UDC: 007.52
Original scientific paper

# A CONCEPTUAL PROLOG ENGINE FOR AUTOMATED DICTIONARY-TO-HYPERTEXT MAPPING

**Mirko Čubrilo**

University of Zagreb, Faculty of Organization and Informatics, Varaždin, Croatia
E-mail: mcubrilo@foi.hr

*This article examines the possibilities of mapping the structure of classical information sources (dictionaries,...) into the hypertext structure. The hypertext structure enables more efficient usage of the mapped resources (enriching it with new multimedia sources, databases and knowledge bases' structuring,...). The idea is to interpret the classical information source as formal language, and it shall be demonstrated using the example of the classical dictionary, but it is nevertheless equally applicable to all (similar) types of classical information sources. The adequate technical basis for the implementation is to be seen in the logic programming language Prolog. The method, called Conceptual Prolog Engine, has been formed and developed in this environment.*

**Keywords:** classical sources of information, hypertext, knowledge base, database, Prolog.

## 1. INTRODUCTION

Every day we are witness to the increasing migration of classical sources of information to new computer memory media, such as floppy and hard disks, magnetic tapes, and, recently, CD - ROM disks. But, the very process of mapping information from old to new media is hard to achieve for several reasons. Some of them are of technical provenience, while the others are structural.

Technical difficulties arise when someone wants information to migrate physically to new media. They concern the quality of scanned text and images, as well as recognizing them as objects of different kinds. At this time this sort of problems is reasonably solved by using the high quality scanners and OCR programs (such as Recognita or Omni Page). New generation scanners and OCR programs can succesfully separate text and images and recognize (at least typewritten) text with high precision (99% or higher).

A more difficult problem to solve is the one of mechanically structuring mapped information onto new media, preserving and possibly improving the old organizational structure.

The most important information sources, living on old media and screaming for mapping to the new ones, are: dictionaries (monolingual, such as *The Oxford Dictionary of Modern English* [ODME for short], and multi-lingual, such as *CAPITOL'S CONCISE DICTIONARY: from and to English, Swedish, Dutch,...*), phone books, hospital data files, government data files (X-Files (joking), driving licence data, crime data, statistical data, etc.).

In the context of our examination, one such useful structure is hypertext structure. Basically, hypertext is nonlinearly organized text, structured in a multi-leveled net of interlinked nodes. The conceptual base of the hypertext structure is described in [1] and became the basis for several implementations. Among the other general sources of information on hypertext are [2] and [6].

Here, we are proposing a method, which is sufficiently founded in theory, so that we can call it a *method*, and which, on the other hand, can be practically realized for mapping some of the information sources to new memory media and new organizational structure, in a uniform and at the same time almost purely mechanical way. The new structure can be a table of a relational database, a hyperbase, or some other structure. Here we will restrict ourselves to mapping classic dictionary structure to the hypertext structure.

The method will be illustrated using the ODME dictionary as a source of information. Here we cannot go into the full details of the implementation, but we will sketch, in Prolog, all the important predicates and describe all the needed data structures.

This paper consists of five sections and two appendices. In Section 2 we describe our Method at the conceptual level. In Section 3 we demonstrate the Method in action. In Section 4 we examine the various possibilities of postprocessing the hypertext structure. Section 5 is the concluding one. Appendix A gives the core structure of a whole application and Appendix B demonstrate a grammar description of a sample dictionary unit of middle complexity.

## 2. THE METHOD

The main goal of the method is to mechanically map the structure of the classical dictionary into the hypertext structure.

We assume that dictionary units are available as strings. The problem of getting strings (problems of scanning the text and character recognition) are out of our concern.

The steps of the method are the following:

### 1. Lexical and syntactical analysis of dictionary units

a) *Preliminary arrangement of dictionary units*

Dictionary units are to be arranged in such a way that their BNF grammar is as simple as possible. This step consists of the uniform exchange of some of the

characters or words in strings with others (e.g. the expansion of the shortcut *adj.* ⇒ *adjective*,...) and embedding some markers or tags in the text. If the text editor concerned is capable of recognizing some of the graphical features of the text (such as font style, font size, etc.), then those features can also be built into the BNF grammar of the dictionary units as terminal objects. That way we can achieve a more detailed description of those units. Such an editor is, for example, the Visual Prolog editor, which reads text in RTF format (Reach Text Format) in its code form, with all of its 79 markers (where applicable), which is much more than we need for the recognition of all the graphical features of dictionary units. Some of those graphical features serve as elements of the grammatical structure of dictionary units. For example, the type of grammatical units of the ODME dictionary is always in italics.

Further, markers (grammar production terminal elements) can be inserted in dictionary units, as, for example, pronunciation, which will decide if its hypertext structure appearance should contain audio and/or video records. It would be logical that every dictionary unit should contain an audio record describing its pronunciation, but it can also contain other audio records, which would explain its semantics (bird song, sound of an animal, engine noise, wind murmur, etc.). Obtaining the first type of audio records can be (at least for some languages) wholly automated, by using the so-called text-to-speech technology (Dragon Naturally Speaking, IBM VoiceType...). Audio and video records should be accessible through the corresponding Prolog predicates by using the prearranged (internal or external) database. Joining the dictionary units with other types of records, audio as well as video records, can be harder to automate, due to their dependency on the meaning of dictionary units.

*b) Preparation for syntactical analysis*

This step includes the shaping of the input (BNF) grammar for the syntactical analyser generator (PARSER.EXE from the PDC Prolog [also Turbo Prolog or Visual Prolog] development environment) which will generate the predicates of the future syntactic analyser on the basis of that grammar.

Syntactical analyser predicates could be developed manually, e.g. using the widely known difference lists technique. However, their automatic generation on the basis of the given grammar will give a more uniform syntactical analyser structure.

*c) Lexical analysis*

Lexical analysis includes structuring the source text in a row (list) of irreducible lexical units (lexemes). Especially important among them are nonterminal symbols which participate in structuring the grammar of dictionary units.

*d) Syntactical analysis*

In this step the list of lexemes created by the the lexical analysis of dictionary units is structured in the BNF grammar language structure with the aid of generated syntactic analyser predicates.

**2. Shaping the hypertext structure**

In this step the hypertext structure is shaped on the basis of the dictionary units global structure, encompassed by the BNF grammar. The choice of building units depends on the goals in mind when executing the reshaping. One goal can be simply the separation of the lexical units and their pronunciation with the intention of forming audio files. Another goal can be the extraction of the dictionary units and the forming of connections between them all and grammar forms for the given individual word, etc. The hypertext structure will be shaped in PDC Prolog.

The input dictionary unit's grammar structure does not have to be (and it is desirable that it is not) universal, meaning that it does not have to describe the structure of *all* dictionary units. The logical scenario is separating the dictionary unit's forms (nouns, verbs, adjectives, ...) and then further improving the grammar which carried out the former in order to accomplish a higher quality hypertext structure. Of course, the improvement level depends on the hypertext structure currently held in mind.

As previously mentioned, a hypertext structure can contain elements which are not present in the dictionary unit's structure, such as audio files and pictures. Those elements have to be predefined and put together in one or more internal or external databases (in the Prolog environment) or classical databases (using the ODBC interface). The input of those elements can be performed manually or automatically, but one should have in mind that the latter is harder to accomplish because it depends on the semantics of the dictionary units connected with audio and picture files.

**3. Implementing hypertext structure with the aid of the structure created by syntactic analysis**

In this step the predicates from the hypertext structure description are defined using the generated syntactic analyser predicates.

**4. Postprocessing**

On the upper level the application can contain a whole sequence of the hypertext structure exploitation mechanisms, ranging from those designed for the classical database structuring to those for the knowledge base structuring.

Different general types of abstraction can be performed on the hypertext structure, like generalization, specialisation, aggregation, defining new views and extraction of the implicit knowledge (through different kinds of deduction). Later, while illustrating our *Conceptual Prolog Engine*, we will give a few examples of postprocessing, specific for the chosen domain, and implement one.

The resulting application does not merely transfer the printed dictionary information content, it enriches it by enabling all sorts of information manipulation and thus provides the users with a higher quality service.

## 3. THE METHOD IN ACTION

We will choose two ODME units. We shall illustrate the whole process of mapping dictionary units on a simple example from a dictionary unit. The second unit (see Appendix B) shall be used to illustrate the cumulative BNF grammar development for all more complex dictionary unit types.

Example 1: (grammar G1)

**accommodating** *adj.* Willing to do as one is asked.

- Preliminary dictionary unit preparation

Changing *adj.* $\Rightarrow$ *adjective* reduces the multifacetedness of the meaning a full stop can have.

The whole method implementation could use the upper dictionary unit fragment RTF file, which has the following form:

```
{{\b\i\f2\lang1050 accommodating}{\f2\lang1050 }{\i\f2\lang1050 adjective}{\f2\lang1050 Willing to do as one is asked.\par }}
```

Delimiters \b, \i and \lang1050 respectively mean font style bold, font style italic and the language of code number 1050. The last piece of information is superfluous, so preliminary unit preparation should eliminate it. The grammar of the dictionary unit can encompass the structure of the RTF file to which it belongs. Delimiter \par marks the end of the given dictionary unit, so that it is easy to distinguish between different units.

We shall, if and when required, asssume that the graphical features of the text could already be separated in the source dictionary unit.

- Input BNF style grammar for the parser generator PARSER.EXE from the development environment of PDC Prolog (Turbo Prolog, Visual Prolog)

```
DICTIONARY_UNIT = CONCEPT PART_OF_SPEECH DESCRIPTION
CONCEPT = symbol
PART_OF_SPEECH = adjective
DESCRIPTION = string
```

The input grammar syntax of the parser generator PARSER.EXE demands that Prolog predicates which will represent individual (non-list) productions in a syntactic analyser must be attributed to those productions. The details of the input grammar's metagrammatic description is not discussed here. The syntactic analyser input grammar designed for the BNF grammar in the example has the following form:

```
productions
DICTIONARY_UNIT = CONCEPT DESCRIPTION -> dictionaryUnit(CONCEPT,DESCRIPTION)
CONCEPT = PURE_CONCEPT PART_OF_SPEECH -> concept(PURE_CONCEPT, PART_OF_SPEECH)
PURE_CONCEPT = pureConcept(SYMBOL) -> pureConcept(SYMBOL)
DESCRIPTION = description(STRING) -> description(STRING)
SORT_OF_GRAMMAR_OBJECT = adjective -> adjective
```

- Generated parsing objects (parsing domains and predicates)

```
/*************************************************************
                    DOMAIN DEFINITIONS
*************************************************************/
DOMAINS
 DICTIONARY_UNIT = dictionaryUnit(CONCEPT,PART_OF_SPEECH,DESCRIPTION)
 CONCEPT       = concept(SYMBOL)
 PART_OF_SPEECH  = adjective()
 DESCRIPTION     = description(STRING)
 TOK     = concept(SYMBOL);
                   adjective();
                   description(STRING);
                   nil

/*************************************************************
                    PARSING PREDICATES
*************************************************************/
PREDICATES
 s_dictionary_unit(TOKL,TOKL,DICTIONARY_UNIT)
 s_concept(TOKL,TOKL,CONCEPT)
 s_part_of_speech(TOKL,TOKL,PART_OF_SPEECH)
 s_description(TOKL,TOKL,DESCRIPTION)

CLAUSES
 s_dictionary_unit(LL1,LL0,dictionaryUnit(CONCEPT,PART_OF_SPEECH,DESCRIPTION)):-
        s_concept(LL1,LL2,CONCEPT),
        s_part_of_speech(LL2,LL3,PART_OF_SPEECH),
        s_description(LL3,LL0,DESCRIPTION),!.

 s_concept([t(concept(SYMBOL),_)|LL],LL,concept(SYMBOL)):-!.
 s_concept(LL,_,_):-syntax_error(concept,LL),fail.

 s_part_of_speech([t(adjective,_)|LL],LL,adjective):-!.
 s_part_of_speech(LL,_,_):-syntax_error(part_of_speech,LL),fail.

 s_description([t(description(STRING),_)|LL],LL,description(STRING)):-!.
 s_description(LL,_,_):-syntax_error(description,LL),fail.
```

TOK is a shortcut of TOKEN, i.e. *lexeme*. A look at the generated predicates proves them automatically generated by the difference list method. The parser generator PARSER.EXE inserts the syntax_error predicate in the code of the generated predicates to handle errors. Generated predicates perform syntactic analysis using the "trial and error" method. The unsuccesful trials do not mean the absolute inability to recognise the structure of the input dictionary unit (transformed in the lexeme list). Therefore, the recognition of the error should be postponed for as long as possible.

The possible strategy is simply to keep track of the error that occurred at the deepest level in the source text. Each time the syntax_error predicate is called, we should compare the cursor position of the current token with the cursor position of the last syntax error. The fact that the new error occurs deeper into the source text may mean that the parser simply tried to apply the wrong production. So, we should temporarily save the new error (in an internal database predicate, say error, whose

domain consists of error messages and cursor positions respectively). If at some point the parsing process absolutely fails, that means that the currently saved error (at the top of the internal database) actually is a syntax error (does not represent the dictionary unit). Predicate syntax_error is not completely implemented here. After all, our Prolog engine is just a *conceptual* engine.

The lexical analysis of the input dictionary unit should be preceded by its syntactical analysis. The lexical analyser of the input dictionary units can be implemented in several different ways. The only condition is that the returned list of tokens should belong to the TOKL domain, to maintain compatibility with parsing predicates. Most of the work will be performed by the fronttoken system predicate. The lexical analyser at hand is implemented and commented on in the HYPER.SCA program (see Appendix A).

- **Results of the lexical and syntactical analysis of our sample dictionary unit**

The HYPER.PRO program applied to the input dictionary unit formed as an input string gives a Tokens list and the output term (dictionaryUnit), which is ready for further structuring into the hypertext structure.

```
Tokens -> [t(concept("accommodating"),_),t(adjective,_),t(description(" willing to do as one is asked ."),_)]
Term ---> dictionaryUnit(concept("accommodating"),adjective,description(" willing to do as one is asked ."))
```

- **Modelling the hypertext structure**

Modelling the hypertext structure has three steps:

1. Forming and creating the physical structure of nodes
2. Physical maintainance of the node structure (adding new nodes, deleting already existing nodes, adding and deleting links between nodes, etc.)
3. Maintainance in use (realising different abstraction types in the hypertext structure).

In the Prolog context, the most suitable "data structure" for forming the physical structure of nodes (hyperbase) is a so-called *external database*. However, for illustrating implementational aspects of our conceptual engine we shall use internal databases, connected with a complex term construction mechanism. The core hypertext structure encompasses a net of nodes.

We shall assume that the input into our module for structuring dictionary units in hypertetxt structure is given in a list form.

```
dictionaryUnit(concept(CONCEPT),PART_OF_SPEECH,description(DESCRIPTION))
```

Generally, the implementations depend upon the chosen programming language and environment, as well as the purpose of the system. The purpose of this system is illustrative, meaning that we shall give minimal (conceptual) implementation, to reach the level of its structure, without considering the navigational aspect.

We shall assume that dictionary units have been structured in the internal database dictionaryUnitBase (with the save("dictUnit.dba",dictionaryUnitBase) predicate), and that dictionaryUnit is its only predicate. Further, we shall assume that all video and audio records have been arranged and saved in their corresponding internal databases ("pictures.dba" and "sounds.dba") with picture(pictureName,pictureFile), and sound(soundName,soundFile) predicates respectively. The names from the domains pictureName and soundName must be equal to the values of the CONCEPT term argument in the dictionaryUnit predicate structure, so that video and audio records can be properly linked to their corresponding dictionary units. The mapping predicate will operate on the three internal databases mentioned above, and map dictionary unit's structure into the hypertext structure.

The assumed minimal hypertext structure is made of nodes and a few primitive links. The links are to connect sound and picture names with the appropriate database files. Each node shall represent a dictionary unit, and hyperfields shall represent its textual, and where present, its video and audio traits. The node identifier must be CONCEPT. Its domain is SYMBOL. The empty term is the default value of picture and sound hyperfields respectively when one of them does not exist.

Hypertext structure is static without links. Natural links exist here, for example, between the sound and picture names and their appropriate files. Generally, it is natural in this environment that links are provided within the application usage, through different abstraction types and hypertext structure remodellings. Abstractions and remodelling types can be predefined in a finished application or it could be left to the user to arrange them himself from irreducible components.

Here, the hypertext structure (embedded through the main mapping predicate dict_to_hyp) is given in a pseudo-Prolog code

```
DOMAINS

CONCEPT,PART_OF_SPEECH,SOUND,PICTURE = SYMBOL
DESCRIPTION = STRING
NODE = node(ID)
SOUND_FILE,PICTURE_FILE = FILE

PREDICATES

dyct-to-hyp(node(CONCEPT),PART_OF_SPEECH,description(DESCRIPTION),SOUND, PICTURE)
/* This is the core mapping predicate (for a static hypertext structure) */
found(CONCEPT,PART_OF_SPEECH,description(DESCRIPTION))
/* This predicate finds (pops up) the dictionary unit in an internal database "dictUnit.dba" */
soundFound(CONCEPT,SOUND)
/* This predicate finds the appropriate symbolic name for the sound file of the concept in case */
pictureFound(CONCEPT,PICTURE).
```

```
/* The same, but for the picture's symbolic name */

link(SYMBOL,FILE)

/* Links the symbolic names for the sound and the picture with the appropriate files. In
implementation, these two kinds of files should be distinguished by their extensions */

CLAUSES

dyct-to-hyp(node(CONCEPT),PART_OF_SPEECH,DESCRIPTION,SOUND,PICTURE):-
        found(CONCEPT,PART_OF_SPEECH,DESCRIPTION),
        soundFound(CONCEPT,SOUND),
        pictureFound(CONCEPT,PICTURE).

found(CONCEPT,PART_OF_SPEECH,DESCRIPTION):-
        consult("dictUnit.dba",dictionaryUnitBase),
        retract(CONCEPT,PART_OF_SPEECH,DESCRIPTION),!.

soundFound(CONCEPT,SOUND):-
        consult("sounds.dba"),
        retract(sound(SOUND,_)),
        assertz(sound(SOUND,_)),
        CONCEPT=SOUND,!.

soundFound(CONCEPT,empty):-!.

pictureFound(CONCEPT,PICTURE):-
        consult("pictures.dba"),
        retract(picture(PICTURE,_)),
    assertz(picture(PICTURE)),
        CONCEPT=PICTURE,!.

pictureFound(CONCEPT,empty):-!.

link(SOUND,SOUND_FILE):-

        consult("sounds.dba"),
        retract(sound(SOUND,SOUND_FILE)),
        assertz(sound(SOUND,SOUND_FILE)).

link(PICTURE,PICTURE_FILE):-
        consult("pictures.dba"),
        retract(picture(picture,PICTURE_FILE)),
        assertz(picture(PICTURE,PICTURE_FILE)).

link(_,empty).
```

## 4. POSTPROCESSING

We can imagine an almost unlimited number of different kinds of abstraction that we could realize in the hypertext structure of dictionary units. Here are a few of them:

**Abstractions which**

- group together all instances of a given concept
- give all verb forms of the verb concept given in the infinitive
- give all synonyms of the given concept
- give all homonym meanings of the given concept

- gives all quasi-homonyms of the given concept (a quasi-homonym of a given concept we define as another concept which has an approximately similar meaning or participates in the description of the first one)
- give some characteristic phrases for concept usage

Here, we implement the abstraction which generates all quasi-homonyms of the given concept (dictionary unit).

```
quasi_homonyms(CONCEPT,QUASI_HOMONYMS):-
    dyct-to-hyp(node(CONCEPT),_,description(DESCRIPTION),_,_),
    suspects(DESCRIPTION,POSSIBLE_QUASI_HOMONYMS),
    quasi_homonyms(CONCEPT,POSSIBLE_QUASI_HOMONYMS,QUASI_HOMONYMS).

/* According to the description above every concept can have many homonyms */

suspects(S,[H|T]) :- fronttoken(S,H,S1),
            !,
     suspects(S1,T).

suspects(_,[]).

/* The suspects predicate takes a DESCRIPTION string as its input and gives as its output the list
POSSIBLE_QUASI_HOMONYMS of possible quasi-homonyms. */

quasi_homonyms(CONCEPT,POSSIBLE_QUASI_HOMONYMS,QUASI_HOMONYMS):-
    findall(QUASI_HOMONYM,quasi_homonym(CONCEPT,POSSIBLE_QUASI_HOMONYMS,
                                        QUASI_HOMONYM),QUASI_HOMONYMS).

quasi_homonym(CONCEPT,POSSIBLE_QUASI_HOMONYMS,QUASI_HOMONYM):-
        element(QUASI_HOMONYM,POSSIBLE_QUASI_HOMONYMS),
        additionalConstraints(QUASI_HOMONYM),
        dictionaryUnit(concept(QUASI_HOMONYM),_,description(DESCRIPTION)),
        searchstring(DESCRIPTION,CONCEPT,_).

/* The quasi_homonym predicate takes the CONCEPT string (dictionary unit) and its list of possible
quasi-homonyms (POSSIBLE_QUASI_HOMONYMS) as inputs and gives a quasi-homonym (if it
succeeds). That is done by checking all the elements of the POSSIBLE_QUASI_HOMONYMS list if
their description contains the initial CONCEPT as a substring in their descriptions. Besides and
before that, the QUASI_HOMONYM string is checked for additional constraints. The searchstring is a
standard Prolog predicate and the predicate element is defined as usually. The predicate findall is a
standard Prolog metapredicate. */
```

## 5. CONCLUSIONS

We have considered the possibilities of an automated mapping of classical information sources (dictionaries, statistical data, phone books,...) into the hypertext structure. The aim of that mapping is twofold. On the one hand, the aim is to "preserve" the old and valuable information sources for use in new environment (multimedia). On the other hand, the aim is to furhter enrich the old contents of such information sources with new contents (such as sound, pictures and video clips) as

well as to improve the old organizational structure, giving the user more possibilities in everyday use. The core idea for achieving all of this is to interpret the information sources in consideration as formal languages. It seems pretty hard, and yet is possible, and we hope that the paper has made it clear.

## APPENDIX A

### Core structure of the application

Although we have on several occasions pointed out that our Prolog engine is merely a *conceptual* engine, its groundings should, however, be verified in the Prolog development environment. Here we give a minimal implementation of the main module of the program, as well as its components and, in short, comment on some important predicates, with a special emphasis on the procedure of making them function in general conditions.

```
/*============================================================================
        HYPER.PRO - Demo of Parser Generator -
============================================================================*/
check_determ

CONSTANTS

DOMAINS
  CURSOR      = INTEGER
  CURSORTOK = t(TOK, CURSOR)
  MESSAGE = STRING
  RESULT      = REAL
  SOURCE      = STRING
  TOKL = CURSORTOK*

include "d:\\pdc320\\hyper\\grammar1.dom"
% Parser domains (created by Parser Generator).

PREDICATES
  expect(CURSORTOK, TOKL, TOKL)
  syntax_error(MESSAGE, TOKL)

/* The expect and syntax_error predicates handle parsing errors */

include "d:\\pdc320\\hyper\\hyper.sca"          % scan/3
include "d:\\pdc320\\hyper\\grammar1.par"       % s_dictionaryUnit/3
include "d:\\pdc320\\hyper\\hyper.ui"           % user_interface/0

CLAUSES
  expect(TOK, [TOK|L],L).

  syntax_error(_,_).

GOAL
        user_interface.
```

```
/*=========================================================================
        HYPER.UI Dictionary-To-Hypertext Demo of Parser Generator  - Simple User Interface
 ========================================================================= */

PREDICATES
 evaluateDictionaryUnit(SOURCE)
 parse(TOKL, DICTIONARY_UNIT)
 tokenize(SOURCE,TOKL)
 user_interface
glueDescription(TOKL,TOKL)
glue(TOKL,CURSORTOK,TOKL)

CLAUSES
 evaluateDictionaryUnit(DictUnit) :-
        tokenize(DictUnit, TOKENS),
        glueDescription(TOKENS,GLUED_TOKENS),
        write("\nTokens -> ", GLUED_TOKENS),
        parse(GLUED_TOKENS, TERM),
        write("\n\nTerm ---> ", TERM),
        !.
  evaluateDictionaryUnit(_) :-
                sound(30, 300),
                write("\n\n<<Illegal Expression>>").

/* The evaluateDictionaryUnit predicate does the initial input string tokenization, glues tokens of the
DESCRIPTION part back into the string, writes the list of tokens, parses it into the resulting term and
writes it on the screen */

 parse(TOKENS, TERM) :-
                s_dictionary_unit(TOKENS, UNUSED_TOKENS, TERM),
                UNUSED_TOKENS = [].

/* The parse and s_dictionary_unit predicates parse the input dictionary unit string into the appropriate
grammar term */

 tokenize(DictUnit, TOKENS) :- scan(0, DictUnit, TOKENS).

/* The scan predicate is doing the job of  input string tokenization */

 glueDescription([],_).

 glueDescription([t(concept(X),_)|TAIL],[t(concept(X),_)|TAIL1]):-
                glueDescription(TAIL,TAIL1),!.

 glueDescription([t(adjective,_)|TAIL],[t(adjective,_)|TAIL1]):-
                glueDescription(TAIL,TAIL1),!.

 glueDescription([t(description(X),_)|TAIL],GLUED_TOKENS):-
    glue([t(description(X),_)|TAIL],t(description(""),_),GLUED_TOKENS).

 glue([],t(description(X),_),[t(description(X),_)]):-!.
```

```
glue([t(description(X),_)|TAIL],t(description(Y),_),TAIL1):-
    concat(Y," ",Y1),
    concat(Y1,X,Z),
    glue(TAIL,t(description(Z),_),TAIL1).
```

/* The glueDescription predicate glues tokens in the DESCRIPTION part of the dictionary unit back to the string. glue predicate is doing the real job, using the standard trick which handles "boundary conditions" */

```
user_interface :-
        write("\n\n\n\nEnter dictionary unit string "),
        readln(DictUnit),
        !,
        evaluateDictionaryUnit(DictUnit),
        user_interface.
user_interface.
```

/* The user_interface predicate implements a minimal user interface (it reads the input string, evaluates the dictionary unit according to parsing predicates) and writes results (a list of tokens and the dictionary unit structure Prolog term) */

```
/* ============================================================================
        HYPER.SCA Dictionary-To-Hypertext Demo of Parser Generator
============================================================================ */
DOMAINS
 NUMBER_OF_EXTRA_CHARACTERS    = INTEGER
 NUMBER_OF_SPACES              = INTEGER

PREDICATES
 is_a_space(CHAR)
 scan(CURSOR, SOURCE, TOKL)
 skip_spaces(SOURCE, SOURCE, NUMBER_OF_SPACES)
 string_token(SYMBOL, TOK)
 counter_inc(INTEGER,INTEGER)

DATABASE

counter(INTEGER)

CLAUSES

 counter(0).

 counter_inc(Y,X):-
    retract(counter(X)),
    Y = X + 1,
    asserta(counter(Y)).
```

/* The counter_inc predicate increments the current value of the database predicate counter */

```
 is_a_space(' ').
 is_a_space('\t').
 is_a_space('\n').
```

```
/* The is_a_space predicate defines a "space" in a dictionary unit string */

scan(STARTING_POSITION, SOURCE, [t(TOKEN, LOCATION_OF_TOKEN)|TAIL]):-
        skip_spaces(SOURCE, NEW_SOURCE, NUMBER_OF_SPACES),
        LOCATION_OF_TOKEN = STARTING_POSITION + NUMBER_OF_SPACES,
        fronttoken(NEW_SOURCE, FRONTTOKEN, REST),
        !,% Make the case of each token unimportant.
        upper_lower(FRONTTOKEN, LOWER_CASE_FRONTTOKEN),
        string_token(LOWER_CASE_FRONTTOKEN, TOKEN),
        str_len(FRONTTOKEN, LENGTH_OF_FRONTTOKEN),
        NEW_STARTING_POSITION = LOCATION_OF_TOKEN + LENGTH_OF_FRONTTOKEN,
        scan(NEW_STARTING_POSITION, REST, TAIL).
scan(_, _, []).

/* The scan  predicate looks for tokens in the input string (dictionary unit) */

skip_spaces(SOURCE, NEW_SOURCE, NUMBER_OF_SPACES) :-
        frontchar(SOURCE, CHAR, SOURCE1),
        is_a_space(CHAR),
        !,
        skip_spaces(SOURCE1, NEW_SOURCE, NUMBER_OF_SPACES_IN_SOURCE1),
        NUMBER_OF_SPACES = NUMBER_OF_SPACES_IN_SOURCE1 + 1.

skip_spaces(SOURCE, SOURCE, 0).

/* The skip_spaces predicate counts and skips spaces between the two consecutive tokens */

string_token(adjective,adjective):-!.

string_token(SYMBOL,concept(SYMBOL)):-
        counter_inc(_,0), !.

string_token(SYMBOL,description(SYMBOL)):-
        counter_inc(_,X),
                X>0, !.
```

/* The string_token predicate makes a distinction between CONCEPT and DESCRIPTION parts in a dictionary unit string and transforms (where needed) graphical representation of a given token into its textual equivalent */

## APPENDIX B

### Yet another dictionary unit and its grammar

**accessory** (ak-**sess**-er-i) *adj.* additional, extra. --n. **1**. a thing that is extra or useful or decorative, but not essential, a minor fitting or attachment. **2**. a person who helps another in a crime. ¶ The spelling *accessory* is now disused.

This dictionary unit is one of middle complexity in the ODME dictionary. Its grammar is a generalization of our first example

```
DICTIONARY_UNIT := CONCEPT DESCRIPTIONS
CONCEPT := CONCEPT_ONLY | CONCEPT_ONLY SPELLING
CONCEPT_ONLY = SYMBOL
SPELLING = lpar SYMBOL rpar % left and right parentheses
DESCRIPTIONS = DESCRIPTION+
DESCRIPTION = MEANING | MEANING ARRAY_OF_MEANINGS
MEANING = PART_OF_SPEECH EXPLANATION
EXPLANATION = STRING
ARRAY_OF_MEANINGS = dash_dash NUMERATED_MEANING+
NUMERATED_MEANING = NUMERAL full_stop MEANING
NUMERAL = INTEGER
PART_OF_SPEECH = adjective | noun
```

As already mentioned, this dictionary unit is of middle level complexity. We could additionally refine its grammar (splitting the description from the notes on usage,...). The lexical analyser should be augmented with the definitions of additional terminal elements (noun, verb,...). The resulting parsing predicate would also have a more complex structure. Of course, it would also grasp our first dictionary unit sample, but with a lot of "empty" values. That means that the value empty must be an element of the appropriate domain of the core predicate (dictionaryUnit).

## REFERENCES


[1] B. Campbell, J. M. Goodman. HAM: A general purpose hypertext abstract machine, *The Journal of Communication of the ACM.* Vol. 31, No. 7, July 1988, pp. 856 - 861.

[2] P. Grag. Abstraction mehanisms in hypertext, in *Hypertext '87 Papers*, Chapel Hill, NC, November 13-15, 1987.

[3] *PDC Prolog User's Guide*, PDC, Copenhagen, 1990.

[4] *PDC Prolog Reference Guide*, PDC, Copenhagen, 1990.

[5] *PDC Prolog Toolbox*, PDC, Copenhagen, 1990.

[6] P. Seyer. *Understanding Hypertext: Concept and Applications*, WINDCREST, Blue Ridge Summit, 1991

[7] *Visual Prolog: Visual Development Environment*, PDC, Copenhagen, 1996.

[8] *Visual Prolog: Visual Programming Interface*, PDC, Copenhagen, 1996.


**Received:** 22 September 1998
**Accepted**: 2 February 1999

**Mirko Čubrilo**

## KONCEPTUALNI PROLOG STROJ ZA AUTOMATSKO PRESLIKAVANJE RJEČNIKA U STRUKTURU HIPERTEKSTA

### Sažetak

*Ovaj rad razmatra mogućnosti preslikavanja strukture klasičnih izvora informacija (rječnika,...) u strukturu hiperteksta. Struktura hiperteksta omogućuje učinkovitije korištenje preslikanih resursa (njihovo obogaćivanje novim, multimedijskim informacijama, povezivanje s bazama podataka i strukturiranje baza znanja,...). Ideja je u tome da se klasični izvori informacija interpretiraju kao formalni jezici i ona će biti ilustrirana na primjeru klasičnog rječnika, ali je jednako primjenjiva i na sve druge (slične) tipove klasičnih informacijskih resursa. Prikladnu tehničku osnovicu za implementacije nalazimo u jeziku logičkog programiranja Prolog. Metoda, pod nazivom Konceptualni Prolog stroj, oblikovana je i razvijena u tom okruženju.*

**Ključne riječi:** klasični izvori informacija, hipertekst, baza znanja, baza podataka, Prolog.