MANAGING TEXT AS DATA

by

Gordana Pavlovic-Lazetic and Eugene Wong

Memorandum No. UCB/ERL M86/21

14 March 1986

MANAGING TEXT AS DATA

by

Gordana Pavlovic-Lazetic and Eugene Wong

Memorandum No. UCB/ERL M86/21

14 March 1986

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

MANAGING TEXT AS DATA

by

Gordana Pavlovic-Lazetic and Eugene Wong

Memorandum No. UCB/ERL M86/21

14 March 1986

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

Managing Text as Data

Gordana Pavlovic-Lazetic and Eugene Wong

University of California
Berkeley

1. Introduction

With all their advances, database management systems of the present generation are designed to handle only data of primitive types, namely, numbers and character strings. Several approaches to extending their capabilities to handle data with higher order semantics exist. One is to add general abstract data type support, so that users can define such data types easily. In this approach, the DBMS makes no attempt to understand the semantics of user-defined data types, and evaluation of operators on such data are done in applications programs. As a supplement rather than an alternative, one can also extend the query language and its processor so that certain common non primitive data types are directly supported by the DBMS. Of these, *text* and *geometric data* are probably the two most prominent examples. This paper deals with the case of text. Direct embedding of complex data in a database management system has obvious advantages, the most important one being performance.

To manage text as data, the first step is to handle words satisfactorily. Words are after all natural atoms of text. Whereas representing texts as strings of characters capture none of their meaning, representing them as sequences of words is a reasonable first order semantic representation. Our first step, then, is to introduce "words" as a data type.

Important operations on words are lexical operators, not string operators. They deal with how words are related to each other and how they are used. For example, "went" is a verb in past tense with "go" as its root. "Verb", "past tense", and "go" are values returned by these distinct operators on the word "went". We refer to "words" together with a class of operators on words as the *lexical data type*. The principal objective of this paper is to deal with issues that

Research supported by the National Science Foundation under Grant DMC-8300463

arise in implementing the lexical data type.

The specific issues that we shall consider are the following:

* efficient storage of words in a relational database

* implementation of lexical operators

* resolving ambiguous words represented by the same character strings.

The principal application that we envisage for textual databases is automatic extraction of facts. We shall consider some simple examples of this using lexical operators.

2. Encoding Words

A natural way of storing texts in a relational database is to represent text by a relation:

textname(seqno, word)

where "seqno" denotes the order of appearance and "word" stands for words, punctuation and special symbols such as "new paragraph". As character strings, words have greatly varying lengths. For storage in a fixed-length field, character strings are grossly inefficient. A solution to this problem is to encode words into a fixed-length representation. Great compression can be achieved. For example, a 4-byte integer suffices to represent a vocabulary of $2^{32} \sim 4*10^9$ words.

There is a second and equally compelling reason to encode. Very little of the lexical information is contained in the character-string representation of a word. Clearly, the fact that "went" has "go" as its root cannot be deduced from the string w-e-n-t alone. If the goal is to implement lexical operators, then words need to be represented in a form whereby the values returned by the operators are explicit in the representation. Basically, the coded form of a word should be a composite of the values returned by the set of all admissible operators on the word.

There is yet a third reason to encode, namely, removing ambiguity. The same character string often has several meanings. In effect, it represents several different words, or more precisely, different "lexical units". For example, "well" has at least two unrelated meanings: "good and proper" and "a hole in the ground".

For these reasons we believe that encoding words is a must in storing text in a database system, if its meaning is to be exploited. The question is: how can this encoding be done? For compression alone, some kind of automatic encoding can probably be devised. However, no automatic encoding using only the character-strings as input can achieve the other two goals, since additional information must be supplied. To provide the lexical information, we shall use a dictionary. To resolve ambiguities, we shall use an expert system.

The amount of lexical information that has to be supplied depends on the lexical operators to be supported. Thus, the first step is define the lexical data type.

3. Lexical Data Type

We adopt the following terminology: a *lexical unit* is the image of a word under encoding, *lexical data set* is a set of lexical units together with certain default values, *lexical data type* is a pair (X, L) where X is a lexical data set and L the set of all supported operators. An element in X is of the form (id, descr) where id is a four byte integer that uniquely identifies the element (lexical unit), and descr a two byte descriptor that incorporates additional semantic information.

Encoding is done using a dictionary that is represented as a relation as follows:

dictionary(word, class, root, prefix, ending, feature, id, descr)

where "word" denotes the character-string representing a word, "class" denotes the syntactic classification of the word (i.e., verb, noun etc.), "feature" denotes semantic feature to be specified later. The meaning of "root", "prefix", and "ending" is clear. The code (id, descr) is a composite made up as follows:

id=(code(root)*100 + code(prefix))*100 + code(ending)

descr=code(form)*100 + code(feature)

Codes for prefixes, endings and semantic features are read from tables, and a root is encoded on the basis of interpolation of words density in a dictionary; starting codes for roots beginning with a specific letter are determined on the basis of the total number of codes available, and proportionally to the number of pages occupied by that beginning letter in a sample dictionary.

Code of a word's form is a number that is joined, in the table containing an entry for every possible form of any word class, to the form of that word (eg. 40 for infinitive form of anomalous verbs like "to have" or "to be", 41 for the first person in singular of the present tense of those verbs -as "have" or "am",46 for participle of those verbs -as "had" or "been", 150 for regular nouns in singular, 151 for regular nouns in plural, 210 for comparatives of adjectives ending on "-er", etc.).

Encoding is done as follows: Given a word as a character string, we first search for the corresponding entry in the dictionary and extract the code (id, descr). If there is more than one entry, then disambiguation is necessary.

The set of operators L consists of four types of operators: *lexical operators* such as finding root, prefix, ending or semantic feature of a lexical unit, building specific lexical forms such as plural for nouns or past tense for verbs, concatenating or deleting one lexical unit with/from another one; *syntactic operators* such as finding word class for a given lexical unit, tense for a given verb, degree for a given adjective, kind, gender, case for a given pronoun; *metric operators* such as length of a lexical unit in characters; *truth operators* such as equality or order of lexical units based on weights of roots, prefixes, endings, word forms and semantic features.

Examples of those operators are:

root(went)=go;

end(action)=ion;

tense(went)=past;

lexform(pl, datum)=data;

lexform(past, go)=went;

lexform(past, datum)=null;

concat(act, ion)=action;

concat(trans, ion)=null;

In what follows, we give a precise and formal specification for the lexical data type.

3.1. Lexical Data Set

LEX (lexical data set) is a union of the following sets of pairs of integers (id, descr):

- encoded *full lexical units* - encoded lexical units from the dictionary, which are images of words,

-sets of pairs (id, descr) having codes of all the entries from PREFIX, ENDING and SEM-FEATURE data relations in the corresponding portions of id, descr, and all the other zeros, and

-"null".

SYNT (syntactic data set) is a union of:

-WCL (word-class set), and

-NFORM, VFORM, AFORM and PFORM sets (sets of all the different forms corresponding to noun, verb, adjective-adverb, and pronoun word classes, respectively, ie.

WCL={reg.noun, reg.verb, reg.adjective, reg.adverb, irreg.noun, irreg.verb, irreg.adjective, irreg.adverb, anom.verb, pronoun, conjunction, prefix, preposition, null};

NFORM={sing, pl, null};

VFORM={pres_1st_sing, pres_2nd, pres_3rd_sing, past, part, null};

AFORM={positive, comparative, superlative, null};

PFORM={pers_f_1st_sing, pers_m_1st_sing, pers_1st_pl, pers_f_4th_sing, pers_m_4th_sing, pers_4th_pl, pos_f_sing, poss_m_sing, poss_n_sing, poss_pl, show_sing, show_pl, null}

Q - set of numbers.

TF - truth values set {T,F}.

3.2. Constants, Variables

Constants:

l_i from LEX;

s_i from SYNT;

q_i from Q;

T,F from TF.

Variables:

L_i from LEX;

S_i from SYNT;

Q_i from Q;

TR_i from TF.

3.3. Operators

lexical operators: $LEX^+ \rightarrow LEX^+$ or

$LEX^+ \times SYNT \rightarrow LEX^+$;

syntactic operators: $LEX^+ \times SYNT \rightarrow SYNT$;

metric operators: $LEX^+ \rightarrow Q$;

truth operators: $LEX^+ \rightarrow TF$.

The *operators* on lexical data type:

unary:

lexical:

root(L_1) ($\in$ LEX);

prefix(L_1) ($\in$ LEX);

end(L_1) ($\in$ LEX);

feat(L_1) ($\in$ LEX);

syntactic:

w_class(L_1) ($\in$ WCL);

tense(L_1) ($\in$ VFORM);

number(L_1) ($\in$ NFORM $\cup$ PFORM);

degree(L_1) ($\in$ AFORM);

kind(L_1) ($\in$PFORM);

gender(L_1) ($\in$PFORM);

case(L_1) ($\in$PFORM);

metric:

length(L_1) ($\in$Q).

binary:

lexical:

lexform(S_1, L_1) ($\in$LEX);

concat(L_1, L_2) ($\in$LEX$^+$);

delete(L_1,L_2) ($\in$LEX$^+$);

truth:

equal(L_1, L_2) ($\in$TF);

less_eq(L_1, L_2) ($\in$TF);

gr_eq(L_1, L_2) ($\in$TF).

3.4. Lexical and Logical Expressions

Lexical expression is a sequence of constants and variables from the set LEX and the sets supporting it, intermixed with operators leading to LEX-type result.

Lexical predicates are of the form truth_op($expr_1$, $expr_2$), where $expr_1$, $expr_2$ are any lexical expressions, and truth_op is any of the binary truth operators defined above.

Logical expressions (and thus qualifications) are extended to accept lexical predicates as arguments of logical operators (not, and, or).

3.5. Procedures for Operator Evaluation

Lexical operators are defined by procedures having encoded lexical units (ie. pairs of integers) as their arguments.

The following are some examples of those procedures written in a C-like language:

root:

```
root(L)
int L[2];

{
        L[0]=(L[0]/10**4) * 10**4;
        L[1]=0;
}
```

lexform:

```
lexform(form,L)
char *form;
int L[2];

{
        if(form=='sing')
                singular(L);
        else if (form=='pl')
                plural(L);
        else if (form=='pres_1st_sing')
                pr1sg(L);
        else if (form=='pres_2nd')
                pr2(L);
        else if (form=='pres_3rd_sing')
                pr3sg(L);
        else if (form=='past')
                past(L);
        else if (form=='part')
                participle(L);
        else if (form=='positive')
                psit(L);
        else if (form=='comparative')
                compar(L);
        else if (form=='superlative')
                superl(L);
        else if (form=='pers_f_1st_sing')
                prf1s(L);
        else if (form=='pers_m_1st_sing')
                prm1s(L);
        else if (form=='pers_1st_pl')
                pr1p(L);
        else if (form=='pers_f_4th_sing')
                prf4s(L);
        else if (form=='pers_m_4th_sing')
                prm4s(L);
        else if (form=='pers_4th_pl')
                pr4p(L);
        else if (form=='pos_f_sing')
                psfs(L);
        else if (form=='pos_m_sing')
                psms(L);
```

```
        else if (form=='pos_n_sing')
                psns(L);
        else if (form=='pos_pl')
                psp(L);
        else if (form='show_sing')
                ss(L);
        else if (form=='show_pl')
                sp(L);
        else
                L=NULL;
}
```

Procedure "singular" might be defined as follows:

```
singular(L)
int L[2];

{
        if(L[1]/1000!=8 && L[1]/1000!=15)
                L=NULL;
        else if(L[1]/100==81 || L[1]/100==151)
        {
                L[0]=L[0]-1;
                L[1]=L[1]-100;
        }
}
```

and similarly for other procedures.

4. Text Representation

Our goal is to take a text in its natural form and automatically convert it into a relation:

text(seqno, lex)

where seqno represents the sequential order and lex (lexical unit) is either the image of a word under encoding or a special symbol. The process of encoding (a) reduces a word to a fixed length representation, (b) makes explicit the lexical properties required to support the desired operators, and (c) resolves any ambiguity that may be present in the character string form. The automatic conversion of text is done using: a text scanner, a dictionary, and an expert system for resolving ambiguity.

4.1. Dictionary

The structure of the dictionary has already been described in section 3. It contains all words except plurals for regular nouns, tenses for regular verbs and comparatives and superlatives for regular adjectives and adverbs. Roots, prefixes and endings are determined by hand and their meaning is obvious; one rule about roots is that they are always words themselves.

Semantic feature is a marker that expresses semantics of a word or of a specific use of a word (e.g., ACTION for the word "work", LOCATION for the word "abroad", TIME for the word "then", QUALITY for the word "brilliance", both MEASURE and EMOTION for the word "content"). The set of semantic features we use is much like the one in [SiCh 82], extended with a hierarchical structure. For example, semantic feature TIME has as its subordinated semantic features FUTURE, PRESENT and PAST. Our set contains about 50 semantic features.

The dictionary encoding is done by an EQUEL-program. For our experimental study, we have built a dictionary with 1400 entries of basic words.

4.2. Lexical Rules

Since different forms of regular words are not present in the dictionary, lexical (ie. morphological) rules for synthesizing them or recognizing them is necessary in order for a text to be encoded.

An example of those rules is the following:

- if a word from the text ends with "ies" and in the dictionary there is a noun equal to that word except for the ending being "y" instead of "ies", then the word is the noun from the dictionary, in plural.

Those rules are stored in a relation "lexrule" which is of the form:

word_ending | dict.entry_ending | word's class | dict.entry's class | descr | code offset

Word and dict.entry endings are the letter groups that should be deleted at the end of the word that is to be encoded and that should be then added to the end of such a word, respectively, in order to obtain a dictionary entry corresponding to the word being encoded (e.g., the ending

"ies" should be deleted at the end of the word "copies" and then the ending "y" should be added to "cop" in order to get a dictionary entry "copy").

Word and dict.entry's classes are word classes that the word being encoded and the corresponding dictionary entry, respectively, belongs to (e.g., noun for both in the previous example).

Descriptor is an explanation of the form found in the text (the code for "plural" in our case), and a code offset says how to calculate the code of the word being encoded on the basis of the code of the corresponding dictionary entry.

The lexical rules relation created contains about 40 rules.

4.3. An Expert System for Resolving Ambiguity

According to the classification of expert systems in [HaWL 83], our expert system is of the *interpretation type*. The components of the system are:

(1) a *blackboard* used to record intermediate results,

(2) a *knowledge base* containing facts from the dictionary and rules used for resolving ambiguity,

(3) an *interpreter* that applies a rule from the knowledge base and posts changes to the blackboard,

(4) a *scheduler* that controls the order of rule processing according as whether the ambiguity is to be resolved syntactically or semantically.

In most cases, ambiguity is between word classes (e.g., noun and verb) and is resolved using context. For example, suppose that the phrase "a set of rules" is encountered. The word "rules" is either "verb - third person singular" or "noun - plural". In this case, the ambiguity is easily resolved by the rule: "preposition-noun" combination is far more likely than "preposition-verb" combination. As in MYCIN [DAVI 77], we use a probability model, and our rules have the form

(antecedent, consequent, probability)

where antecedent specifies a set of conditions under which the rule is applicable, consequent is the

conclusion and probability gives a weight to the conclusion. For example, we might have:

antecedent: if x is a noun or a verb and if x follows a preposition

consequent: then x is a noun with

probability: weight 0.9.

The architecture of the expert system was chosen on the basis of knowledge, data and solution space appropriate to our problem. Using the terminology found in [STEF 82], we find that we have a small solution space (few possible choices), unreliable data and knowledge (the context used for resolving ambiguity of a word might be ambiguous as well, and rules, representing knowledge, are not absolutely correct), and fixed (time - independent) data. For such an environment, the [STEF 82] suggests an expert system organization that applies exhaustive search and combines evidence from multiple sources and a probability model.

Thus, our strategy is a MYCIN-like one [DAVI 77]. It is designed to make an exhaustive search through the set of rules applicable to a given situation, and stops short of exhaustion only when ambiguity is resolved with certainty.

Backward chaining control strategy is used. The search is hypothesis driven: from possible solutions to related antecedent conditions and to their required data.

Our expert system was built using EQUEL [INGR 81], which is QUEL (QUEry Language for INGRES) coupled with general purpose programming language "C" [KeRi 78], rather than knowledge representation languages [HaWL 83].

In our experimental system, we have 110 rules, 50 of which involve *word class* (e.g., noun vs. verb), 30 involve *semantic feature* (e.g., time or place), and 30 are word specific (e.g., noun "drama" or adverb/noun "back"). Both rules and facts as well as the dictionary are stored as relations in INGRES. Figure 1 depicts the flow of control among the basic procedures. All procedures have read and write access to blackboards, which are "C" arrays of structures.

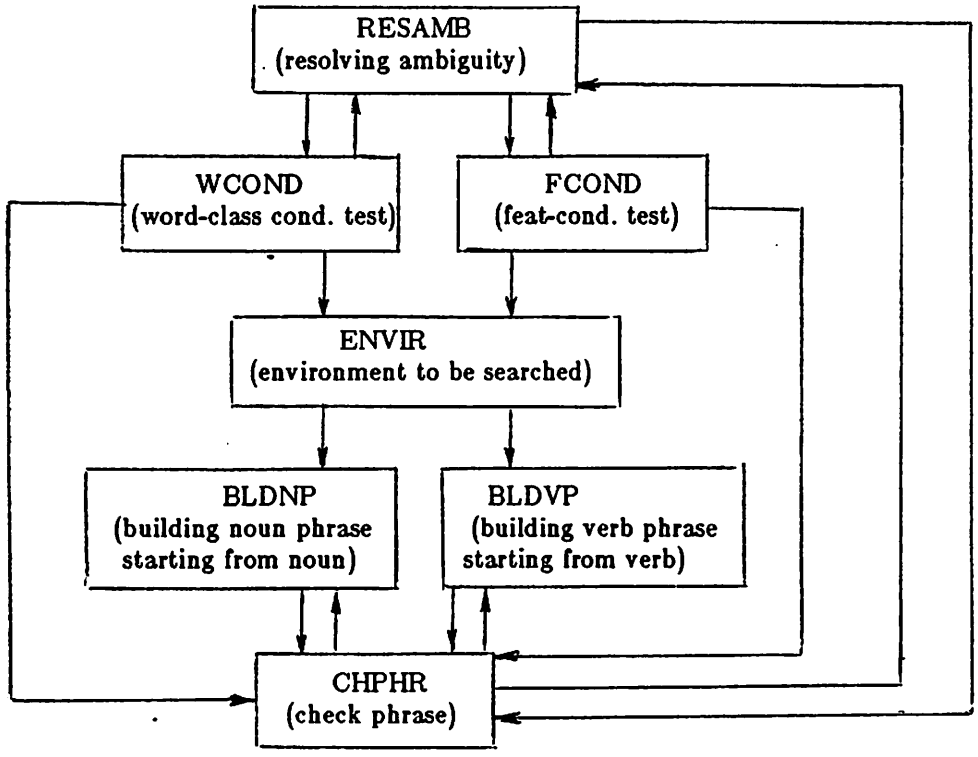

Figure 1. Basic cooperating procedures in the expert system

4.4. Text Encoding

Texts are scanned first, and then encoded and stored on a sentence by sentence basis. A current word is matched against the dictionary entries, taking into account lexrule relation. It is appended to the blackboard 1 together with the information about its position in the text, and, if unambiguous, with its code and descriptor. If a word is ambiguous, then it is marked indicating the kind of ambiguity that is encountered. The procedure for resolving ambiguities in a sentence is then called, which fires the expert system procedures for every word on the blackboard 1 marked as ambiguous. The contents of the blackboard 1 is then written into an output file, and at the end stored in a relation.

As an experiment, Albert Einstein's biography [ENCY 79] has been used as a text that contained 4096 words (including numbers, punctuations and special symbols) within 140 sentences. The following are some numbers that are obtained as a result of applying the system to the text: 82% of all the sentences (115 sentences) were found to contain ambiguous words, 251 in total (5% of all the words). Out of all the ambiguous words, 147 were found to be syntactically ambiguous and 104 semantically ambiguous. In the process of resolving ambiguities, 139 out of 147 syntactic ambiguities were resolved correctly (94%), examples of incorrect resolution being some occurrences of the word "after" (adverb/adjective/preposition/conjunction), of the word "found" (regular/irregular verb) and the word "divorce" (noun/verb) in the phrase "was to lead to divorce". Semantic ambiguities were mostly on semantics of prepositions. Out of 104 semantic ambiguities, 82 were resolved correctly (79%), examples of incorrect being several occurrences of the preposition "by" (TIME/ SOURCE/ INSTRUMENT) as in the phrase "rejection of his ideas by statesmen", and the word "content" (EMOTION/MEASURE) in the phrase "energy content".

. Source of incorrect resolution of ambiguities is mostly in that we decided on a very limited and simple analysis of context, and it would significantly improve with addition of more complex analysis. In order to resolve semantic ambiguities better, the system would also have to be enchanced with context-dependent semantics.

As an example of what has been successfully resolved, the following is an extract from the text been encoded:

> Albert Einstein was born in Ulm, Germany, on March 14, 1879.
>
> ...
>
> His theories of relativity were a profound advance over the old Newtonian physics and revolutionized scientific and philosophic inquiry.

The words "on", "advance" and "over" were recognized as ambiguous ones (first one as having more than one semantic feature, last two as belonging to more than one word-classes) and were successfully resolved(TIME, regular noun, preposition, respectively).

5. Example Applications

Operations on texts that we have experimented with include: extraction of keywords and phrases, (information retrieval application), stylistic homogeneity testing (computer linguistics application) and extracting precise informations from texts. We shall describe the last one in greater detail.

Extracting precise informations from texts consists of asking a question about a fact from the text (e.g., when a person named "X" was born) and finding the answer (e.g., 1879).

Our approach to extracting facts from texts is to view texts as a virtual relational database corresponding to a specific schema. The schema defines, a priori, the universe of all queries that may be posed, and the answer to a query is found from one or more texts at execution time. Thus, except for the encoding at load time, the texts are not preprocessed. Query processing makes heavy use of the syntactic and semantic features of words that we have designed into the code.

We have constructed an experimental system with a collection of biographies in the (virtual relational) database and the following schema:

relations with attribute:domain pairs:

birth(author:person, birth_place:place, birth_date:date);

degree(name:person, degree:degree, degree_institution:institution, field:field_of_science, degree_date:date);

education(name:person, attend.instit:institution, field:field_of_science, period:(date,date));

emp_history(name:person, employer:institution, position:position, date_started:date, date_left:date);

location(inst_name:institution, place:place);

research_interest(name:person, area:field_of_science, period:(date,date) U {w|feat(w)="PRT"} U (date-date));

publication(author:person, title:citation, published:institution, date:date).

A priori, lexical information concerning some of the relations and domains may be supplied, for example,

birth:

root: "birth";

person:

word class: proper phrase;

semantic feature: HUM;

place:

word class: proper phrase;

semantic feature: LOC;

As we have explained, no stored relations correspond to the schema above. Instead, the collection of texts is stored as a relation

cod_text(tno, sno, id, descr)

where tno (text number) identifies a particular biography, sno identifies a sentence, and (id, descr) is the coded form of a lexical unit. Now the question: "Where was Albert Einstein born?" can be expressed as a virtual query:

```
range of e is birth
retrieve (e.birth_place) where e.author="Albert Einstein",
```

Using the facts: code("LOCATION")=46, code("Albert Einstein")= -1607030, code(root="birth")=12636, we can translate virtual query into a real query:

```
range of e is cod_text
range of u is cod_text
range of v is cod_text
retrieve(e.id) where e.id<0 and feat(e.id)=46
        and e.sno=u.sno and e.tno=u.tno
        and root(e.id,e.descr)=(126360000,0)
        and v.tno=u.tno and v.sno=u.sno
        and v.id=-1607030
```

which yields the answer "Ulm, Germany".

6. Conclusion

We have presented a way of handling texts in a relational database system so that: (a) storage efficiency is maintained, (b) ambiguity of words is resolved, and (c) lexical (word based) information, both syntactic and semantic, is made explicit. These goals are achieved through encoding, which in turn uses a dictionary and an expert system for resolving ambiguity. Once a dictionary is built, any machine readable text can be automatically encoded with no human intervention.

Our long term goal is to apply what we have done to the problem of extracting facts from texts. A simple and rather primitive version of such a system is given as an example. However, considerable more work will be required for a fact-extraction system of general utility, and we are in the midst of such a development.

References:

[DAVI 77] Davis,R., et al., Production rules as a representation for a knowledge-based consultation program, Artificial Intelligence, 8(1977), pp.15-45;

[ENCY 79] The new Encyclopaedia Britannica, Macropaedia, 15th edition, Vol. 6, pp.510-514;

[HaWL 83] Hayes-Roth,F.,Waterman,D.A.,Lenat,D.B.(Eds.), Building expert systems, Addison Wesley Publ. Comp. Inc., 1983;

[INGR 81] INGRES Version 7 Reference Manual,ERL, UC Berkeley, Memo. No. UCB/ERL M81/61, Aug 1981;

[KeRi 78] Kernighan,B.W., Ritchie,D.M., The C programming language, Prentice Hall Software series, 1978.

[SiCh 82] Simmons,R.F., and Chester,D., Relating sentences and semantic networks with procedural logic, CACM, Aug. 1982, vol. 25, No.8, pp.527-547;

[STEF 82] Stefik,M., et al., The organization of expert systems, A tutorial, Artificial Intelligence, 18(1982), pp. 135-173;