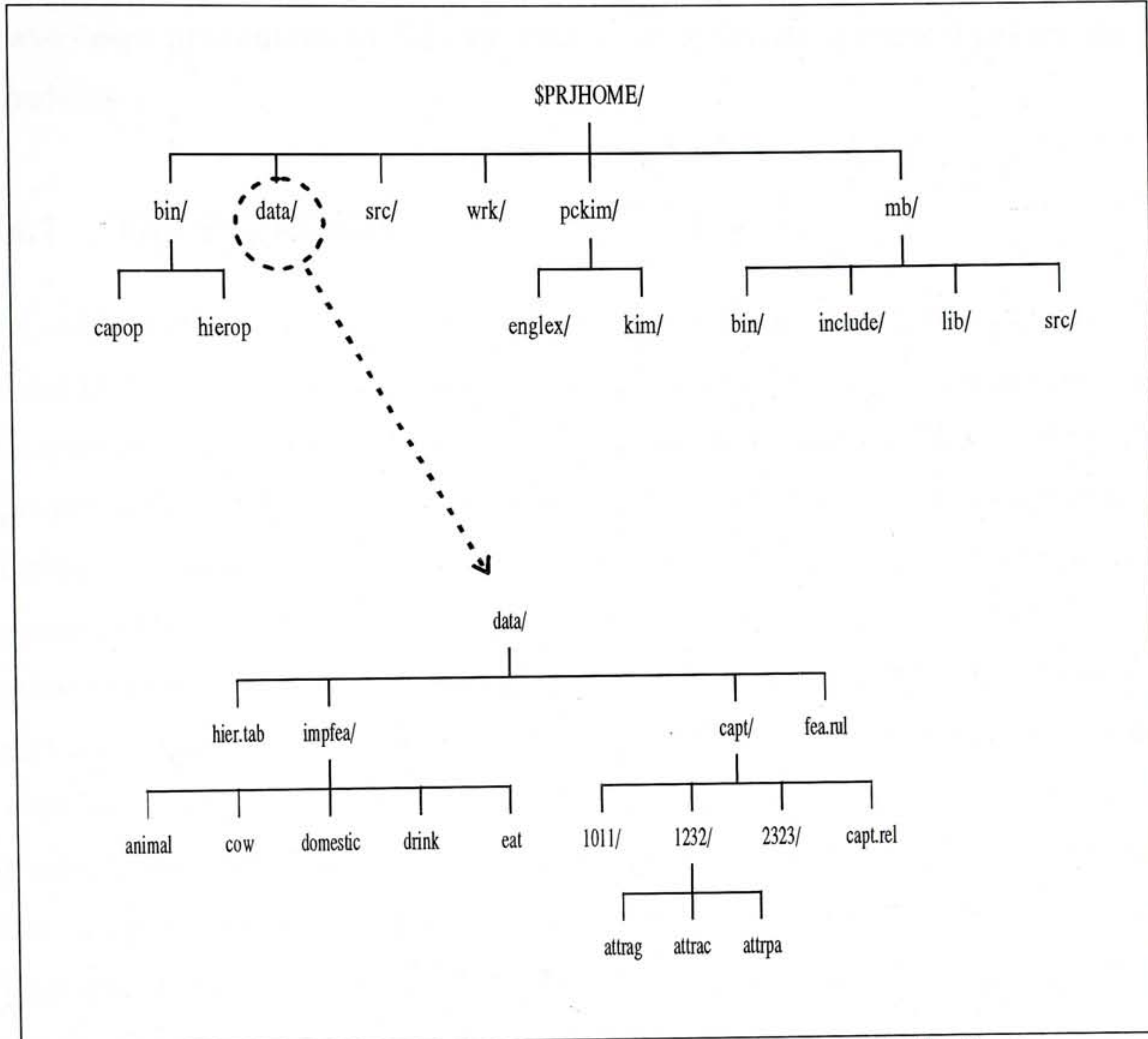


Figure F.1 ARMON Components under the Unix File System

Appendix G Imported Toolkits

In developing this project, two external software toolkits had been integrated, namely the PC-KIMMO and MetalBase. Some discussions on these modules have been presented. In this appendix, we provide more details on these modules.

G.1 The PC-KIMMO

PC-KIMMO is a successor of an older software KIMMO, named after its originator, Kimmo Koskenniemi who is a Finnish computational linguist [Antw90]. It made use of the 2-level morphology model. The PC-KIMMO project is currently led by E.L. Antworth in Summer Institute of Linguistics, Dallas. PC-KIMMO was initially written for PC with Intel x86 processors running MS-DOS. Later revisions were extended to become portable to run in other operating systems, including the popular names like Macintosh System 7 and Unix. Version 1 and mirror revisions of PC-KIMMO were released between 1990 and 1992. Version 1.0.8 was finally a stable version of PC-KIMMO Version 1 which was chosen as the parser in our project ARMON. Useful information could be found in the on-line text included in the PC-KIMMO package [Antw92a]. Background details of the 2-level morphology concept were fully discussed in Koskenniemi and Antworth's books [Kosk83, Antw90].

This parser will parse each lexical token (word) as an independent item. It does not associate the inter-word relationship, i.e. the context of the original statement is always ignored. This is the major pitfall of this "parser". Version 2 of PC-KIMMO [Antw95a] was aimed to correct this deficiency. One main task in ARMON was to build certain context analysing rules fed from the parser's raw output.

PC-KIMMO contains two major functional modules, namely the generator and the recognizer. The generator composes a word token from its components which are sometimes the morphemes [Antw90] of the token. According to the theory of morphology, each word is generally composed of three components,

namely the prefix, root and suffix. Each component gives a particular contribution to the meaning of the word. Obviously the prefix and/or suffix can be null. The function of the generator looks like the following :

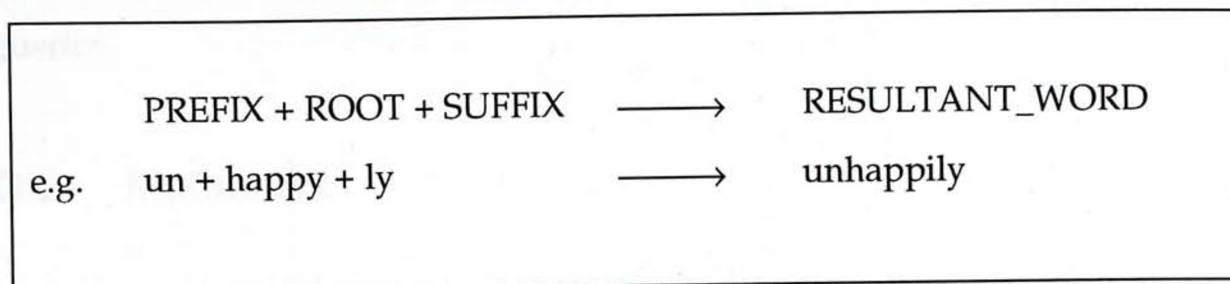


Figure G.1 The Function of Generator

The recognizer just does the opposite; it basically decomposes each word into root, prefix and suffix.

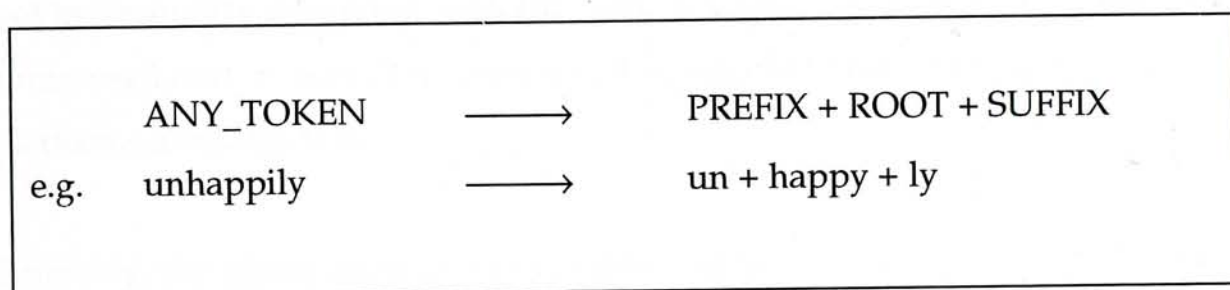


Figure G.2 The Function of Recognizer

The recognizer module was directly integrated in our system but the generator was not.

In ARMON, ENGLEX was chosen to work with PC-KIMMO. ENGLEX is an English lexicon specially designed for PC-KIMMO. It contains approximately 20000 lexical entities [Antw92b] consisting 11000 nouns, 4000 verbs and 3400 adjectives. According to the author, ENGLEX was tested with several running text. These included Lewis Carroll's "Alice's Adventures in Wonderland", and Herman Melville's "Moby Dick". The author claimed that all their tests produced good results [Antw92b]. All these claims built up our confidence to take PC-KIMMO as the parser and ENGLEX as the lexicon for building ARMON.

We took PC-KIMMO version 1.0.8 and ENGLEX version 1.0 to develop ARMON. In March 1995, a beta release of version 2 of both PC-KIMMO [Antw95a] and ENGLEX [Antw95b] became available from E.L. Antworth, the original author. However, there was not sufficient time to include these recent

releases in ARMON. The improvements in these new releases had been briefly discussed in Chapter 6. In spite of many shortcomings already discussed, we feel that the current version, ENGLEX 1.0, is a good lexicon for ARMON. It covers almost all the words appearing in our sentences used for captions and queries.

G.2 Metalbase

Metalbase [Jern92] was another software library originally written in C Language. Because this module is far less essential than PC-KIMMO, much fewer words had been and would be spent to discuss it. This toolkit was prepared to run on several platforms. In the beginning of the project, it could not be smoothly compiled with the GNU C Compiler on our machines. After some revisions, it was then successfully compiled and it ran on several Unix platforms available to us.

Similarly, the whole archive was available simply through anonymous FTP on common FTP sites, such as Sunsite and its mirrors [Jern92]. The documentation of this package was also kept on-line with the source codes as an archive. The available documentation provided useful and adequate information for application development like this.

The toolbox is a function library for simple relational-like database operations. It is useful for software developers who want to store and retrieve data at a manner similar to relational database. MetalBase defines its own syntax for writing schema for its database structure. It looks like a kind of DDL in RDBMS although it is still far from completeness.

Several small sample programs included in the package demonstrate what MetalBase can do. The author said that the samples were not much more than toys but they did achieve their purpose - to demonstrate that MetalBase is really working well on mid-sized databases. No further particulars of MetalBase will be discussed. Readers can find much more interesting and useful material in the on-line documentation of the package.

Appendix H Program Listing

```
################################################################################
#
#  Created by     : KPWONG
#  Path           : $PRJHOME/wrk/Makefile
#  Last Updated   : 12 April 1995
#  Function       : Top-level Makefile for ARMON on Linux, Solaris 2.3
#                   and HP/UX 9.05
#
################################################################################
#
PRJHOME=/users/kpwong/prj/
PRJDATA=$(PRJHOME)/data/
PRJBIN=$(PRJHOME)/bin/
PRJSRC=$(PRJHOME)/src/
PRJWRK=$(PRJHOME)/wrk/
#
KIMHOME=$(PRJHOME)/pckim
KIMSRC=$(KIMHOME)/kimmo/
LEXDIR=$(KIMHOME)/englex/
#
MBHOME=$(PRJHOME)/mb
MBSRC= $(MBHOME)/src
MBBIN= $(MBHOME)/bin
MBINC= $(MBHOME)/include
MBLIB= $(MBHOME)/lib
#
#OS = HPUX, SOLARIS or LINUX
## OSTYPE=SOLARIS
## OSTYPE=HPUX
OSTYPE=LINUX
#
MAKCMD=make PRJHOME=$(PRJHOME) OSTYPE=$(OSTYPE)


all:
        make mb
        make kim
        make prj

clean:
        (cd $(MBSRC);  $(MAKCMD) clean)
        (cd $(KIMSRC); $(MAKCMD) clean)
        (cd $(PRJSRC); $(MAKCMD) clean)
prj:
        make _par
        make _rcapt
        make _build

mb:
        (cd $(MBSRC); $(MAKCMD))

kim:
        (cd $(KIMSRC); $(MAKCMD))

_par:
        (cd $(PRJSRC); $(MAKCMD) par)

_rcapt:
        (cd $(PRJSRC); $(MAKCMD) rcapt)

newcapt:
        (cd $(PRJSRC); $(MAKCMD) newcapt)

_build:
        (cd $(MBSRC); $(MAKCMD) build)

TGZFILE = xtr_wrk.tgz
CUR=wrk/
TARED = $(CUR)*.s $(CUR)nl* $(CUR)q*
tgz:
        (cd ../; \
          cp -p $(TGZFILE) _$(TGZFILE); \
          gtar zcvf $(TGZFILE) $(TARED); \
        )
##        rm _$(TGZFILE)



################################################################################
#
#  Created by     : KPWONG
#  Path           : $PRJHOME/src/Makefile
#  Last Updated   : 12 April 1995
#  Function       : Source Makefile for ARMON on Linux, Solaris 2.3
#                   and HP/UX 9.05
#
################################################################################
#
```

```
PRJHOME=/users/kpwong/prj/
PRJDATA=$(PRJHOME)/data/
PRJBIN=$(PRJHOME)/bin/
PRJSRC=$(PRJHOME)/src/
PRJWRK=$(PRJHOME)/wrk/
#
MBHOME=$(PRJHOME)/mb
KIMHOME=$(PRJHOME)/pckim
#
KTXLIB=$(KIMHOME)/ktext/src/opaclib/
KIMLIB=$(KIMHOME)/kimmo/
KIMOBJDIR=$(KIMHOME)/kimmo/
KIMINCDIR=$(KIMHOME)/kimmo/
LEXDIR=$(KIMHOME)/englex/
#
MBSRC= $(MBHOME)/src/
MBBIN= $(MBHOME)/bin/
MBINC= $(MBHOME)/include/
MBLIB= $(MBHOME)/lib/

CC=gcc -g
## HPUX
## INSTCMD=install

## INSTCMD=install -o kpwong -g stf

## for Solaris, HPUX
INSTCMD=/etc/install

###CFLAGS= -I/usr/local/include
### CFLAGS= -I$(MBINC) -L$(MBLIB) -L$(KTXLIB)
#
#### CAPTDATA="$(MBHOME)/data"
### CFLAGS= -DCAPTDATA="/users/kpwong/mb_sol/mb/data" \
### -I$(MBINC) -I$(KIMINCDIR) -L$(MBLIB) -L$(KTXLIB)

CFLAGS= -I$(MBINC) -I$(KIMINCDIR) -L$(MBLIB) -D$(OSTYPE) \
	-DPRJHOME=\"$(PRJHOME)\"

all : rcapt capop hierop

clean:
	rm -f *.o testsent rcapt wcapt capop
	(cd $(MBSRC); make cleanbuild)

newcapt:
	(cd $(PRJWRK);\
	 $(MBBIN)/build capt ;\
	 $(INSTCMD) -m 644 capt.rel $(PRJDATA)/capt/ ;\
	)
##  be careful ***
	rm -r /users/kpwong/prj/data/capt/[0123456789]*

capop :   capop.o hierfunc.o capfunc.o
	$(CC) $(CFLAGS) -f -o capop capop.o hierfunc.o capfunc.o $(R_OBJS) -lmb
##	cp -p capop search
	$(INSTCMD) -m 755 capop $(PRJBIN)
	(cd $(PRJBIN); rm -f search; ln -sf capop search)

capfunc.o : capfunc.c capfunc.h hierfunc.h
	$(CC) $(CFLAGS) -c capfunc.c

capop.o : capop.c capt.h capfunc.h reldef.h Makefile
	$(CC) -c $(CFLAGS)  capop.c

rhier : rhier.o hierfunc.o
	$(CC) $(CFLAGS) -o rhier rhier.o hierfunc.o

hierop : hierop.o hierfunc.o
	$(CC) $(CFLAGS) -o hierop hierop.o hierfunc.o
	$(INSTCMD) -m 755 hierop $(PRJBIN)
	(cd $(PRJWRK); rm -f hierop; ln -sf $(PRJBIN)/hierop .)

hierop.o : hierop.c hierfunc.c hierfunc.h Makefile
	$(CC) $(CFLAGS) -c hierop.c

hierfunc.o : hierfunc.h hierfunc.c capfunc.h Makefile
	$(CC) -c $(CFLAGS)  hierfunc.c

rcapt : rcapt.o
	$(CC) $(CFLAGS) -o rcapt rcapt.o -lmb    ### -lmb -lopac
	$(INSTCMD) -m 755 rcapt $(PRJBIN)
	(cd $(PRJWRK); rm -f rcapt; ln -sf $(PRJBIN)/rcapt .)

rcapt.o : rcapt.c capt.h hierfunc.h hierfunc.c reldef.h
	$(CC) -c $(CFLAGS)  rcapt.c

build:
	(cd $(MBSRC); make cleanbuild; make)
	$(INSTCMD) -m 755 $(MBSRC)/build $(PRJBIN)

rifea : rifea.o
	$(CC) -f -o rifea rifea.o

rifea.o : rifea.c
	$(CC) -c rifea.c

wcapt : wcapt.o
	$(CC) $(CFLAGS) -f -o wcapt wcapt.o $(R_OBJS) ###  -lmb -lopac

wcapt.o : wcapt.c capt.h
	cc -c $(CFLAGS)  wcapt.c
```

```
TGZFILE = xtr_src.tgz
CUR=src/
TARED = $(CUR)?akefil* $(CUR)*.h $(CUR)*.c
tgz:
	(cd ../; \
	   cp -p $(TGZFILE) _$(TGZFILE); \
	   gtar zcvf $(TGZFILE) $(TARED); \
	)

R_OBJS = $(KIMOBJDIR)/rules.o $(KIMOBJDIR)/lexicon.o \
	$(KIMOBJDIR)/recogniz.o $(KIMOBJDIR)/pckfuncs.o

R_OBJS = $(KIMOBJDIR)/rules.o $(KIMOBJDIR)/lexicon.o \
	$(KIMOBJDIR)/recogniz.o $(KIMOBJDIR)/pckfuncs.o


###############################################################################
#  METALBASE 5.0
#
#  Released October 1st, 1992 by Huan-Ti [ richid@owlnet.rice.edu ]
#                                        [ t-richj@microsoft.com ]
#
#  Generic Makefile for 5.0 Library and Utilities
#
###############################################################################
#
#  Modified by   : KPWONG
#  Path          : $PRJHOME/mb/src/Makefile
#  Last Updated  : 12 April 1995
#  Purpose       : To be used with ARMON on Linux, Solaris 2.3
#                  and HP/UX 9.05
#
###############################################################################
#
## CFLAGS= -DSTRUCT_3
#    -DSTRUCT_1       -- Read lower for an explanation of these, and how to
#    -DSTRUCT_2       --    determine which is appropriate for your system.
#    -DSTRUCT_3
#    -DSTRUCT_4
#                     --  for Solarix, HPUX and Linux, STRUCT_1 and
#                         and STRUCT_3 are defined in stdinc.h
#                                              wkp   12.2.95
#    -DLONGARGS       -- To produce ansi-style prototypes ("void fn(int)")
#    -DNOSYNC         -- Removes calls to sync() and fsync(), and in-line _asm
#    -DNOVOIDPTR      -- To use char* instead of void* (automatic for COHERENT)
#    -DNOENCRYPT      -- To remove encryption crap from library and utilities
#    -DNEED_USHORT    -- If your compiler doesn't have ushort yet (COH again)
#    -DNEED_ULONG     -- If your compiler doesn't have ulong yet (most don't)
#    -DUNIX_LOCKS     -- To enable Unix-style locking
#    -DSIG_TYPE=void  -- void or int; needed only if you define UNIX_LOCKS
#    -DVI_EMU         -- To add vi emulation to input.c
#    -DMSDOS          -- MS-DOS users should define this if their CC doesn't.
#    -DHPUX
#    -DLINUX          -- Indicate OS favours
#    -DSOLARIS
#
# MBBIN=              -- Directory where executables should go
# MBINC=              -- Directory where include files should go
#                             include/*.h ln -s from src/*.h  wkp 24.2.95
# MBLIB=              -- Directory where libmb.a / mbase.lib should go
#
# LDOPTS=-f           -- To include floating point stuff for printf()
#
###############################################################################
#
# All users: Update the flags just below here FIRST (don't worry about
#            setting -DSTRUCT_?); then just type "make".  It will compile and
#            run struct/struct.exe, which will tell you how to determine how
#            -DSTRUCT_? should be set for your system.  Update this in the
#            Makefile and type "make install".  You may delete struct/
#            struct.exe after you've used it.
#
# DOS users: Try adding -DMSDOS to CFLAGS=; if you get a compiler error,
#            take it back out.  The code expects MSDOS to be defined for all
#            DOS compilers--most already set it, but some may not.
#
# Unix users: set -DUNIX_LOCKS to use flock() for file locking; otherwise,
#             MetalBase's inherent system will be used (which MAY cause
#             problems with code which does not exit properly, but which is
#             operationally identical).
#
###############################################################################

# defaults for Linux
## MBBIN=/usr/local/bin
## MBINC=/usr/local/include
## MBLIB=/usr/local/lib

## wkp
## KIMHOME=/home/kimmo/kimmo
## MBHOME=/home/mb
## How to pass from env variables $MB_HOME  ?

### --> PRJHOME=/users/kpwong/prj
PRJBIN=$(PRJHOME)/bin
KIMHOME=$(PRJHOME)/kimmo/kimmo
MBHOME=$(PRJHOME)/mb

MBBIN=$(MBHOME)/bin
```

```
MBINC=$(MBHOME)/include
MBLIB=$(MBHOME)/lib
### KIMMBINC=$(KIMHOME)
### KIMMBLIB=$(KIMHOME)

OBJ=.o
LIB=libmb.a
CURSES= -lncurses
NCURSES= -DNCURSES -I/usr/local/include
COPY=cp
CC=gcc
## HPUX, SOlaris
INSTCMD=/etc/install
## HPUX
## INSTCMD=install
# Solaris /usr/ucb/install, Linux
### INSTCMD=install -o kpwong -g stf
LDOPTS= -s

## -DSTRUCT_3 for Linux
## -DSTRUCT_1 for Solarix and HPUX


### OSTYPE passed from Makefile
OSFLAG= -D$(OSTYPE)

CFLAGS= -Wall -O $(OSFLAG) -DSIG_TYPE=void -DNOENCRYPT \
        -DVI_EMU -DUNIX_LOCKS  -I.

## HPUX needs "-DNEED_ULONG"
##         -DVI_EMU -DUNIX_LOCKS -DNEED_ULONG -I.

## for with NCURSES
## CFLAGS= -Wall -O $(NCURSES) -DSTRUCT_3 -DSIG_TYPE=void -DUSE_CURKEY -DNOENCRYPT -DVI_EMU -DLONGARGS -
DUNIX_LOCKS -I.

BLAST   = blast
BUILD   = build
FORM    = form
MBCONV  = mbconv
REPORT  = report
VR      = vr
LIBRARY = libmb.a

HEADERS=stdinc.h mbase.h
TARGETS=$(BLAST) $(BUILD) $(FORM) $(MBCONV) $(REPORT) $(SAMPLE) $(VR)

ARCHIVE = ar rv $(LIBRARY)
RANLIB  = ranlib $(LIBRARY)


## Rules created for ARMON            wkp, 2-JAN-95

EXE=

mbforprj: struct$(EXE) $(HEADERS)
        make $(BUILD)
        make $(LIBRARY)
        make instforprj

instforprj:
        $(INSTCMD) -m 0755 $(BUILD) $(MBBIN)
        $(INSTCMD) -m 0755 $(BUILD) $(PRJBIN)
##      $(INSTCMD) -m 0644 mbase.h stdinc.h $(MBINC)
        $(INSTCMD) -m 0644 $(LIBRARY) $(MBLIB)

cleanbuild:
        rm -f $(BUILD).o $(BUILD)$(EXE)

all: struct$(EXE) $(HEADERS) $(TARGETS)

install : all
        $(INSTCMD) -m 0755 $(TARGETS) $(MBBIN)
##      $(INSTCMD) -m 0644 mbase.h stdinc.h $(MBINC)
        $(INSTCMD) -m 0644 $(LIBRARY) $(MBLIB)

struct$(EXE) : struct$(OBJ)
        $(CC) -o $@ struct$(OBJ)
        @./struct
        @echo Now update the Makefile and make install

clean:
        rm -f *.o $(TARGETS) $(LIBRARY)


$(BLAST)    : blast$(OBJ)
        $(CC) -o $@ blast$(OBJ)

$(BUILD)    : build$(OBJ) $(LIBRARY)
        $(CC) $(LDOPTS) -o $@ build$(OBJ) $(LIBRARY)
        $(INSTCMD) -m 0755 $(BUILD) $(PRJBIN)

$(FORM)     : form$(OBJ) form_wr$(OBJ) $(LIBRARY)
        $(CC) $(LDOPTS) -o $@ form$(OBJ) form_wr$(OBJ) $(LIBRARY)

$(MBCONV)   : mbconv$(OBJ) $(LIBRARY)
        $(CC) $(LDOPTS) -o $@ mbconv$(OBJ) $(LIBRARY)

$(REPORT)   : report$(OBJ) $(LIBRARY)
        $(CC) $(LDOPTS) -o $@ report$(OBJ) $(LIBRARY)

$(VR)       : vr$(OBJ) $(LIBRARY)
        $(CC) $(LDOPTS) -o $@ vr$(OBJ) $(LIBRARY) $(CURSES)
```

```
## $(LIBRARY_0) : entry$(OBJ) lock$(OBJ) input$(OBJ) mbase$(OBJ)\
##             parse$(OBJ) timedate$(OBJ) util1$(OBJ)\
##             util2$(OBJ) cache$(OBJ) create$(OBJ)
##      $(ARCHIVE) entry$(OBJ) lock$(OBJ) input$(OBJ) mbase$(OBJ)\
##              parse$(OBJ) timedate$(OBJ)\
##              util1$(OBJ) util2$(OBJ) cache$(OBJ) create$(OBJ)
##      $(RANLIB)
##
$(LIBRARY) : lock$(OBJ) mbase$(OBJ)\
            parse$(OBJ)  timedate$(OBJ) util1$(OBJ)\
            util2$(OBJ) cache$(OBJ) create$(OBJ)
        $(ARCHIVE) lock$(OBJ) mbase$(OBJ)\
            parse$(OBJ) timedate$(OBJ) util1$(OBJ) \
            util2$(OBJ) cache$(OBJ) create$(OBJ)
        $(RANLIB)
```

```
# UNIX makefile for PC-KIMMO
#                                          Steve McConnel,  14-Jul-90
##########################################################################
#
#  Modified by    : KPWONG
#  Path           : $PRJHOME/pckim/kimmo/Makefile
#  Last Updated   : 12 April 1995
#  Purpose        : To be used with ARMON on Linux, Solaris 2.3
#                   and HP/UX 9.05
#
#--------------------------------------------------------------------------
# choose your system by the CFLAGS definition
#
#       System V:
#CFLAGS=-O -DUNIX -DUSG
#
#       BSD or SunOS:
## CFLAGS=-O -DUNIX -DBSD
#
#       ULTRIX:
## CFLAGS=-O -DUNIX -DBSD -DULTRIX
#
#################
# choose your compiler by the CC definition (CC=cc is standard)
#
## CC=cc
##OK for Solaris and Linux:  CC=gcc -g -Wall -DUSG -O

## -DHPUX

###  OSTYPE passed from Makefile
OSFLAG=-D$(OSTYPE)

CC=gcc -g -Wall -DUSG -DUNIX $(OSFLAG) -O
##CC=gcc -g -Wall -DUNIX -DUSG -DHPUX -O

OBJS=pckimmo.o usercmd.o userfunc.o\
 lexicon.o rules.o generate.o recogniz.o pckfuncs.o

SOURCES=pckimmo.c usercmd.c userfunc.c\
 lexicon.c rules.c generate.c recogniz.c pckfuncs.c

pckimmo: $(OBJS)
        $(CC) $(CFLAGS) -o pckimmo $(OBJS)

pckimmo.lint: $(SOURCES)
        lint $(LINTFLAGS) $(SOURCES) >pckimmo.lint

pckimmo.calls: $(SOURCES)
        calls $(CALLFLAGS) $(SOURCES) >pckimmo.calls

pckimmo.o:  pckimmo.h version.h
usercmd.o:  pckimmo.h
userfunc.o: pckimmo.h

lexicon.o:  pckimmo.h
rules.o:    pckimmo.h
recogniz.o: pckimmo.h
generate.o: pckimmo.h
pckfuncs.o: pckimmo.h

######################################################################
#  simple test programs to check out modularity
#
G_OBJS = g.o rules.o generate.o pckfuncs.o

g: $(G_OBJS)
        $(CC) $(CFLAGS) -o gener $(G_OBJS)

g.o: g.c pckimmo.h

R0_OBJS = r0.o rules.o lexicon.o recogniz.o pckfuncs.o

r0: $(R0_OBJS)
        $(CC) $(CFLAGS) -o r0 $(R0_OBJS)
```

```
r0.o: r0.c pckimmo.h

R_OBJS = r.o rules.o lexicon.o recogniz.o pckfuncs.o

r: $(R_OBJS)
	$(CC) $(CFLAGS) -o recog $(R_OBJS)

r.o: r.c pckimmo.h

all: pckimmo g r

clean:
	rm -f pckimmo g r *.o pckimmo.lint pckimmo.calls

TGZFILE = kim_kp.tgz
tgz   : *.c *.h
	tar zcvf $(TGZFILE) ?akefile* *.h *.c
	mcopy $(TGZFILE) c:
	mcopy $(TGZFILE) d:
```

```
################################################################################
#
#  Created by     : KPWONG
#  Path           : $PRJHOME/wrk/capt.s
#  Last Updated   : 8 March 1995
#  Function       : Definition of caption database, written
#                   in Metalbase Schema, manupulated with
#                   MetalBase tollbox
#
################################################################################
#
relation capt

field capid    type string length 12; # caption ID
field agent    type string length 18; # the agent of action
field action   type string length 18; # the main action of the sentence
field patient  type string length 18; # the patient of action

index ix_capid   on capid   with duplicates;
index ix_agent   on agent   with duplicates;
index ix_action  on action  with duplicates;
index ix_patient on patient with duplicates;
## index ix_patient on patient with duplicates;

end
```

```
/*****************************************************************
 *      reldef.h - Header file for defining
 *                 paths and fields of CAPT on
 *                 Metalbase
 *****************************************************************
 *
 *      Written       : Kit-pui Wong
 *
 *      Last Updated : 21 March 1995
 *
 *****************************************************************
 */
#define CAPT_REL_FILE "capt.rel"
#define CAPT_HFILE "capt.h"
#define CAPT_FLD_LENS capt_str_len
#define CAPT_STR_DEF  capt_str

#include CAPT_HFILE            /* Created during "% build capt.s"  */
```

```
/*****************************************************************
 *      capt.h - Caption operations header file
 *****************************************************************
 *
 *      Written       : Kit-pui Wong
 *
 *      Last Updated : 23 May 1995
 *
 *****************************************************************
 */

#ifndef CAPT_H
#define CAPT_H
```

```
/*
 * This file was created by MetalBase version 5.0 to reflect the structure
 * of the relation "capt".
 *
 * MetalBase 5.0 released October 1st, 1992 by virtual!richid@owlnet.rice.edu
 *
 */

typedef struct
 { char     capid[12];              /* field capid type string length 12    */
   char     agent[18];              /* field agent type string length 18    */
   char     action[18];             /* field action type string length 18   */
   char     patient[18];            /* field patient type string length 18  */
   } capt_str;

/*====== Added by KP Wong (in build.c),    25Aug94 ======*/
/*== unsigned xx_str_len[] = {ArySize, len0, len1.....} ==*/
#ifdef NOW_DEF_LEN
  unsigned capt_str_len[] = {4, 12, 18, 18, 18};
#else
  extern unsigned capt_str_len[];
#endif

#ifndef MODULE
   capt_str capt_rec;
#else
   extern capt_str capt_rec;
#endif

#endif
```

```
/*****************************************************************
 *      capfunc.h - Header files for caption operations
 *****************************************************************
 *
 *      Written by      : Kit-pui Wong
 *
 *      Last updated    : 21 March 1995
 *
 *****************************************************************
 */

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


/**** remember to define char _buf_[127]    ****/
/* wherei is rules file and and lexicon file (ENGLEX)    */
#define RUL_FILE ((char *) strcat(strcpy(_buf_, PRJHOME), "/pckim/englex/english.rul"))
#define LEX_FILE ((char *) strcat(strcpy(_buf_, PRJHOME), "/pckim/englex/english.lex"))

#include <time.h>
#include <stdio.h>

#ifndef CAPTPATH
#define CAPTPATH "/users/kpwong/prj/data/capt/"
#endif

#define IFEA_PATH "/users/kpwong/prj/data/impfea/"
#define FEA_RUL   "/users/kpwong/prj/data/fea.rul"

#define ATTR_AG "attrag"
#define ATTR_AC "attrac"
#define ATTR_PA "attrpa"

#ifndef ATTRNAME
#define ATTRNAME 1
char attrname[4][10] = {"", ATTR_AG, ATTR_AC, ATTR_PA};
#endif

#define QID  1

#define BSIZE 200

/*==== Parts of Speech from KIMMO  =======*/
#define VERB 1
#define NOUN 2
#define  ADJ 3
#define  ADV 4
#define   PP 5
#define  DET 6
#define  UNKNOWN -1

/*======== Semantic role ========*/
#define AGENT 1
#define ACTION 2
#define PATIENT 3
#define EFEA_AGENT 4
#define EFEA_ACTION 5
#define EFEA_PATIENT 6
#define UNKNOWN -1
#define EOL -1
```

```
#define YES 1
#define NO 0

#include <mbase.h>
#include "reldef.h"    /* Created during "% build *.s"  */

/*== defined in PC-KIMMO ==*/

LANGUAGE Lang;
long   elaparr[79];
extern char *skipwhite();
extern RESULT *recognizer();
```

```
/****************************************************************
 *      hierfunc.h - Header file for hierarchy operations
 ****************************************************************
 *
 *      Written      : Kit-pui Wong
 *
 *      Last Updated : 21 March 1995
 *
 ****************************************************************
 */

/** where is table of type hierarchy  **/
#define HFILE ((char *) strcat(strcpy(_buf_, PRJHOME), "/data/hier.tab"))

#define FAILED   0
#define SUCCESS  1
#define INEXIST  0
#define EXIST    1
#define MAXSIBLS 50
#define MAXCHILDN 50

/*   The seq of field in hier table  */
#define FIELD_NODEID 1
#define FIELD_NODENAME 2

typedef struct
{  char     node[30];
   char     *next;
} StringListStr;
```

```
/****************************************************************
 *      hierop.c - HIER operations in ARMON
 ****************************************************************
 *
 *      Written      : Kit-pui Wong
 *
 *      Last Updated : 21 March 1995
 *
 ****************************************************************
 */
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include "hierfunc.h"
#include "capfunc.h"

extern long FindNodeId(), FindAnces(), dist();
extern char *FindNodeName();
extern int GetAllChildn();
extern int GetImChildn();
extern int GetAllSibls();

void main();
long  GetImAnces();
int CheckExist();
int CheckArg();
int insHier();
int AskMember();

char _buf_[200];
char hfile[200];
char AssumedRoot[]="phy_obj";

#define INSCODE 'i'
/****
#define FAILED   0
#define SUCCESS  1
#define INEXIST  0
#define EXIST    1
#define MAXSIBLS 50
```

```
#define MAXCHILDN 50
***/

/*
 *  main starts
 */
void main  (argc, argv)
int   argc;
char **argv;
{
  int res;
  char opcode;
  char operand[30];
  strcpy(hfile, HFILE);
  if ((res = CheckArg(argc, argv, &opcode, operand)) != SUCCESS) {
    exit(0);
  };
  printf ("opcode=%c  operand=%s\n", opcode, operand);
  switch (opcode) {
    case INSCODE : insHier(hfile, operand);
                   break;
    default :   fprintf (stderr, "Invalid Opcode: %c\n", opcode);
  };
  exit(0);
};

/*
 *   CheckArg() - Check Arguments
 */
int CheckArg(argc, argv, opcode, operand)
int argc; char **argv; char *opcode; char *operand;
{
  char tempOpcode[30];
  if (argc == 3) {
    strcpy (tempOpcode, argv[1]);
    strcpy (operand, argv[2]);
  } else {
    fprintf (stderr, "Syntax: %s [i|l] <newentity>\n", argv[0]);
    return FAILED;
  };
  if (strlen(tempOpcode) != 1) {
    fprintf (stderr, "Invalid Opcode: %s\n", tempOpcode);
    return FAILED;
  } else {
    *opcode = tempOpcode[0];
  };
  return SUCCESS;
};

/*
 *   insHier() - Insert to HIER
 */
int insHier(hfile, operand)
char *hfile, *operand;
{
  int i;
  long newid;
  char tempstr[20];
  int n, lastdigit;
  char aChar='?';
  char InvalidSet[20]="";
  char ChildnBuff[MAXCHILDN][20];
  char ParentName[30];
  long ImmedParent, idtemp;
  int IsExist;
  IsExist = CheckExist(hfile, FIELD_NODENAME, operand);
  /**  printf ("IsExist=%u\n", IsExist);  */
  if (IsExist == EXIST) {
    printf ("Error: candidate \"%s\" exists\n", operand);
    return FAILED;
  };
  ImmedParent = GetImAnces(hfile, AssumedRoot, operand);
  FindNodeName(ParentName, hfile, ImmedParent);
  printf ("Immediate parent of \"%s\" is \"%s\", id:%ld\n",
          operand, ParentName, ImmedParent);
  memset(ChildnBuff, 0, sizeof(ChildnBuff));
  n = GetImChildn(hfile, MAXCHILDN, ChildnBuff,
                  sizeof(ChildnBuff[0]),
                  ParentName);
  printf (">> Entity %s has %u immediate child(ren) :\n", ParentName, n);
  for(i=0; (i<MAXCHILDN) && (ChildnBuff[i][0]); i++) {
    idtemp = FindNodeId(hfile, ChildnBuff[i]);
    printf ("%ld %s\n", idtemp, ChildnBuff[i]);
    lastdigit = idtemp - (idtemp / 10 ) * 10;
    sprintf(InvalidSet, "%s%d", InvalidSet, lastdigit);
            /*  careful: recursive */
  };
  printf("New entity %s should have a new id of the form \"%ld_\"\n",
         operand, ImmedParent);
  printf("Please enter the last digit of the new id\n");
  while ((aChar < '0') ||  (aChar > '9')
        || strchr(InvalidSet, aChar) ) {
    aChar = AskDigit("Please enter a digit : ");
  };
  newid = (10 * ImmedParent) + (aChar - '0');
  printf ("The id of %s is %ld\n", operand, newid);
};

/*
 *  GetImAnces() - Get Immediate Ancestor of a node
 */
long GetImAnces(hfile, localroot, target)
```

```
char *hfile; char *localroot, *target;
{
  char ChildnBuff[MAXCHILDN][20];
  long ltemp;
  int i, nChildn;
  memset(ChildnBuff, 0, sizeof(ChildnBuff));
  nChildn =
        GetImChildn(hfile, MAXCHILDN, ChildnBuff,
                    sizeof(ChildnBuff[0]), localroot);
  if (nChildn==0) {
    ltemp = FindNodeId(hfile, localroot);
    return ltemp;
  };
  /*** printf("nChildn=%u\n", n);  ***/
  for(i=0; i<nChildn; i++) {
    if (AskMember (target, ChildnBuff[i]) == YES) {
      ltemp = GetImAnces(hfile, ChildnBuff[i], target);
      return ltemp;
    };
  }
  ltemp = FindNodeId(hfile, localroot);
  return ltemp;
  /*  *res = (restemp > *res) ? restemp : *res; */
};

/*
 *   AskMember() - Whether two nodes in membership relation
 */
int AskMember (lownode, upnode)
char *lownode, *upnode;
{
  char qn[200];
  int ans=0;
  sprintf (qn, "Is %s a member of %s ?", lownode, upnode);
  ans = AskYesNo (qn);
  return ans;
};
```

```
/****************************************************************
 *      hierfunc.c - Functions library for type hierarchy
 ****************************************************************
 *
 *      Written      : Kit-pui Wong
 *
 *      Last Updated : 21 March 1995
 *
 ****************************************************************
 */
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include "hierfunc.h"
#define  ATTRNAME
#include "capfunc.h"

extern int AskMember();

char *FindNodeName();
long FindNodeId();
long FindAnces();
void FindAllAnces();
void AllIfea();
long dist();
void XtractIfea();
float SimIFea();
float SimAllFea();
void StripNL();
int  isBelongTo();
int CountLines();
int CountIntersect();
int AskYesNo();
void ReadAllFea();
void ReadEfea();

char _buf_[100];


/*
 *   FindNodeName() - Find the nodename from a nodid
 */
char *FindNodeName(ent, hfile, node)
char *ent; char *hfile; long node;
{
#define BSIZE 90
  char sh[80];
  char buf[BSIZE], buf1[BSIZE];
  long id = 0;
  FILE *fp;
  ent[0]='\0';
  sprintf (sh, "grep %ld %s", node, hfile);
 /*  printf ("sh=%s\n", sh);  */
  if ((fp = popen(sh,"r")) != NULL) {
    while (fgets (buf, BSIZE, fp) != NULL) {
      sscanf(buf, "%ld%s", &id, buf1);
```

```
      if (id == node) {
        strcpy (ent, buf1);
        break;
      };
    };
  };
  pclose(fp);
  return ent;
}

/*
 *   FindNodeId() - Find the node id from a node name
 */
long FindNodeId(hfile, node)
char *hfile; char *node;
{
#define BSIZE 90
  char sh[80];
  char buf[BSIZE], ent[BSIZE];
  long id =0;
  long idWork=0;
  FILE *fp;
  sprintf (sh, "grep %s %s", node, hfile);
/**  printf ("sh=%s\n", sh);  **/
  if ((fp = popen(sh,"r")) != NULL) {
    while (fgets (buf, BSIZE, fp) != NULL) {
      sscanf(buf, "%ld%s", &idWork, ent);
      if (!strcmp(ent, node)) {
        id = idWork;
        break;
      };
    };
  };
  pclose(fp);
  return id;
}

/*
 *   FindAnces() - Find the ancesotr of a node by node id
 */
long FindAnces(hfile, nodeid)
char *hfile; long nodeid;
{
   char buf[15];
   if (nodeid <= 9) {
     return 0;
   };
   sprintf(buf, "%ld", nodeid);
   memset(strchr(buf, 0)-1, 0, 1);
   return atol(buf);
};

/*
 *   isBelongTo() - Whether the lower node is a sub-class
 *                  of the other
 */
int  isBelongTo(hfile, lower, upper)
char hfile[], lower[], upper[];
{
  char lowid[20], upid[20];
  int yesno;
  sprintf(lowid, "%ld", FindNodeId(hfile, lower));
  sprintf(upid, "%ld", FindNodeId(hfile, upper));
  yesno = ! strncmp(lowid, upid, strlen(upid));
  return yesno;
}

/*
 *   FindAllAnces() - Find all ancestor of a node
 */
void FindAllAnces (hfile, nodeid)
char *hfile; long nodeid;
{
   while (nodeid > 9) {
     nodeid=FindAnces(hfile,nodeid);
     printf ("nodeid=%ld\n", nodeid);
   };
};


/*
 *   AllIfea() - Find all impfea of a node
 */
void AllIfea(hfile, nodeid, outfile)
char *hfile; long nodeid; char *outfile;
{
   char name[200];
   char sh[40];
   sprintf(sh, "rm %s", outfile);
   system(sh);
   printf ("in AllIfea\n");
   printf("** nodeid=[%ld]\n", nodeid);
   for (; nodeid > 9; nodeid=FindAnces(hfile,nodeid)) {
     FindNodeName(name, hfile, nodeid);
     printf ("name=[%s]\n", name);
     XtractIfea(name, outfile);
   };
   printf ("exit AllIfea\n");
};

/*
 *   dist() - the dist function of two nodes
```

```
 */
long dist(id1, id2)
long id1, id2;
{
  char node1[20], node2[20], res[20];
  int i=0;
  memset(node1, 0, sizeof node1);
  memset(node2, 0, sizeof node2);
  sprintf(node1, "%ld", id1);
  sprintf(node2, "%ld", id2);
  printf ("%s %s\n", node1, node2);
  for (i=0; node1[i]; i++) {
    res[i] = '0' + ((node1[i]-'0') != (node2[i]-'0'));
  };
  for (; node2[i]; i++) {
    res[i]='0';
  };
  res[i]='\0';
/*  printf ("res=%s", res);  */
  return atol(res);
};


/*
 *   XtractIfea() - Extract the impfea of an entity
 */
void XtractIfea(entity, outfile)
char *entity; char *outfile;
{
#define BSIZE 90
  char sh[80];
  char buf[BSIZE], slot[30], val[30];
  FILE *pfp, *fp;
 /* char path[]="./impfea/"; */
  char path[]=IFEA_PATH;
/*
  sprintf (sh, "cat %s%s >> %s", path, entity, outfile);
  system (sh);
*/
  /*  printf ("** in XtractIfea\n"); */
  if (!(fp = fopen(outfile,"a"))) {
    printf("File open error: %s\n", outfile);
    exit(0);
  };
  sprintf (sh, "cat %s%s", path, entity);
  printf ("sh=%s\n", sh);
  if ((pfp = popen(sh,"r")) != NULL) {
      /*    printf(".\n");  */
    while (fgets (buf, BSIZE, pfp) != NULL) {
      /*       printf(" #\n");   */
      sscanf(buf, "%s%s", slot, val);
     /*  printf("slot=[%s]\tval=[%s]\n", slot, val);  */
      fprintf(fp, "%s\t%s\n", slot, val);
    };
  };
  pclose(pfp);
  fclose(fp);
};


 /*
  *   ReadAllFea() - Read all features in a caption
  */
 void ReadAllFea(tempfn, hfile, capid, entity, attrtype)
 char *tempfn; char *hfile; int capid; char *entity; char *attrtype;
 {
   printf ("in ReadAllFea\n");
   if (entity[0]) {
    /*  NodeId = FindNodeId(hfile, entity); */
      AllIfea (hfile, FindNodeId(hfile, entity), tempfn);
   };
   ReadEfea (tempfn, capid, attrtype);
 };

 /*
  *   ReadEFea() - Read expfea in a caption
  */
 void ReadEfea(tempfn, captid, attrtype)
   char *tempfn; int captid; char *attrtype;
   {
     char sh[120];
         /*  append file */
    printf ("in ReadEfea\n");
     sprintf(sh, "cat %s/%d/%s >> %s",
                 CAPTPATH, captid, attrtype, tempfn);
     printf ("=====>sh=%s\n", sh);
     system (sh);
   };

  /*
   *   SimAllFea() - Calc the similarity between the query
   *                 and a caption
   */
  float SimAllFea(capidQ, capidC, entq, entc, attrtype)
  int capidQ; int capidC; char *entq;
  char *entc; char *attrtype;
  {
    char tmpq[80], tmpc[80];
    char _buf_[127];
    int nMatched = 0, nQ = 0;
    sprintf(tmpq, "/tmp/%d%s.fea", capidQ, entq);
    sprintf(tmpc, "/tmp/%d%s.fea", capidC, entc);
    printf ("tmpq->%s  tmpc->%s\n", tmpq, tmpc);
```

```
  ReadAllFea(tmpq, HFILE, capidQ, entq, attrtype);
  ReadAllFea(tmpc, HFILE, capidC, entc, attrtype);
  nMatched = CountIntersect(tmpq, tmpc);
  nQ = CountLines (tmpq);
  printf ("nMatched=%i nQ=%i div=%f\n", nMatched, nQ,
            (float) nMatched/ (float) nQ);
  return ((float) nMatched / (float) nQ);
};


/*
 *   SimIFea() - Similarity evaluation from impfea
 */
float SimIFea(entq, entc, capno)
char *entq; char *entc; char *capno;
{
  char tmpq[40], tmpc[40];
  char _buf_[127];
  int nMatched = 0, nQ = 0;
  sprintf(tmpq, "/tmp/%s.ife", entq);
  sprintf(tmpc, "/tmp/%s.ife", entc);
  printf ("-->%s\n", tmpc);
  AllIfea(HFILE, FindNodeId(HFILE, entq), tmpq);
  AllIfea(HFILE, FindNodeId(HFILE, entc), tmpc);
  nMatched = CountIntersect(tmpq, tmpc);
  nQ = CountLines (tmpq);
  printf ("nMatched=%i nQ=%i div=%f\n", nMatched, nQ,
            (float) nMatched/ (float) nQ);
  return ((float) nMatched / (float) nQ);
};


/*
 *   CountIntersect() - Count the elements of intersection
 *                      of the feature set in query and
 *                      caption
 */
int CountIntersect(fnQ, fnC)
char *fnQ, *fnC;
{
  int nI = 0;
  FILE *pfp;
  char sh[150];
  printf ("in countIntersect, nI=[%d]\n", nI);
  sprintf (sh, "fgrep -f %s %s | wc -l", fnQ, fnC);
  printf ("sh=[%s]\n", sh);
  if ((pfp = popen(sh,"r")) != NULL) {
    fscanf (pfp, "%d", &nI);
  };
  pclose(pfp);
  printf ("in countIntersect, nI=[%d]\n", nI);
  return nI;
}


/*
 *   StripNL() - Strip the trailing "\n"
 */
void StripNL(s)
char *s;
{
    memset(strchr(s, '\n'), 0, 1);
};


/*
 *   CountLines() -  How many lines in file
 */
int CountLines(fn)
char *fn;
{
  FILE *fp;
  int i;
  char *p;
  char buff[BSIZE+10];
  if ((fp = fopen(fn,"r")) == NULL) {
    return 0;
  };
  p = fgets(buff, BSIZE, fp);
  for (i=0; p != NULL; i++) {
    p = fgets(buff, BSIZE, fp);
  };
  return i;
};


/*
 *   GetAllChildren() - Get all childern of a node, put
 *                      result into a list
 */
int GetAllChildn(hfile, max, outlist, cellsize, mother)
char *hfile; int max; char *outlist; int cellsize; char *mother;
{
  char sh[200], cmotherid[20], buf[BSIZE], namebuf[BSIZE];
  FILE *pfp;
  long motherid, idwork;
  int buffidx = 0, lenmotherid;
  char *ptr;
  motherid=FindNodeId(hfile, mother);
  sprintf (cmotherid, "%ld", motherid);
  lenmotherid=strlen(cmotherid);
  sprintf(sh, "cut -f1 %s | cut -f1 -d \" \" | grep %ld", hfile, motherid);
  if ((pfp = popen(sh,"r")) != NULL) {
```

```
    while (fgets (buf, BSIZE, pfp) != NULL) {
      memset(strchr(buf, '\n'), 0, 1);    /* delete the trailing "\n"  */
      /* printf ("%s =?= %s len=%ld  ans: ", buf,cmotherid, lenmotherid); */
      if ((! strncmp(buf, cmotherid, lenmotherid)) &&
          (strlen(buf)!=lenmotherid) ) {
        /*  printf ("Yes\n");  */
        ptr = outlist + (cellsize*buffidx);
        strcpy (ptr, FindNodeName(namebuf, hfile, atol(buf)));
        /*  printf ("%u  %p  %s\n", cellsize, ptr, ptr); */
        buffidx++;
      };
    };
  };
  return buffidx;
};

/*
 *   GetImChildn() - Get immediate children of a node
 */
int GetImChildn(hfile, max, outlist, cellsize, mother)
char *hfile; int max; char *outlist; int cellsize; char *mother;
{
  char sh[200], cmotherid[20], buf[BSIZE], namebuf[BSIZE];
  FILE *pfp;
  long motherid, idwork;
  int buffidx = 0, lenmotherid;
  char *ptr;
  motherid=FindNodeId(hfile, mother);
  sprintf (cmotherid, "%u", motherid);
  lenmotherid=strlen(cmotherid);
  sprintf(sh, "cut -f1 %s | cut -f1 -d \" \" | grep %ld", hfile, motherid);
  if ((pfp = popen(sh,"r")) != NULL) {
    while (fgets (buf, BSIZE, pfp) != NULL) {
      memset(strchr(buf, '\n'), 0, 1);    /* delete the trailing "\n"  */
      /* printf ("%s =?= %s len=%u  ans: ", buf,cmotherid, lenmotherid); */
      if ((! strncmp(buf, cmotherid, lenmotherid)) &&
          (strlen(buf)==lenmotherid+1 )) {
        /*  printf ("Yes\n");  */
        ptr = outlist + (cellsize*buffidx);
        strcpy (ptr, FindNodeName(namebuf, hfile, atol(buf)));
        /*  printf ("%u  %p  %s\n", cellsize, ptr, ptr); */
        buffidx++;
      };
    };
  };
  return buffidx;
};

/*
 *   GetAllSibls() - Get all siblings, put result into a list
 */
int GetAllSibls(hfile, max, outlist, cellsize, currnode)
char *hfile; int max; char *outlist; int cellsize; char *currnode;
{
  char sh[200], cmotherid[20], ccurrid[20],
       buf[BSIZE], namebuf[BSIZE];
  FILE *pfp;
  long currid, motherid, idwork;
  int buffidx = 0, lenmotherid, lencurrid;
  char *ptr;
  currid=FindNodeId(hfile, currnode);
  motherid=FindAnces(hfile, currid);
  sprintf (cmotherid, "%u", motherid);
  sprintf (ccurrid, "%u", currid);
  lenmotherid=strlen(cmotherid);
  lencurrid=strlen(ccurrid);
  sprintf(sh, "cut  -f1 %s | cut -f1 -d \" \" | grep %ld", hfile, motherid);
  printf("sh=%s\n", sh);
  if ((pfp = popen(sh,"r")) != NULL) {
    while (fgets (buf, BSIZE, pfp) != NULL) {
      memset(strchr(buf, '\n'), 0, 1);    /* delete the trailing "\n"  */
      /* printf ("%s =?= %s len=%u  ans: ", buf,cmotherid, lenmotherid); */
      if ( (! strncmp(buf, cmotherid, lenmotherid)) &&
           (strcmp (buf, ccurrid)) &&
           (strlen(buf)==lencurrid) ) {
        /*  printf ("Yes\n");  */
        ptr = outlist + (cellsize*buffidx);
        strcpy (ptr, FindNodeName(namebuf, hfile, atol(buf)));
        /*  printf ("%u  %p  %s\n", cellsize, ptr, ptr); */
        buffidx++;
      };
    };
  };
  return buffidx;
};

/*
 *   AskYesNo() - Ask user yes or no to a question
 */
int extern AskYesNo(qn)
char *qn;
{
  char ansbuff[100]="X";
  char choice='X';
  while (! ((choice == 'Y') || (choice == 'N'))) {
    printf ("%s", qn); printf ("\n");
/*    gets (ansbuff); */
    scanf ("%s", ansbuff);
    choice = toupper(ansbuff[0]);
  };
  printf ("Your choice is [%c]\n", choice);
  return (choice=='Y' ? YES : NO);
};
```

```
/*
 *  CheckExist() - Check something in the HIER file
 */
int CheckExist(hfile, fieldno, candidate)
char *hfile; int fieldno; char *candidate;
{
   char sh[200], buf[BSIZE];
   FILE *pfp;
#ifdef HPUX
   sprintf (sh, "cut -f%d %s | grep -x -i %s",
   fieldno, hfile, candidate);
#else
   sprintf (sh, "cut -f%d %s | grep -w -i %s",
   fieldno, hfile, candidate);
#endif
   /** printf("sh--->%s\n", sh);  */
   if ((pfp = popen(sh,"r")) != NULL) {
     if (fgets (buf, BSIZE, pfp) != NULL) {
          printf("exist\n");
          pclose(pfp);
          return EXIST;
     };
   };
   pclose(pfp);
   return INEXIST;
};

/*
 *   AskDigit() - Ask for a digit from user
 */
int AskDigit(qn)
char *qn;
{
  char ansbuff[100]="X";
  char choice='X';
  while ((choice < '0') || (choice > '9')) {
    printf ("%s", qn); printf ("\n");
/*    gets (ansbuff); */
    scanf ("%s", ansbuff);
    choice = ansbuff[0];
  };
  printf ("Your choice is [%c]\n", choice);
  return (choice);
};
```

```
/****************************************************************
 *      capfunc.c - Function Library for
 *                  caption operations in ARMON
 ****************************************************************
 *
 *      Written      : Kit-pui Wong
 *
 *      Last Updated : 23 May 1995
 *
 ****************************************************************
 */

#define  NOW_DEF_LEN 1
#include "capfunc.h"
#include "hierfunc.h"

#define MAX_GROSS 50
#define MAX_SIM_ARY 150
char RootTab[8][10][20];
char ResTab[8][10][20];
char OrigTab[8][10][20];
int  RoleList[20];
int  TagTab[8][10];
char* attrib[4][10];  /* attrib table */
int  Matched[4][MAX_GROSS];
float SimCap[MAX_SIM_ARY];
int  SimCapid[MAX_SIM_ARY];
int  SimCapPtr = 0;

extern void StripNL();
extern char *hfile;
char _buf_[127];

int GenRnd();
char *mystrtok(), *GetOneFld(), *PutOneFld(), *GetRoot();
void extrCaps(), bubble(), swapint();
void ReadAllFea();
void EachAttrib();
void WriteEfea();
float SimAllRoles();

extern float SimAllFea();

/*
 *    ProcSent() -  Process a sentence
 */
ProcSent(pRec, RecLens, inSent, cpid)
capt_str *pRec; unsigned RecLens[]; char *inSent; int *cpid;
```

```
{
 char buf[5][30];
 char *p, *p1;
 char cpidc[20];
 char fld1[40], fld2[40], fld3[40];
 int i=0;
 int FldNo=-1;
 char RecBuff [400], WordBuff[30];
 int curr[] = {0,0,0,0,0,0,0,0};
 int hist[] = {0,0,0,0,0,0,0,0};

 memset(pRec, 0, sizeof(*pRec));
 for (i=0; i<=7; i++) curr[i] = 0;
 InitTab();
 strcpy(RecBuff, inSent);
 sscanf(inSent, "%s %s %s", fld1, fld2, fld3);
 *cpid = 0;
 if (fld2[0] == '>') {
   *cpid = atoi(fld1);
   strcpy (RecBuff, skipwhite(strrchr(inSent, '>') + 1));
 };
 if (*cpid == 0) {
   (int) (*cpid = GenRnd());
 };
 sprintf (cpidc, "%d", *cpid);
 PutOneFld(pRec, RecLens, 0, cpidc);
 p = skipwhite(strtok (RecBuff, " ,"));
 if (*p == ';') return;     /* it's a comment line */
 while (p && *(p)) {
   strcpy (WordBuff, p);
   FldNo++;
   /* ParseWord(WordBuff);  */
   ProcOneWord(pRec, RecLens, FldNo, WordBuff, hist, curr);
   p = skipwhite(strtok ('\0', " ,"));
 };
 printf(SNGCR);
};

/*
 *  ProcOneWord() - Process a particular word
 */
ProcOneWord (CaptTpl, RecLens, seq, w, hist, curr)
char *CaptTpl; unsigned RecLens[]; int seq; char *w;
int hist[]; int *curr;
{
  int RoleType, FldNo, nMeanings;
  int count[] = {0,0,0,0,0,0,0,0};
  nMeanings = ParseWord(w, seq, count, curr);
  RoleType = GuessRole(w, seq, nMeanings, count, hist, curr);
  switch (RoleType) {
    case AGENT:
       FldNo = 1;
       PutOneFld(CaptTpl, RecLens, FldNo, w);
       break;
    case ACTION:
       FldNo = 2;
       PutOneFld(CaptTpl, RecLens, FldNo, w);
       break;
    case PATIENT:
       FldNo = 3;
       PutOneFld(CaptTpl, RecLens, FldNo, w);
       break;
  };
};

/*
 *  GuessRole() - Guess the role for a word
 */
int GuessRole(p, seq, nMeanings, count, hist, curr)
char *p; int seq; int nMeanings; int count[]; int hist[]; int curr[];
{
  int POS;
  int nRoles;
  nRoles = (count[0]>0) + (count[1]>0) + (count[2]>0) + (count[3]>0)
         + (count[4]>0) + (count[5]>0) + (count[6]>0) + (count[7]>0);
  if ((nMeanings == 1) || (nRoles == 1)) {
  /*  POS = WhatPOS(p->feat) */
    if (count[VERB]) {
      if (TagTab[VERB][curr[VERB]-1] == seq) {
       RoleList[seq]=ACTION;
       strcpy(p, RootTab[VERB][curr[VERB]-1]) ;  /***/
       hist[ACTION]++;
       return ACTION;
      }
    }
    if (count[NOUN]) {
      if (TagTab[NOUN][curr[NOUN]-1] == seq) {
        if (hist[AGENT]) {
          RoleList[seq]=PATIENT;
          strcpy(p, RootTab[NOUN][curr[NOUN]-1]) ;  /***/
          hist[PATIENT]++;
          return PATIENT;
        };
        if (! hist[ACTION]) {
          RoleList[seq]=AGENT;
          strcpy(p, RootTab[NOUN][curr[NOUN]-1]) ;  /***/
          hist[AGENT]++;
          return AGENT;
        };
      };
    };
  };  /*==== end of nMeanings = 1, etc ====*/
```

```
  if (count[NOUN]) {
    if ((! hist[AGENT]) &&  (! hist[ACTION])) {
        RoleList[seq]=AGENT;
        strcpy(p, RootTab[NOUN][curr[NOUN]-1]) ;  /***/
        hist[AGENT]++;
        return AGENT;
    }
    /* (hist[AGENT] */
    if (hist[ACTION]) {
       RoleList[seq]=PATIENT;
       strcpy(p, RootTab[NOUN][curr[NOUN]-1]) ;  /***/
       hist[PATIENT]++;
       return PATIENT;
    };
  };
  if (count[VERB]) {
    if (hist[AGENT]) {
      RoleList[seq]=ACTION;
      strcpy(p, RootTab[NOUN][curr[NOUN]-1]) ;  /***/
      hist[ACTION]++;
      return ACTION;
    }
  };
  return UNKNOWN;
};

/*
 *  ParseWord() -  Try to parse one word
 */
int ParseWord(word, seq, count, curr)
char *word; int seq; int count[]; int curr[];
{
  int POS;
  RESULT *resp, *rp;
  int dummyint=0;
  int nMeanings=0;
  if (!isatty(fileno(stdin))) {
    puts(word);
  };
  resp = recognizer( word, &Lang, 0, 0, (FILE *)NULL);
  for ( rp = resp ; rp ; rp = rp->link ) {
    nMeanings++;
    if ((POS = WhatPOS(rp->feat)) == UNKNOWN){
      break;
    };
    count[POS]++;
    FillResTab(seq, word, rp, POS, curr);
  }
  if (resp != (RESULT *)NULL)
        free_result(resp);
  else
    printf("    form not recognized\n");
    printf("after if..else\n");
    return nMeanings;  /* POS: dummy, no info now  */
};

/*
 *  PutOneFld() -  Write one field into the CAPT database
 */
char *PutOneFld (aRec, str_len, fldno, content)
char *aRec; unsigned *str_len; int fldno; char *content;
{
  int i, offset=0;
  char *p;
  int FldLen;
  if (fldno < 0) {
    printf ("Invalid Field no: %i\n", fldno);
  };
  for (i=0; i<fldno; i++) {   /* not "from 0 to i-1" */
    offset = offset + *(str_len +i +1);
    /*equiv. to     offset = offset + str_len[i+1];  */
  }

#ifdef DEBUG
  printf ("offset=%i\n", offset);
#endif
   /* failed  p =  aRec + ((char *) offset) */ ;
   p =  (char *) aRec + offset ;

#ifdef DEBUG
  printf ("p=%p %u %x str_len=%i\n", p,p,p, str_len + fldno);
#endif

  FldLen = *(str_len + fldno + 1);
  memcpy (p, content, FldLen);
  return p;
};

/*
 *  GetOneFld() -  Read on field from the CAPT database
 */
char *GetOneFld (fld, aRec, str_len, fldno)
char *fld; char *aRec; unsigned *str_len; int fldno;
{
  int i, offset=0;
  char *p;
  int FldLen;
  char buff[200];
  if (fldno < 0) {
    printf ("Invalid Field no: %i\n", fldno);
  };
  for (i=0; i<fldno; i++) {   /* not "from 0 to i-1" */
    offset = offset + *(str_len +i +1);
```

```
    /*equiv. to     offset = offset + str_len[i+1];  */
  }
  p =  (char *) aRec + offset ;

#ifdef DEBUG
  printf ("p=%p %u %x capt_str_len=%i\n", p,p,p, capt_str_len[fldno]);
  printf ("p=%p %u %x capt_str_len=%i\n", p,p,p, capt_str_len[fldno]);
#endif

  FldLen = *(str_len + fldno + 1);
  memcpy (fld, p, FldLen);
  return fld;
};

/*
 *  CaptSearch() -  Try to search for captions
 */
CaptSearch(rel, pRecQ)
relation  *rel; capt_str  *pRecQ;
{
   int i;
   int arysize;
   int CapsSorted[4][25];
   if ( ! (pRecQ->agent[0]) &&
        ! (pRecQ->action[0]) &&
        ! (pRecQ->patient[0])) {
       return;
   };
   memset(CapsSorted, 0, sizeof(CapsSorted));
   memset(Matched, 0, sizeof(Matched));
   SearchForRole(Matched[AGENT], rel, 1, *pRecQ);
   SearchForRole(Matched[ACTION], rel, 2, *pRecQ);
   SearchForRole(Matched[PATIENT], rel, 3, *pRecQ);
   SortCaps(CapsSorted[1], CapsSorted[2], CapsSorted[3],
            Matched[AGENT], Matched[ACTION], Matched[PATIENT]);
   ScoreAllMatched(rel, *pRecQ, CapsSorted[1],
                   CapsSorted[2], CapsSorted[3]);
   ShowSimCap();
};

/*
 *  SimAllRoles() -  Calc the similarity of all roles
 */
float SimAllRoles(rel, RecQ, capidC)
relation  *rel; capt_str RecQ; int capidC;
{
  int weight[4];
  float sim[4];
  float sum = 0;
  int capidQ, att;
  float ResSim;
  char buff[200];
  capt_str aim; capt_str RecC;
  char roleQ[30], roleC[30];
  if (capidC == 0) {
    return(-1.0);
  };
  weight[AGENT]  =1;
  weight[ACTION] =1;
  weight[PATIENT]=1;
  memset (&aim, 0, sizeof(aim));
  memset (&RecC, 0, sizeof(RecC));
  for (att=1; att<=3; att++) {
    sprintf (aim.capid, "%d", capidC);
    if ((mb_sel (rel, 0, buff, EQUAL, &aim)) == MB_OKAY) {
         /*          ^^ which index: 0,1,2....  */
         /*   size(buff)  must be >>> rec len  */
         /* mb_sel (rel, 0, RecC, NEXT, &aim);  */
      GetOneFld (roleQ, &RecQ, capt_str_len, att);
      GetOneFld (roleC, buff, capt_str_len, att);
     /* strcpy (roleC, RecC.agent); */
      capidQ = atoi(RecQ.capid);
      if (roleQ[0] && roleC[0]) {
         sim[att] = SimAllFea(capidQ, capidC, roleQ, roleC, attrname[att]);
      } else {
         sim[att] = 0;
      };
      sum = sum + sim[att];
    };
  };
  printf ("sum = %f\n", sum);
  return (sum/(weight[1]+weight[2]+weight[3]));
};

/*
 *  ScoreAllMatched() -  Fine score of all matched captions
 */
ScoreAllMatched(rel, RecQ, CapAry1, CapAry2, CapAry3)
relation *rel; capt_str RecQ; int CapAry1[], CapAry2[], CapAry3[];
{
  int i;
  float SimVal;

  memset(SimCap, 0, sizeof(SimCap));
  memset(SimCapid, 0, sizeof(SimCapid));
  CalcAllCaps (rel, &RecQ, CapAry3);
  CalcAllCaps (rel, &RecQ, CapAry2);
  CalcAllCaps (rel, &RecQ, CapAry1);
};

/*
 *  CalcAllCaps() -  Fine Scoring of the selected captions
 */
```

```
CalcAllCaps(rel, pRec, CapsAry)
relation *rel; capt_str *pRec; int CapsAry[];
{
  int i;
  float SimVal;
  for (i=0; CapsAry[i]; i++) {
    SimVal = SimAllRoles(rel, *pRec, CapsAry[i]);
    SimCap[SimCapPtr] = SimVal;
    SimCapid[SimCapPtr]= CapsAry[i];
    SimCapPtr++;
  };
};

/*
 *  SortCaps() - Sort the candidate captions by their ids
 */
SortCaps(S1, S2, S3, L1, L2, L3)
int *S1, *S2, *S3, *L1, *L2, *L3;
{
  int R[100];
  int nR=0, n1=0, n2=0, n3=0;
  for (n1=0; L1[n1]; n1++) {
    R[nR++]=L1[n1];
  };
  for (n2=0; L2[n2]; n2++) {
    R[nR++]=L2[n2];
  };
  for (n3=0; L3[n3]; n3++) {
    R[nR++]=L3[n3];
  };
  bubble(R, nR);
  extrCaps(S1, S2, S3, R, nR);
}


/*
 *  SearchForRole() -  Try to match a role in query
 */
SearchForRole(outlist, rel, roleSeq, rec)
int *outlist; relation *rel; int roleSeq; capt_str rec;
{
#define SUFFICIENT 1
  int fldno;
  int LstIdx = 0;
  char TargetRole[20];
  int nMatchedExact = 0;
  int nMatchedImChildn = 0;
  int nMatchedSibls = 0;
  int nSibls = 0;
  int nTotalMatched = 0;
  char ChildnBuff[MAXCHILDN][20];
  char SiblsBuff[MAXSIBLS][20];
  int i;
  fldno = roleSeq;
  GetOneFld (TargetRole, &rec, capt_str_len, fldno);
  nMatchedExact = SearchExactRole(outlist, rel,
                  roleSeq, &LstIdx, TargetRole, &rec);
  if (nMatchedExact >= SUFFICIENT) {
    return nMatchedExact;
  };
  nTotalMatched = nMatchedExact;
  memset(SiblsBuff, 0, sizeof(SiblsBuff));
  nSibls = GetAllSibls(HFILE, MAXSIBLS, SiblsBuff,
              sizeof(SiblsBuff[0]), TargetRole);
  for(i=0; (i<MAXSIBLS) && (SiblsBuff[i][0]); i++) {
    nMatchedSibls += SearchExactRole(outlist, rel,
                roleSeq, &LstIdx, SiblsBuff[i], &rec);
  };
       /**** ^^^ check !! FAILED in find "sibls of Targ... "  **/
  nTotalMatched += nMatchedSibls;
  return nTotalMatched;
};

/*
 *  SearchExactRole() -  Process to exact matching a role
 *                       in query
 */
SearchExactRole(outlist, rel, roleSeq, pIdx, aimRole, pRec)
int *outlist; relation *rel; int roleSeq; int *pIdx;
char *aimRole; capt_str *pRec;
{
  char buff_rec[200]; /* more than sufficient */
  char fld[20];
  char ccpid[20];
  capt_str KeyRec;
  if (! aimRole[0]) {
    return (0);
  }
  PutOneFld(&KeyRec, capt_str_len, roleSeq, aimRole);
                  /**  ^^^^^^ ********/
   /* Be Care: roleSeq not necessary = idx */
  /** if ((mb_sel (rel, roleSeq, buff_rec, EQUAL, pRec)) == MB_OKAY) { **/
  if ((mb_sel (rel, roleSeq, buff_rec, EQUAL, &KeyRec)) == MB_OKAY) {
    /*               ^^ which index: 0,1,2....  */
    StoCapid(outlist, pIdx, buff_rec, capt_str_len);
  } else {
  };
  while ((mb_sel (rel, roleSeq, buff_rec, NEXT, pRec)) == MB_OKAY) {
    PrintRec(pRec);
    GetOneFld(fld, buff_rec, capt_str_len, roleSeq);
    /*   printf ("*******fld=[%s] aimRole=[%s]\n", fld, aimRole);
         PrintRec(pRec);
    */
```

```
    if (strcmp(fld, aimRole) != 0) {
      break;
    };
    printf (" ### found ## ");
    StoCapid(outlist, pIdx, buff_rec, capt_str_len);
  };
  return *pIdx;
};

/*
 *  StoCapid() - Store a capid in result list
 */
StoCapid(list, idx, recp, str_len)
int *list; int *idx; capt_str *recp; unsigned *str_len;
{
  char ccpid[20];
  GetOneFld(ccpid, recp, str_len, 0);
     /*    atoi(ccpid);  */
  list[(*idx)++] = atoi(ccpid);
      /*   printf(" *(outlist[LstIdx]) =%d\n",
           *(outlist[*idx]));
            PrintRec(buff_rec);
      */
};

/*
 *  PrintRec() -  Print a CAPT record (for debugging)
 */
PrintRec(pRec)
capt_str  *pRec;
{
  printf ("CapId=[%s]  Agent[%s]  Action[%s]  Patient[%s]\n",
          pRec->capid, pRec->agent,
          pRec->action, pRec->patient);

};

/*
 *  AddOrUpdate() -  Check whether updating an old record
 *                   or adding a new record to the CAPT database
 */
AddOrUpdate(rel, rec_tp)
relation *rel; char *rec_tp;
{
   char buff_rec[200];  /* more than sufficient */
   char sh[100], oldcid[20];
/* #ifdef DEBUG
   printf ("capt-rec->capid=[%s]\n", ((capt_str *)rec_tp)->capid);
 #endif */
   if ( !(((capt_str *) rec_tp)->agent[0]) &&
        !(((capt_str *) rec_tp)->action[0]) &&
        !(((capt_str *) rec_tp)->patient[0])) {
       return;
   };
   if ((((capt_str *)rec_tp)->capid != 0) &&
        ((mb_sel (rel, 0, buff_rec, EQUAL, rec_tp)) == MB_OKAY)) {
        /*             ^^ which index: 0,1,2....  */
         sprintf (oldcid, "%s", ((capt_str*) &buff_rec)->capid);
         sprintf (sh, "rm -r %s%s", CAPTPATH, oldcid);
         system(sh);
  /*************** don't use rm -r ***********/
         printf ("Update now\n");
         if ((mb_upd (rel, rec_tp)) != MB_OKAY) {
            printf ("UPD FAILED: %s%s", mb_error, SNGCR);
            mb_exit(3);
         };
         return;
   };
   if ((mb_add (rel, rec_tp)) != MB_OKAY) {
         printf ("ADD FAILED: %s%s\n", mb_error, SNGCR);
         mb_exit(3);
         printf ("Rec added.\n");
   };
 };

 /*
  *  DispRelLen() -  Display the record length of
  *                  a caption record
  */
 DispRelLen()
 {
   int i=0, curr=0;
   int NoFlds;
   NoFlds = (sizeof (capt_str_len)) / sizeof (capt_str_len[0]);
   printf ("No of fields = %i\n", NoFlds);
   for (i=0; i<NoFlds; i++) {
     printf ("%i   ", capt_str_len[i]);
   }
 }

  /*
   *  InitTab() -  Initialize the global tables
   */
  int InitTab()
  {
    memset(ResTab, 0, sizeof(ResTab));
    memset(OrigTab, 0, sizeof(OrigTab));
    memset(RootTab, 0, sizeof(RootTab));
  };

  /*
   *  FillResTab() -  Fill the result of parsed word
   *                  into the result table
```

```
 */
int FillResTab(tag, orig, rp1, POS, curr)
int tag;
char *orig;
RESULT *rp1;
int POS;
int curr[];
{
  char buff[40];
  if (POS != UNKNOWN) {
    strcpy (OrigTab[POS][curr[POS]], orig);
    strcpy (ResTab[POS][curr[POS]], rp1->str);
    strcpy (RootTab[POS][curr[POS]], GetRoot(rp1->str));
    TagTab[POS][curr[POS]] =  tag;
    curr[POS]++;
  };
};

/*
 *   WhatPOS() -  Check the Part-of-speech of a word
 */
int WhatPOS (feat)
char *feat;
{
  char buff[40];
  strcpy (buff, feat);
    /*****  The sequence is critical ***/
  if (strstr (buff,"AV") != NULL) {    /* found AV */
    return ADV;
  }
  if (strstr (buff,"AJ") != NULL) {    /* found AJ */
    return ADJ;
  }
  if (strstr (buff,"NR") != NULL) {    /* found NR */
    return NOUN;
  }
  if (strstr (buff,"VR") != NULL) {    /* found VR */
    return VERB;
  }
  if ('N' == buff[0])  {             /* N at the beginning */
    return NOUN;
  }
  if ('V' == buff[0]) {             /* V at the beginning */
    return VERB;
  };
  return UNKNOWN;
};

/*
 *   AskForRole () -  Check the role of a word
 */
int AskForRole (word, hist, listPOS)
char *word; int hist[]; int listPOS[];
{
  char desc[8][20], ansbuff[20];
  int choice =-1;
  int i = 0 ;
  strcpy (desc[ACTION], "action");
  strcpy (desc[AGENT], "agent");
  strcpy (desc[PATIENT], "patient");
  printf ("Possible parts of speech %s\n", word);
  for (i=0; (listPOS[i] != EOL); i++) {
    printf (" %i. %s\n", i, desc[i]);
  };
  while ((choice<0) || (choice>=i)) {
    printf ("Please choose the closest one => ");
    gets (ansbuff);
    choice = atoi(ansbuff);   /* problem:  return 0 when unsolved !!*/
  };
  printf ("Your choice is (%i) %s\n", choice, desc[choice]);
  return choice;
};

/*
 *   ProcAttrib() -  Process all attributes
 */
ProcAttrib(captid)
int captid;
{
  int i;
  expfea_str  EfTpl;
  char sh[100];
  char *p;
  int curr[]={0,0,0,0};
  memset (&EfTpl, 0, sizeof(EfTpl));
  memset (attrib, 0, sizeof(attrib));
  for (i=0; (i<=7) && OrigTab[ADJ][i][0]; i++) {
    EachAttrib(ADJ, i, attrib, curr);
  };
  for (i=0; (i<=7) && OrigTab[ADV][i][0]; i++) {
    EachAttrib(ADV, i, attrib, curr);
  };
  if (captid == QID) {
    sprintf (sh, "rm -r %s%d", CAPTPATH, QID);
    system(sh);
  };
  memset(curr, 0, sizeof (curr));
  p = attrib[AGENT][curr[AGENT]];
  if (p) {
    WriteEfea(captid, p, ATTR_AG);
  };
  p = attrib[ACTION][curr[ACTION]];
  if (p) {
```

```
    WriteEfea(captid, p, ATTR_AC);
   };
   p = attrib[PATIENT][curr[PATIENT]];
   if (p) {
     WriteEfea(captid, p, ATTR_PA);
   };
   /*....... action, pat.... */
};

/*
 *    WriteEfea()  -  confirm the expfea in a caption
 */
void WriteEfea(captid, attr, attrtype)
  int captid; char *attr; char *attrtype;
  {
    char EfSlot[30], EfVal[30];
    char expfea[80];
    char outfile [80], sh[120];
    Attr2Efea("*", attr, EfSlot, EfVal);
    sprintf(sh, "mkdir %s%d", CAPTPATH, captid);
    system (sh);
    sprintf(expfea, "%s\t%s", EfSlot, EfVal);
    sprintf(outfile,"%s%d/%s", CAPTPATH, captid, attrtype);
    sprintf (sh, "echo \"%s\" >  %s", expfea, outfile);
    system (sh);

  };

/*
 *    GetRoot()  -  Extract the root of a word from
 *                  the parsed result string
 */
char *GetRoot(ResStr)
char *ResStr;
{
   char *p;
   if (p=strchr(ResStr,'+')) {
     memset(p,'\0',1);
   };
   if (p=strchr(ResStr,''')) {
     return p+1;
   };
   return ResStr;
};

/*
 *    EachAttrib()  -  Process each attribute
 */
void EachAttrib(POS, i, attrib, curr)
int POS; int i; char *attrib[4][10]; int curr[];
{
  char temp;
  int seq;
  /*  if (ResTab[ADJ][i]) {  */
  if (POS==ADJ) {
    seq = TagTab[ADJ][i];
    switch (RoleList[seq+1]) {
      case AGENT :  attrib[AGENT][curr[AGENT]++] = RootTab[ADJ][i];
                    RoleList[seq]=EFEA_AGENT;
                    return;
      case PATIENT: attrib[PATIENT][curr[PATIENT]++] = RootTab[ADJ][i];
                    RoleList[seq]=EFEA_PATIENT;
                    return;
    };
  };
  if (POS==ADV) {
    seq = TagTab[ADV][i];
    switch (RoleList[seq-1]) {
      case ACTION:  attrib[ACTION][curr[ACTION]++] = (RootTab[ADV][i]);
                    RoleList[seq]=EFEA_ACTION;
                    return;
    };
  };
 };

 /*
  *    GenRnd() -  Generate an integer
  */
 GenRnd()
 {
   int i, r;
   FILE *fp;
   char fn[60];
   srand(time(NULL));
   r = OneRndInt();
   sprintf(fn, "%d", r);
   /* test of existence) */
   for (;FileExist(CAPTPATH, fn);) {
     /*==  fn existing  ==*/
     r = ((unsigned) (rand()/100000)) + 100;
     sprintf(fn, "%d", r);
   };
  /*  fclose(fp);
   unlink(fn);  */
   return r;
 };

  /*
   *    FileExist() - Check the existence of a file
   */
  FileExist(dname, fname)
  char *dname, *fname;
  {
```

```
  char sh[200], buf[BSIZE];
  FILE *pfp;
  sprintf (sh,
           "(cd %s; ls) | grep %s | cut -f1 -d / | fgrep -x %s",
           dname, fname, fname);
  if ((pfp = popen(sh,"r")) != NULL) {
    if (fgets (buf, BSIZE, pfp) != NULL) {
        printf("@");
        pclose(pfp);
        return 1;
    };
  };
  pclose(pfp);
  return 0;
};

/*
 *    Attr2Efea() - Convert attributes into expfeas
 */
Attr2Efea(host, attr, efSlot, efVal)
/* Attr2Efea(attr, efSlot, efVal) */
char *attr, *efSlot, *efVal;
{
  char sh[BSIZE], buf[BSIZE];
  char *p;
  FILE *fp;
/*
  efSlot[0] = '\0';
  efVal[0] = '\0';
*/
  strcpy(efSlot, "attr");
  strcpy(efVal, attr);
  sprintf(sh, "grep %s %s | cut -f1", attr, FEA_RUL);
  system(sh);
  if ((fp = popen(sh,"r")) != NULL) {
    while (fgets (buf, BSIZE, fp) != NULL) {
      StripNL(buf);
      p = (char *) strchr(buf, (int) ':');
      strcpy (efVal, p+1);
      *p = (char) NULL;
      strcpy (efSlot, buf);
    };
    pclose(fp);
  }
}

/*
 *    extrCaps() - Extract captions from candidates
 */
void extrCaps(outary1, outary2, outary3, intary, size)
int *outary1, *outary2, *outary3; int *intary; int size;
{
  int i,j;
  int q[] = {0,0,0,0};  /* useful 1..3 */
  int count=1;
  int last = -1;
  int curr;
  for (i=0; i<size; i++) {
    curr = intary[i];
    if (curr==last) {
      count++;
    };
    if ((curr != last) || (i==size-1)) {
      if (last>0) {
        switch (count) {
          case (1) : outary1[q[count]] = last;
                     break;
          case (2) : outary2[q[count]] = last;
                     break;
          case (3) : outary3[q[count]] = last;
                     break;
        };
          /*      outary[count][q[count]] = intary[i];   */
        (q[count])++;
        count=1;
      };
      if ((i == size-1) && (last!=curr)) {   /* flush the last one */
        outary1[q[count]] = curr;
      };
      last = curr;
    };
  };
  printf("Matched 1 roles -> ");
  for (j=0; (j<size) && outary1[j]; j++) printf ("[%d] ", outary1[j]);
  printf("\n");
  printf("Matched 2 roles-> ");
  for (j=0; (j<size) && outary2[j]; j++) printf ("[%d] ", outary2[j]);
  printf("\n");
  printf("Matched 3 roles-> ");
  for (j=0; (j<size) && outary3[j]; j++) printf ("[%d] ", outary3[j]);
  printf("\n");
};

/*
 *    bubble() - sort the integer array with bubble sort
 */
void bubble(intary, size)
int *intary; int size;
{
  int i, j;
  for (i=size-1; i>=1; i--) {
    for (j=1; j<=i; j++) {
      if (intary[j] < intary[j-1]) {
```

```
      swapint(&(intary[j]), &(intary[j-1]));
     };
   };
  };
};

/*
 *    swapint() - Swap two integers
 */
void swapint(highint, lowint)
int *highint, *lowint;
{
   int i;
   i = *lowint;
   *lowint = *highint;
   *highint = i;
};

/*
 *    OneRndInt() - Get a randomized random integer
 */
int OneRndInt()
{
    long k1, k2, m;
    char buff[20];
    k1 =  rand();
    k2 =  rand();
    sprintf (buff, "%ld%ld", k1, k2);
    memset(buff+6, 0, 1);
    memset(buff, '0', 1);
    m = (int) (atoi(buff)/4) + 100;
    /*   printf ("[%d %s %d]  ", k1, buff, m);  */
    return m;
};

/*
 *    skipwhite() - skip the leading space in a string
 */
char *skipwhite(cp)
char *cp;
{
  register char *p;

  if ((cp == (char *)NULL) || (*cp == NUL)) {
    return(cp);
  };
  for ( p = cp ; isspace(*p) ; ++p ) ;
  return(p);
}
```

```
/****************************************************************
 *      capop.c - Caption operations in ARMON
 ****************************************************************
 *
 *      Written by   : kpwong
 *
 *      Last Updated : 23 May 1995
 *
 ****************************************************************
 */

#define  ATTRNAME
#include "capfunc.h"
#include "hierfunc.h"

LANGUAGE Lang;
long   elaparr[79];
char hfile[200];
char _buf_[127];

extern char *skipwhite();
extern RESULT *recognizer();

void  main ();

/*
 *   external modules found in *.c
 */
extern int GenRnd();
extern char *mystrtok(), *GetOneFld(), *PutOneFld(), *GetRoot();
extern int DispRelLen();
extern int ProcSent();
extern int ProcAttrib();
extern int ShowOrigTab();
extern int ShowResTab();
extern int ShowTagTab();
extern int ShowRoleList();
extern void ShowAttrib();
extern int ShowMatchedCaps();

/*
```

```
 *   global tables defined in *.h
 */
extern char RootTab[];
extern char ResTab[];
extern char OrigTab[];
extern int RoleList[];
extern int TagTab[];
extern char* attrib[];

/*
 *  main program starts
 */

void main  (argc, argv)
int    argc;
char **argv;
{
#define DO_ENTER  0
#define DO_SEARCH 1
#define SEARCH_BINNAME "search"

    int      cpid;
    int      curr_op=0;
    char     buff[400];

    char     inCapt[20];
    char     CaptRelRawFn[] = "capt.rel";
    char     CaptRelFn[120];
    FILE     *inCaptFp;
    relation  *CaptRel;
    capt_str  capRec;
    /* First, parse the command line, and set {num} */
    strcpy(hfile, HFILE);  /*** init ***/
    if (argc == 2) {
       strcpy(inCapt, argv[1]);
       printf ("inCapt=%s\n", inCapt);
    } else {
       fprintf (stderr, "Syntax: %s textfile\n", argv[0]);
       mb_exit (1);
    };
    printf ("cmd=%s\n", argv[0]);
    sprintf (CaptRelFn, "%s/%s", CAPTPATH, CaptRelRawFn);
    /*
    *  It's a searh operation
    */
    if (strstr(argv[0], SEARCH_BINNAME) != NULL) {
      curr_op = DO_SEARCH;
    };
    printf ("&\n");
    if ((inCaptFp = fopen(inCapt, "r")) == NULL) {
       fprintf (stderr, "File %s open error, please check!\n", inCapt);
       mb_exit (1);
    }
    printf ("&& CaptRelFn=%s\n", CaptRelFn);
    if ((CaptRel = mb_inc (CaptRelFn, 0)) == RNULL){
       fprintf (stderr, "%s.%s", mb_error, SNGCR);
       mb_exit (2);
    }
    printf ("&&&\n");

    /*
     *  load the rules file
     */
    if (load_rules(RUL_FILE, &Lang, ';') < 0)
      exit(1);
    /*
     *  load the lexicon file
     */
    if (load_lexicons(LEX_FILE, &Lang, ';') < 0)
      exit(1);
    DispRelLen();
    while (!feof(inCaptFp)){
      if (!fgets(buff,199,inCaptFp)) {
        break;
      };
      StripNL(buff);
  /*    ProcSent(&pRec, capt_str_len, buff, &cpid); */
      ProcSent(&capRec, capt_str_len, buff, &cpid);
      if (curr_op != DO_SEARCH) {
        /*
         * Add new caption
         */
        AddOrUpdate(CaptRel, &capRec);
        ProcAttrib(cpid);
        /*  These functions are useful for debugging
        ShowOrigTab();
        ShowResTab();
        ShowTagTab();
        ShowRoleList();
        ShowAttrib();
        */
      } else {
        /*
         * Search for existing captions
         */
        printf ("will preform searching .....\n");
        ProcAttrib(QID);
        sprintf(capRec.capid, "%d", QID);
        CaptSearch(CaptRel, &capRec);
        /*  These functions are useful for debugging
        ShowMatchedCaps();
        ShowOrigTab();
        ShowResTab();
```

```
      ShowTagTab();
      ShowRoleList();
      ShowAttrib();
      */
    };
  };
  mb_exit(1);
};
```

```
/*
 * METALBASE 5.0
 *
 * Released October 1st, 1992 by Huan-Ti [ richid@owlnet.rice.edu ]
 *                                       [ t-richj@microsoft.com ]
 *
 * Special thanks go to Mike Cuddy (mcuddy@fensende.rational.com) for his
 * suggestions and code.
 *
 *-----------------------------------------------------------------
 *
 * Modified by   : KPWONG
 * Purpose       : To be used with ARMON
 * Path          : $PRJHOME/src/build.c
 * Last Updated  : 9 Jan 1995
 *
 */
 */

#define BLAST_C  /* I know, I know... */
#include "mbase.h"
#include "internal.h"

#define cr(x) ((x) == 0) ? DUBCR : SNGCR

#ifdef MSDOS
#define DESCLINE "/*\r\n * This file was created by MetalBase version 5.0 to reflect the structure\r\n * of
the relation \"%s\".\r\n *\r\n * MetalBase 5.0 released October 1st, 1992 by richid@owlnet.rice.edu\r\n *\r\n
*/\r\n\r\ntypedef struct\r\n { "
#else
#define DESCLINE "/*\n * This file was created by MetalBase version 5.0 to reflect the structure\n * of the
relation \"%s\".\n *\n * MetalBase 5.0 released October 1st, 1992 by virtual!richid@owlnet.rice.edu\n *\n
*/\n\ntypedef struct\n { "
#endif

#define lineF "Fields____________________________________________________%s"
#define lineI "\nIndices__________________________________________________%s"

#define RBC '}'

#ifdef LONGARGS
   void   strlwrcpy (char *, char *);
   void   struprcpy (char *, char *);
   void   strmax    (char *, int);
   char  *repeat    (char,   int);
   void   main      (int,    char **);
   void   endoffile (int,    int);
   int    get_names (int,    char **);
   void   write_it  (int,    int);
   int    contains_serial (char *);
#else
   void   strlwrcpy();
   void   struprcpy();
   void   strmax();
   char  *repeat();
   void   main();
   void   endoffile();
   int    get_names();
   void   write_it();
   int    contains_serial();
#endif

#define  Printf   if (!quiet)  printf
#define fPrintf   if (!quiet) fprintf

#define qt(x) (quiet ? "" : x)

#define usage() \
   fprintf (stderr, "build: format: build [-q] [-h] schema.s%s", SNGCR);

#define fatal() { \
                fflush(stdout); \
                fprintf(stderr,"Cannot build relation--%s.%s",mb_error,SNGCR); \
                break; \
                }

#define comment() skip(fh,";"); while (skip (fh, "#"))  goeol(fh,NULL);

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

static char *types[] =
```

```
{ "char *", "short", "ushort", "long", "ulong",  "float",
  "double", "money", "time",   "date", "serial", "phone"  };

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

relation *data;

char    strname[40] = "";     /* Structure name (set by "typedef") */
char    rel[128],  hdr[128];  /* Filenames for relation and header */
char    names[20], nameb[40]; /* Name, and upper-case name         */
int     column=1;             /* Column we're displaying data in   */
int     header=0,  quiet=0;   /* Set by -q and -h on command-line  */
int     num_f=0,   num_i=0;   /* Start with 0 fields and 0 indices */
int     hasser=0;             /* 1 if we encounter a serial field  */


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

void
main  (argc, argv)
int    argc;
char **argv;
{
   int        stage;      /* Processing stage; 1==fields, 2==indices, 3==done */
   int        fh;         /* File handle for schema                           */
   char       name[20];   /* Field/Index name                                 */
   ftype      typ;        /* Field type (or, for indices, 0==nodups, 1==dups) */
   int        siz;        /* Field size (for character arrays only)           */
   int        isExt;      /* TRUE if it's an external type, FALSE if not      */
   char       desc[128];  /* Character array of field numbers, for indices    */

   long       nexts = 0L;
   char       temp[128];
   char       t2[128];
   int        i;


   fh = get_names (argc, argv);  /* fh = file handle of relation */

   if ((data = mb_new()) == RNULL)
      {
      fprintf (stderr, "Cannot build relation--%s.%s", mb_error, SNGCR);
      exit(1);
      }

   for (stage = 1; stage != 3; )
      {
      strlwrcpy (temp, getword (fh));    /* temp = keyword */

      if (! strcmp (temp, "field"))
         {
         if (stage == 2)  /* Done with fields? */
            {
            fflush (stdout);
            fprintf (stderr, "%s%sField %s declared after indices.%s",
                     qt(SNGCR), qt(cr(column)), getword(fh), SNGCR);
            break;
            }

         strlwrcpy (temp, getword (fh));  /* New field?  Obtain, in lower,  */
         strmax (temp, 20);               /* its name.  Put it in 'temp' first */
         strcpy (name, temp);             /* in case it's really long.      */

         if (mb_getname (data, name, 0) != -1)
            {
            fflush  (stdout);
            fprintf (stderr, "%sField %s declared twice.%s", qt(cr(column)),
                             name, SNGCR);
            break;
            }

         (void)skip (fh, "type");         /* Got its name, and it's new.  So */
         strlwrcpy (temp, getword (fh));  /* get its field type...           */

         isExt = 0;
         if (! strcmp (temp, "extern") || ! strcmp (temp, "external"))
            {
            isExt = 1;
            strlwrcpy (temp, getword (fh));  /* External?  Get the next word. */
            }

         typ = (ftype)-1;
         if (! strcmp (temp, "char") || ! strcmp (temp, "character") ||
             ! strcmp (temp, "string"))
            {
            typ = T_CHAR;
            }
         if (! strcmp (temp, "short"))    typ = T_SHORT;
         if (! strcmp (temp, "ushort"))   typ = T_USHORT;
         if (! strcmp (temp, "long"))     typ = T_LONG;
         if (! strcmp (temp, "ulong"))    typ = T_ULONG;
         if (! strcmp (temp, "float"))    typ = T_FLOAT;
         if (! strcmp (temp, "double"))   typ = T_DOUBLE;
         if (! strcmp (temp, "money"))    typ = T_MONEY;
         if (! strcmp (temp, "time"))     typ = T_TIME;
         if (! strcmp (temp, "date"))     typ = T_DATE;
         if (! strcmp (temp, "serial"))   typ = T_SERIAL;
```

```
if (! strcmp (temp, "phone"))    typ = T_PHONE;

if (typ == (ftype)-1)
   {
   fflush  (stdout);
   fprintf (stderr, "%sType %s (field %s) undefined.%s",
                    qt(cr(column)), temp, name, SNGCR);
   break;
   }

if (isExt)
   {
   sprintf (temp, "ix_%s", name);
   sprintf (desc, "%d",    num_i);

   if (mb_addindex (data, temp, 1, desc) != MB_OKAY)
      fatal();

   if (typ == T_SERIAL)
      typ = T_LONG;
   }

if (typ == T_SERIAL)
   {
   if (hasser)
      {
      fflush  (stdout);
      fprintf (stderr, "%sMore than one serial field specified.%s",
                  qt(cr (column)), SNGCR);
      break;
      }
   hasser = 1;

   if (skip (fh, "start"))
      nexts = atol (getword (fh));
   }

switch (typ)
   {
   case T_CHAR:
      (void)skip (fh, "length");
      (void)skip (fh, "*");
      siz = atoi (getword(fh));
      sprintf (temp, "%s [%s%d]", name, types[(int)typ], siz);
      mb_addfield (data, name, T_CHAR, siz);
      break;

   case T_SERIAL:
      sprintf (temp, "%s [%s @%ld]", name, types[(int)typ], nexts);
      mb_addfield (data, name, T_SERIAL, nexts);
      break;

   default:
      sprintf (temp, "%s [%s]", name, types[(int)typ]);
      mb_addfield (data, name, typ, 0);
      break;
   }

if (mb_errno)
   fatal();

if ((column = 1-column) == 0)
   { Printf ("%s%-30.30s%s", SUBD, temp, NORM); }
else
   { Printf ("%s%s%s%s",     SUBD, temp, NORM, SNGCR); }

num_f ++;

comment();

continue;
}

if (strcmp (temp, "index") == 0)
   {
   if (stage == 1)
      {
      if (column == 0)
         Printf (SNGCR);

      if (num_f == 0)
         {
         fflush  (stdout);
         fprintf (stderr, "%sNo fields declared before indices.%s",
                          qt(SNGCR), SNGCR);
         break;
         }

      Printf (lineI, SNGCR);

      stage  = 2;
      column = 1;
      }

   strlwrcpy (temp, getword (fh));  /* New index?  Get the name (in   */
   strmax (temp, 20);               /* temp first in case it's long) and */
   strcpy (name, temp);             /* make sure it's unique.           */

   if (mb_getname (data, name, 1) != -1)
      {
      fflush  (stdout);
      fprintf (stderr, "%sField %s declared twice.%s", qt(cr(column)),
                       name, SNGCR);
```

```
            break;
            }

         (void)skip (fh, "on");

         for (temp[0] = desc[0] = 0; ; )
            {
            strlwrcpy (t2, getword (fh));

            if ((i = mb_getname (data, t2, 0)) == -1)
               {
               fflush  (stdout);
               fprintf (stderr, "%sIndex placed on undeclared field %s.%s",
                                qt(cr(column)), t2, SNGCR);
               exit (1);
               }

            strcat  (temp, t2);
            sprintf (t2, "%d", i);
            strcat  (desc, t2);

            if (! skip (fh, ","))
               break;

            strcat (temp, ",");
            strcat (desc, ",");
            }

         Printf ("%s%s", name, repeat ('.', 15-strlen (name)));
         Printf ("%s%s", temp, repeat ('.', 22-strlen (temp)));

         typ = (ftype)0;

         if (skip (fh, "without"))
            {
            if (skip (fh, "duplicates") || skip (fh, "dups"))
               typ = (ftype)0;
            else
               typ = (ftype)2;
            }
         else if (skip (fh, "with"))
            {
            if (skip (fh, "duplicates") || skip (fh, "dups"))
               typ = (ftype)1;
            else
               typ = (ftype)2;
            }
         if (typ == (ftype)2)
            {
            fflush  (stdout);
            fprintf (stderr, "?%sIncorrect syntax%s", qt(DUBCR), SNGCR);
            exit    (1);
            }

         if ((int)typ)  { Printf ("Duplicates allowed%s", SNGCR);     }
         else           { Printf ("Duplicates not allowed%s", SNGCR); }

         if (contains_serial (desc))
            typ = (ftype)1;

         if (mb_addindex (data, name, (int)typ, desc) != MB_OKAY)
            {
            fatal();
            }

         num_i ++;

         comment();

         continue;
         }

      if (strcmp (temp, "end") == 0 || temp[0] == 0)
         {
         Printf ("%s", cr (column));
         endoffile (num_f, num_i);
         stage = 3;

         continue;
         }

      if (! strcmp (temp, "typedef"))
         {
         strlwrcpy (strname, getword (fh));
         continue;
         }

      fflush  (stdout);
      fprintf (stderr, "%sIdentifier %s%s%s not recognized.%s",
                       qt(cr(column)), BOLD, temp, NORM, SNGCR);
      exit    (1);
      }

   if (stage != 3)
      {
      exit (1);
      }

   write_it (num_i, num_f);

   Printf ("Relation created -- zero entries.%s", SNGCR);

   exit (0);
```

```
}

void
write_it (num_i, num_f)
int       num_i, num_f;
{
   char  temp[512], temp2[200], buf1[64], buf2[64], buf3[64];
   char  tempkp[32];     /*wkp*/
   int   R, H;
   int   i, j;
   int   k;      /* wkp 25Aug94 */

   if ((R = openx (rel, OPENMODE)) != -1)
      {
      if (read (R, temp, 1) != -1)
         {
         if (temp[0] != 50 && temp[0] != 42)  /* Check for 4.1a or 5.0 sig */
            {
            fprintf (stderr, "%s%s%s%32.32s%-28.28s%s%s", SNGCR, SUBD, INVR,
                              "*** ERR", "OR ***", NORM, SNGCR);
            fprintf (stderr,
                     "%s   This relation is not in MetalBase 5.0 format.%s",
                     qt(SNGCR), DUBCR);
            close (R);
            exit (1);
            }
         }

      Printf ("%s%s%32.32s%-28.28s%s%s", SUBD, INVR, "** WARN", "ING **", NORM,
                                         SNGCR);
      Printf ("%s   The file about to be created already exists under the%s",
              SNGCR, SNGCR);
      Printf ("         target directory!  This data will be lost!%s", DUBCR);

      close (R);
      }

 /*
  * That was ugly.  Now make sure they wanna continue first...
  *
  */

   if (! quiet)
      {
      Printf ("Continue with the creation of the relation [Y/n] ? ");
      gets(temp);  i = (int)temp[0];
      if (i == 'n' || i == 'N' || i == 'q' || i == 'Q')  exit (0);
      }

   if (header || quiet)
      {
      i = (header ? 'y' : 'n');
      }
   else
      {
      Printf ("Create header file for this relation       [y/N] ? ");
      fflush(stdin);  gets(temp);  i = (int)temp[0];
      }
   Printf (SNGCR);

 /*
  * That was uglier.  At any rate, we now have permission to create the thing:
  *
  */

   if (mb_create (data, rel, 0) != MB_OKAY)
      {
      fflush(stdout);
      fprintf (stderr, "Cannot build relation--%s%s.", mb_error, SNGCR);
      return;
      }

 /*
  * Now if they want the header created, we've gotta do all kindsa special shit:
  *
  */

   if (i != 'y' && i != 'Y')
      {
      return;
      }

   if ((H = openx (hdr, O_RDWR)) != -1)
      {
      close  (H);
      unlink (hdr);
      }
   if ((H = creatx (hdr)) == -1)
      {
      fprintf (stderr, "%sSorry--cannot create header file%s", qt(DUBCR),
                       SNGCR);
      return;
      }
   modex (hdr, 0666);    /* Make the file   -rw-rw-rw-  */

   sprintf (temp, "#ifndef %s_H%s", nameb, SNGCR);
   writx   (H, temp, strlen(temp));
   sprintf (temp, "#define %s_H%s", nameb, DUBCR);
   writx   (H, temp, strlen(temp));
   sprintf (temp, DESCLINE, names);
   writx   (H, temp, strlen(temp));
```

```
for (j = 0; j < data->num_f; j++)
  {
  switch (data->type[j])
    {
    case T_CHAR:   sprintf (temp, "char     %s[%d];",
                            data->name[j], data->siz[j]);   break;
    case T_SHORT:  sprintf (temp, "short    %s;", data->name[j]);
                   data->siz[j] = sizeof (short);                /*WKP */
                   break;
    case T_USHORT: sprintf (temp, "ushort   %s;", data->name[j]);
                   data->siz[j] = sizeof (ushort);               /* WKP */
                   break;
    case T_LONG:   sprintf (temp, "long     %s;", data->name[j]);
                   data->siz[j] = sizeof (long);                 /* WKP */
                   break;
    case T_ULONG:  sprintf (temp, "ulong    %s;", data->name[j]);
                   data->siz[j] = sizeof (ulong);                /* WKP */
                   break;
    case T_FLOAT:  sprintf (temp, "float    %s;", data->name[j]);
                   data->siz[j] = sizeof (float);                /* WKP */
                   break;
    case T_DOUBLE: sprintf (temp, "double   %s;", data->name[j]);
                   data->siz[j] = sizeof (double);               /* WKP */
                   break;
    case T_MONEY:  sprintf (temp, "double   %s;", data->name[j]);
                   data->siz[j] = sizeof (double);
                   break;
    case T_TIME:   sprintf (temp, "mb_time  %s;", data->name[j]);
                   data->siz[j] = sizeof (mb_time);
                   break;
    case T_DATE:   sprintf (temp, "mb_date  %s;", data->name[j]);
                   data->siz[j] = sizeof (mb_date);
                   break;
    case T_PHONE:  sprintf (temp, "mb_phone %s;", data->name[j]);
                   data->siz[j] = sizeof (mb_phone);
                   break;
    default:       sprintf (temp, "long     %s;", data->name[j]);  break;
                   data->siz[j] = sizeof (long);                 /* WKP */
    }

  i = 24;
  if (data->type[j] == T_CHAR)
     i -= 3 +(data->siz[j] >10) +(data->siz[j] >100) +(data->siz[j] >1000);

  strcat (temp, repeat (' ', i-strlen(data->name[j])));

  strcat (temp, "/");
  strcat (temp, "* field ");
  strcat (temp, data->name[j]);
  strcat (temp, " type ");

  if (data->type[j] != T_CHAR)
     {
     strcat (temp, types[(int)data->type[j]]);
     }
  else
     {
     sprintf (nameb, "string length %d", data->siz[j]);
     strcat  (temp, nameb);
     }
  if (data->type[j] == T_SERIAL && data->serial != 0L)
     {
     sprintf (nameb, " start %ld", data->serial);
     strcat  (temp, nameb);
     }
  strcat (temp, repeat (' ', 73-strlen (temp)));
  strcat (temp, " *");
  strcat (temp, "/");
  strcat (temp, SNGCR);
  strcat (temp, "   ");
  writx (H, temp, strlen (temp));
  }

if (strname[0])
   {
   strcpy (temp2, strname);
   }
else
   {
   strcpy (strname, names);
   strcat (strname, "_str");
   strcpy (temp2,   names);
   }

strcat (temp2, "_rec");

sprintf (temp, "%c %s;%s", RBC, strname, DUBCR);
writx   (H, temp, strlen (temp));

/*==================== WKP ===================*/

sprintf(temp, "%s%s",
  "/*====== Added by KP Wong (in build.c),    25Aug94 ======*/\n",
  "/*== unsigned xx_str_len[] = {ArySize, len0, len1.....} ==*/\n");
writx (H, temp, strlen (temp));

sprintf(temp, "  unsigned %s_str_len[] = {%u, ",
        names, data->num_f);
for (k = 0; k < data->num_f; k++) {
  if (k>0) strcat (temp, ", ");
  fprintf (stderr, "data->siz[%u]=[%u]\n", k, data->siz[k]);
  sprintf (tempkp, "%u", data->siz[k]);
  strcat (temp, tempkp);
```

```
   };
   strcat (temp, "};");
   strcat (temp, SNGCR);
   sprintf(buf1, "%s%s", "#ifdef NOW_DEF_LEN", SNGCR);
   sprintf(buf2, "%s%s", "#else", SNGCR);
   sprintf(buf3, "%s%s", "#endif", DUBCR);
   writx  (H, buf1, strlen (buf1));
   writx  (H, temp, strlen (temp));
   writx  (H, buf2, strlen (buf2));
   sprintf(temp, "  extern unsigned %s_str_len[];%s", names, SNGCR);
   writx  (H, temp, strlen (temp));
   writx  (H, buf3, strlen (buf3));

/*=============================================*/

   sprintf (temp, "#ifndef MODULE%s   %s %s;%s",
            SNGCR, strname, temp2, SNGCR);
   writx   (H, temp, strlen (temp));

   sprintf (temp, "#else%s   extern %s %s;%s#endif%s#endif%s",
            SNGCR, strname, temp2, SNGCR, DUBCR, DUBCR);
   writx   (H, temp, strlen (temp));

   Printf  ("Header file created.%s", SNGCR);
   close   (H);
}

void
endoffile (num_f, num_i)
int         num_f, num_i;
{
   if (num_f == 0)
      {
      fprintf (stderr, "No fields declared before end reached%s", SNGCR);
      exit    (1);
      }
   if (num_i == 0)
      {
      fprintf (stderr, "No indices declared before end reached%s", SNGCR);
      exit    (1);
      }
}

void
strlwrcpy (new, old)
char       *new,*old;
{
   register char *a,*b;
   if (!new || !old)  return;
   for (a=new,b=old; *b; a++,b++)
      *a = tolower (*b);
   *a=0;
}

void
struprcpy (new, old)
char       *new,*old;
{
   register char *a,*b;
   if (!new || !old)  return;
   for (a=new,b=old; *b; a++,b++)
      *a = toupper (*b);
   *a=0;
}

void
strmax (str, siz)
char   *str;
int          siz;
{
   register int   i;
   register char *a;

   for (i=0, a=str; *a; i++, a++)
      if (i == siz)
         {
         *a = 0;
         break;
         }
}

int
get_names (agc, agv)
int         agc;
char            **agv;
{
   char  temp[128];
   int   i, fh;

   while (agc > 1 && agv[1][0] == '-')
      {
      switch (agv[1][1])
         {
         case 'q':  quiet  = 1;  break;
         case 'h':  header = 1;  break;
         default:   fprintf (stderr,"unrecognized option '%s'%s",agv[1],SNGCR);
                    usage   ();
                    exit    (1);
                   break;
         }
      switch (agv[1][2])
         {
         case 'q':  quiet  = 1;  break;
```

```
        case 'h':  header = 1;  break;
        }

      agc--;  agv++;
      }

  if (agc != 2)
      {
      usage ();
      exit  (1);
      }

  strcpy (temp, agv[1]);
  if (strcmp (&temp[strlen(temp)-2], ".s"))
      strcat (temp, ".s");

  strcpy (rel, temp);

  for (i = strlen(temp)-1; i > -1 && temp[i] != ':' && temp[i] != DIRSEP; i--)
      ;
  if (i < 0)  i = 0;

  rel[i] = 0;

  if ((fh = openx (temp, O_RDONLY)) == -1)
      {
      fprintf (stderr, "cannot open %s.%s", temp, SNGCR);
      exit    (1);
      }

  comment();

  (void)skip (fh, "relation");

  strcpy (temp, getword (fh));

  if (temp[0] == 0)
      {
      fprintf (stderr, "file holds no schema definition.%s",SNGCR);
      exit    (1);
      }

  Printf ("%s", CLS);
  Printf ("Building relation %s under ", temp);

  strlwrcpy (names, temp);
  struprcpy (nameb, temp);

  if (rel[0] != 0)
      {
      Printf ("directory %s%s",   rel, DUBCR);
      sprintf (hdr, "%s%c%s.h",   rel, DIRSEP, temp);
      sprintf (rel, "%s%c%s.rel", rel, DIRSEP, temp);
      }
  else
      {
      Printf ("current directory%s", DUBCR);
      sprintf (hdr, "%s.h",   temp);
      sprintf (rel, "%s.rel", temp);
      }

  Printf (lineF, SNGCR);

  comment();

  return fh;
}

char *
repeat (ch, nm)
char    ch;
int          nm;
{
   static char buf[80];

   buf[(nm = (nm < 0) ? 0 : nm)] = 0;

   for (nm--; nm >= 0; nm--)  buf[nm] = ch;

   return buf;
}

int
contains_serial (desc)
char             *desc;
{
   char *line, *pch;

   for (line = desc; (pch = strchr (line, ',')) != NULL; line = pch+1)
      {
      *pch = 0;
      if (data->type[atoi(line)] == T_SERIAL)
         return 1;
      }
   if (data->type[atoi(line)] == T_SERIAL)
      return 1;

   return 0;
}
```

CUHK Libraries
000733834