The PDB hypermedia package: why and how it was built

Citation for published version (APA):

Document status and date:
Published: 01/01/1991

Document Version:
Publisher’s PDF, also known as Version of Record (includes final page, issue and volume numbers)

Please check the document version of this publication:
• A submitted manuscript is the version of the article upon submission and before peer-review. There can be important differences between the submitted version and the official published version of record. People interested in the research are advised to contact the author for the final version of the publication, or visit the DOI to the publisher’s website.
• The final author version and the galley proof are versions of the publication after peer review.
• The final published version features the final layout of the paper including the volume, issue and page numbers.

Link to publication

General rights
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

• Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
• You may not further distribute the material or use it for any profit-making activity or commercial gain
• You may freely distribute the URL identifying the publication in the public portal.

If the publication is distributed under the terms of Article 25fa of the Dutch Copyright Act, indicated by the “Taverne” license above, please follow below link for the End User Agreement:
www.tue.nl/taverne

Take down policy
If you believe that this document breaches copyright please contact us at:
openaccess@tue.nl
providing details and we will investigate your claim.

Download date: 04. Sep. 2020
Eindhoven University of Technology
Department of Mathematics and Computing Science

The PDB Hypermedia Package.
Why and how it was built.

by

Pim Lemmens

Computing Science Note 91/14
Eindhoven, July 1991
This is a series of notes of the Computing Science Section of the Department of Mathematics and Computing Science Eindhoven University of Technology. Since many of these notes are preliminary versions or may be published elsewhere, they have a limited distribution only and are not for review. Copies of these notes are available from the author or the editor.
The PDB Hypermedia Package.

Why and how it was built

by Pim Lemmens.

Abstract:

This report describes the construction and use of a hypermedia package that has been developed at the computing science group of the Eindhoven University of Technology. The package has been created as an experiment in hypermedia software technology. Among its features are: dual representation of the structure of a hyperdocument, for reliability and speed, and support for the use of multiple formats for the contents of the hypermedia objects.

The report demonstrates how the design goals of ease of use, reliability, versatility and extensibility have been realised. It offers a formal specification of the packages' functionality and shows how it has been implemented using a functional database package previously developed at our institution.
1 PDB and hypermedia.

PDB is a hypertext package that has been developed in the Computer Science group at the Eindhoven University of Technology. It is written in Modula-2 to run on IBM-compatible PCs. It has been built on top of (part of) the ELDORADO functional database management software package [Lemmens, 1987], which later on was adapted specifically to its use in PDB. So in a way the PDB project is a continuation of the ELDORADO project of our group, which took place in 1986-1988.

PDB consists of a general purpose software library for making different types of hypertext systems on the one hand, and two self-contained programs constructed from this library, one for authoring (hyper)texts and one for consulting a hypertext, on the other. The library may be the basis for other kinds of hypertext programs that use the same underlying software and data structures.

1.1 Goals.

The goals of the PDB project have been:

1. Creation of a test bed for architectural and functional features of a hypertext system. As a test bed, PDB has been used to try out the use and implementation of named links and other novel or unusual features, as indicated in section 1.3.

2. Creation of a system for the support of authoring tasks. Its main use in practice has been as an authoring system, to produce (mostly) technical papers and reports (see ”Intended use”).

3. Creation of a system for on-line documentation. The third goal has been realised in the form of a program specifically made for consulting the hypertexts that have been produced using the authoring system.

1.2 Hypertext.

PDB is a hypertext program. What hypertext is and what a hypertext program should do is as yet not generally agreed upon by the people who work on it. What is clear is that hypertext is a form of non-linear data storage and presentation in which data items are connected by links that may be traversed from item to item in search of the item one needs [Conklin, 1987].

Hypertext systems are non-linear in the sense that their contents are not ordered along one dimension, as in a book. The items that make up the system are accessible by many different paths. Every user may follow a different route through the system, depending on his or her interests and knowledge. A hypertext may also be regarded as a network of concepts tied together by certain relations. Next page shows a diagram of an actual hypertext.

Most systems offer different ways to access the information contained in them. In addition to links that are marked on the display, they may offer keywords as a basis for a search through the complete hypertext, or show a map containing a selection of items and links from which a user may choose (‘browsers’). Some offer a kind of structured access, where a node in a net of items may in turn represent another net of items.

Modern systems are often called ‘hypermedia’ instead of hypertext. They may contain any kind of information. One item may be a text that is to be shown on a computer display screen, an other item may be a piece of sound to be produced by a loudspeaker system and yet another may be a video sequence.

3 Pim Lemmens
Hypertext example: PDB manual

For a linear rendering of this hypertext, see appendix A.

There are many hypertext systems and the options a user will have at any point will differ from one system to another.

Hypertext systems differ in:

Whether items should be discrete pieces (partitions) of information or just arbitrary points in a continuous field of data (e.g. a large text).
Whether links may be traversed in both directions or only in some 'forward' direction.
Whether links have names of their own or are only identified by the item that they are pointing at.
Whether links depart from specific points within a body of text (or an image, or any extended data field) or from the body as a whole.
What other means there are to find and access the information one needs (browsers, bookmarks, keywords, contexts, structured nodes).
Whether it contains only text, or text and graphics, or is truly multi-media.
Our system, PDB, offers discrete data items (objects), connected (in their entirety) by directed, named links that may be traversed in both directions, complemented by a global search mechanism using keywords. The nodes in the network, the objects, may contain any information and may use for presentation and/or modification any program that runs on the platform on which PDB is installed.

1.3 Intended use.

Regarding their purpose, there are three main groups of hypertext systems. There are encyclopedic systems ("structured browsing systems") to be used for consulting an extended body of information, like repair or user manuals of complex systems. There are authors' systems, for registering and structuring heaps of raw data and ideas to be used for the creation of an article, a book or a hypertext of the first category. And there are systems for collective use that serve as a medium for discussing issues that occur within a task area of e.g. a project group ("problem exploration systems").

The PDB program is primarily intended to be used as an authoring system. A different program, using the same software library, has been created for on-line (encyclopedic) documentation purposes.

These two forms of hypertexts have to meet different user requirements. An authoring system should be able to capture a lot of information fast and in such a way that it may easily be extended and modified. It should be very flexible, allowing the user to enter and organise the data in a way that corresponds most closely to his or her personal view.

A documentation system should produce precisely the information the user needs as fast as possible. In addition to being fast, a documentation system should be easy to use, requiring a minimum of actions to produce a required result and a minimum of previous knowledge (of some command language or the structure of the system) to handle it.

1.3.1 Authoring system.

The process of creation (of a book, a design, etc.) consists of a number of subprocesses that should be performed in a certain order to arrive at a complete result. These subprocesses (phases) are:

- data collection
- incubation
- inspiration
- consolidation
- elaboration
- preparation

It is evident that one needs sufficient data to support the result that should be produced. PDB may be used for capturing these data while they are collected. The collected data may also be the basis for the inspiration necessary to arrive at a good plot or outline design or whatever the human touch in creation is needed for. In order to get this inspiration it is necessary to have a certain incubation period, in which the creator "plays around" to familiarise himself with the data. PDB also supports this stage.

Once ideas start to emerge they must be captured fast and efficient (consolidation) because of
their often rather volatile character. PDB does not require concepts to be fully worked out in order to be registered. Later on they should be evaluated, possibly modified, and documented (elaboration) with the support of PDB to organise the data and with one or more editors to process them. And finally, the finished product should be prepared on the basis of all that has been registered before. This may be done with a desk top publisher to which PDB may deliver the raw material.

1.3.2 Hypertext documentation systems.

A (multimedia) hypertext system can be very useful for documentation purposes, like service documentation of complex technical systems and for systems used for explanation of certain subjects to the general public, as in public exhibitions. These systems will not necessarily need a browser that shows a map of the objects and their relations, but it may be very useful if objects may be accessed by several different routes and using several different mechanisms.

The mode of access should agree as closely as possible with the user's view of the structure of the information to be handled. A system meant for creating and managing such systems should thus have the flexibility to accommodate possible different views. As these systems address many different kinds of users they should also be simple to use, offer extensive help and allow the user to easily undo any unintended action he/she has performed.

Hypertext data may evidently be organised in several different ways, using different types of relations, stemming from different paradigms. There is the notice-board paradigm, a way to pin down information without really organising it. Then there is the semantic net paradigm that allows organisation of data according to its meaning, with e.g. 'is-a' and 'part-of' relations. There is the book paradigm that is structure-oriented, arranging the data in chapters, sections and paragraphs, with a preface, a table of contents, an index, etc. [Oman, 1990]. And there is the encyclopedia paradigm, with references and cross-references. Yet another paradigm is the discussion style organisation used by the IBIS system [Begeman, 1988] to cooperatively solve so-called "wicked problems". As a system meant to help authors at their jobs, PDB supports several different paradigms, by providing named links and keywords, as shown below.

1.3.3 Use of links.

Links are used to structure a hypertext by connecting related objects. Such a structure may be hierarchical or non-hierarchical. A hierarchical structure is established by e.g. a subdivision of the system into chapters, sections and subsections. There may however be cross-links also, linking objects from different branches of the hierarchy tree. Links may be used to navigate through the system, moving from object to object by way of the links that connect them.

In contrast to the custom of many other hypertext systems, PDB links have names that are different from those of the objects they connect. These names may indicate the relation of the connected objects, like 'annotation', 'continuation', 'has as a part', or simply 'association'. In a description of a certain geography the objects may be connected by 'east', 'west', 'north' and 'south' links, indicating their relative positions. The names may however also be used as possible answers to questions posed by an object, an object 'question' may be linked to two objects, 'wrong answer' and 'right answer' by links named respectively 'yes' and 'no'.
1.3.4 Use of keywords.

Keywords may be used for a number of purposes:

1. For a rough description of the contents of an object.
2. To indicate the category or categories to which an object belongs.
3. As arguments for a global search through the complete hypertext.

During the creation of a hypertext one may want to establish a number of objects fast, defining a global structure, without adding contents to the objects rightaway. Then keywords may be used to indicate the intended contents of the nodes for later reference. Therefore, in the authoring system, the keywords are always displayed on the description window of an object.

Normally, a hypertext will have a hierarchical structure with additional cross links. The hierarchy will define a number of sections, consisting of objects with related contents within the system, but it may be useful to have a different classification as well. Some subjects may appear in different branches of the hierarchy. We may, for example have a division into requirements, design and implementation of a certain project, where the specific subprojects appear in each of these chapters. In that case an additional subdivision may be made, using keywords to indicate the subprojects.

In most cases a hypertext will be traversed using links to move from one object to another. At a given instant it may however be necessary to move to a totally different area of the system. The required object may then be found through a global keyword search.

1.4 PDB Requirements.

The primary requirements for the PDB system have been that it should provide:

1 Ease of use by means of well-chosen features (see next section), a well-designed and well-tested user interface and a reasonable response time.

2 Reliability through redundancy in the storage of data and the provision of a restoration facility for damaged data.

3 Versatility by offering several ways to move across a hypertext and by having an open structure that allows the use of external programs for the display and modification of data. Also, PDB should support any type of directed graph structure.

4 Extensibility through modular architecture of the software and again the open structure of the system, which allows "on the fly" extension of object types.

1.5 PDB features.

A PDB hypertext is built from a number of entities:

Objects of different types, containing the pieces of information the user may need.

Links between objects as a means of getting from one to another.

Keywords, to be attached to the objects, for a short characterisation of their contents.
The PDB package

The authoring and consulting programs for using these entities are controlled by one-letter commands and several kinds of menu. Both programs present different kinds of windows. There are summary windows that show all kinds of data on the current object, like its name, keywords, its forward links and its backward links. Of course there are windows that display the contents of an object. There are also the menu windows that offer a choice of link names, object names or keywords. Some will even allow modification of the selection offered.

The PDB system offers a number of features, many of which are not generally found in existing hypertext systems:

1. Multiple formats of object values (contents types). The contents of an object may be of any type. For the PDB system they are just a file of bytes, without any implied meaning. The system will however recognise the type of contents and accordingly invoke a processing program when needed. Whether this is a viewer for presenting the contents to the user, or an editor for modification, will depend on the wishes and the authorisation of the user.

2. Keyword search. Once in a while, a user may want to switch from one environment in a hypertext to a totally different one. To get from the one to the other using links may be very time-consuming. In this case a global search, using keywords as search arguments, may be performed.

3. Link names (typed links). Most hypertext systems use unnamed links, which are identified from a given object by their target object. In PDB, each link will have a name, so there may be more than one link between a pair of objects, each of a different link name.

4. Insensitivity to spelling errors in keywords and names. PDB uses a modified soundex coding algorithm [Knuth, 1973] that allows retrieval of names that are related in spelling to a given string. So any misspellings may be corrected by choosing a name from the alternatives presented by the system.

5. 'Set editor' user interface. Menus may be used for more than just choosing one option from a limited number presented to the user. In some situations it may be necessary to choose more than one entry, or it may be useful to add elements to the menu. In PDB, all this is possible. In principle, menus are regarded as sets of options that may be edited at will. Dependent upon the situation (e.g. retrieval of objects or update of keyword lists) different restrictions may be posed to the freedom of the user to do so.

6. Hypertext update and definition language. Hypertexts are mostly created interactively. In some situations, such as when converting a 'flat' text to a hypertext, it may however be useful to have at one's disposal a language that allows insertion of commands to produce a hypertext from a non-hypertext (linear) original.

7. Restoration facilities. If a hypertext is fully or partly destroyed, e.g. because of a disk crash, it is possible to restore it from a backup file. Backup files are generated automatically, each time garbage collection is started. Even when no backup files are available, the hypertext may be reconstructed automatically from the 'objects' database file, partly or fully, dependent upon the part of the file that is still intact.

8. Generation of 'flat' files. In an authoring system, the ultimate goal may be to produce a linear rendering of the information available. PDB supports this process by appending the contents of selected objects to a user specified file.
1.6 Subsystems.

The PDB system is composed of a number of subsystems:

1. The management system for the contents of objects.
2. The objects description system with the extensions file.
3. The index database.
4. The user interaction.

As mentioned before, PDB does not itself ascribe any meaning to the contents of an object, nor does it impose any predefined format. Contents are regarded as just a string of bytes, the meaning of which depends on external programs that use it. The contents of each object are characterised by a specific kind name. To each kind name two processor names are specified: an editor name and a viewer name. Viewers are only meant for presentation of the contents, while editors may modify it. If the user demands a presentation of the contents, the viewer will be invoked. The editor is started when the user requests a modification of the contents. The link between kind names on one side and viewer or editor names on the other is made on the basis of the configuration tables, which are stored with the other data and which may be modified interactively. More on the contents subsystem in chapter 5.

A list of contents processors, the configuration table, together with a full description of the hypertext is stored in the extensions file 

( "OBJECTS.FDB" ), which may be interpreted by the system, or by any human interpreter, as they are described in a humanly readable language. PDB contains an interpreter for this language that is used for the presentation of the data during an interactive session. A further description may be found in chapter 3.

For fast access, an index into the extensions file is created, a database that holds the relationships between objects, keywords and links. During an interactive session the extensions file is accessed by way of this database. Interactive modifications to the hypertext or the configuration subsystem will update this database as well as the extensions file. The database may however also be created from the extensions file alone, using the hypertext language interpreter. See chapter 4.

User interaction is provided by the interaction subsystem. This system controls all user I/O. It provides generalised menus and a line editor that is invoked for all input of (link, keyword, object or file) names. The interaction is treated in chapter 6.

Each subsystem has a part in the functionality of the total system. But the functionality of the extensions file system and the index system overlaps: a number of operations may be realised both by the former and by the latter. The PDB program now uses of both these subsystems only those operations that execute faster than those of the other subsystem, thus offering optimum performance.

For a graphical representation of the structure of PDB, see below.
2 Formal model of PDB hypertext.

2.0 Abstract Specifications.

The formal specifications in this paper all consist of three parts: a structure description, the axioms (invariants) for the variables described in this structure, and the operations that may be performed on these variables, defined in terms of pre- and postconditions. The variables and operations used do not necessarily have any relationship to those used for the actual implementation. They are purely mathematical constructs, to be used for reasoning about the functional aspects of the system, the way it behaves to the outside world.

The structure description sums up the types of variables used in the process of reasoning about the functionality of the system. It defines their structure, mainly in terms of (mathematical) sets, cartesian products (vectors) and functions (N:1 mappings).

The axioms (or invariants) define the values these variables may have, in the form of predicates. So the axioms add some semantical aspects. They establish interdependencies and constraints...
The PDB package

on values of components of structures. They are valid initially and after whatever change the
system in operation is subjected to.

The operations describe the ways in which the variables may be changed. They are defined by
preconditions, conditions that must be true in order to perform the operation, and postconditions
that define the values of the relevant variables after the operation. The axioms may thereby be
viewed as additional postconditions for all operations. Taken together, the operations describe
another set of invariants, in addition to the axioms, as changes of values, other than by the
specified operations, are not allowed.

2.1 PDB structure.

PDB hypertexts may be described as a triple:

\[ H = <O, R, K> \]

where

- \( O \subset S \ast S \ast S \) is the set of all objects in the hypertext and each object from the set
  may be described by a 3-tuple \(<N, T, C>\), of name, type and contents.
- \( R \subset S \ast S \ast S \) is a relation with elements \(<N, S, T>\) (Name, Source, Target) indicating named connections (links) between objects.
- \( K \subset S \ast S \) is the keywords-relation that associates a set of keywords with each object.
  \( k = <N, R> \) for \( k \in K \), with \( N \) = the keyword name, and \( R \) the name of the object to
  which it refers.

\( S \) indicates the set of all possible character strings.

2.2 Axioms.

Definitions:

- \( ON = \{o.N \mid o \in O\} \) are the object names.
- \( RN = \{r.N \mid r \in R\} \) are the relation names.
- \( KN = \{k.N \mid k \in K\} \) are the keywords.

Now the following axioms apply:

\( \forall o1, o2 \in O: o1.N = o2.N \Rightarrow o1 = o2 \)

All objects have unique names.

'Start' \( \in ON \)

One object, named 'Start', may never be removed.

\( ON \cap RN = \phi \)
The PDB package

No object has a name that is also a relation name.

\[ \forall r \in R: r.S \in \text{ON} \text{ AND } r.T \in \text{ON} \]

Sources and targets of links are names of existing objects.

\[ \forall k \in K: k.R \in \text{ON} \]

Keywords refer only to existing objects; there are no keywords in the K set that have no referents.

2.3 Basic operations.

The operations to be performed on a hypertext H may be divided into two classes: query operations and update operations. Query operations do not change the hypertext. They only search for, and present, specific data. Update operations will persistently change the hypertext.

2.3.1 Query operations.

**Keywords(x)** = \{k.N \mid k \in K \land k.R = x\}

Find the keywords associated with the object named x.

**KeywFind(W)** = \{x \in \text{ON} \mid W \subseteq \text{Keywords(x)}\}

Find all objects that have the elements of W as keywords.

**Forward(x)** = \{r.N \mid r \in R \land r.S = x\}

Find the links from the object with name x.

**Backward(x)** = \{r.N \mid r \in R \land r.T = x\}

Find the 'backward' links, i.e. the links to the x object.

**Targets(L, x)** = \{r.T \mid r \in R \land r.N = L \land r.S = x\}

Find the targets of link L from object x.

**Sources(L, x)** = \{r.S \mid r \in R \land r.N = L \land r.T = x\}

Find the source objects of the links named L that have x as a target.

**Contents(x)** = \{o.C \mid o \in O \land o.N = x\}

Supplies the contents of x.

2.3.2 Update operations.

The following operations change the hypertext. They are specified by their effect (postcondition) upon H (Note that although they are not explicitly mentioned as pre- or postcondition the axioms always apply. Preconditions are only shown if they are different from TRUE):
The PDB package

NewObj(x):
   Postcondition: x ∈ ON
   Create a new object with name x.

PutContents(x, V, Kind):
   Postcondition: V = o.C ∧ Kind = o.T ∧ o.N = x
   Assign contents V of type Kind to the x object.

PutKeywords(x, W):
   Postcondition: W = {k.N | k ∈ K ∧ k.R = x}
   Assign a given set of keywords to the x object.

PutLinks(x, Y, L):
   Postcondition: Y = {r.T | r ∈ R ∧ r.N = L ∧ r.S = x}
   Specify a number of links, collectively named L, as forward links from the x object to the objects in Y.

Forward links may be removed by:
   PutLinks(x, φ, L)

DisposeObj(x):
   Precondition: H = <O', R', K'>
   Postcondition: (x ≠ 'Start' ∧ Forward(x) = ) ⇒ ¬ (x ∈ ON)
                   ∧ (x = 'Start' ∨ Forward(x) ≠ ) ⇒ H = <O', R', K'>
   Remove the x object from the hypertext, if x does not have any forward links. Because of the relevant axiom, object 'Start' may never be removed.

2.4 PDB operations.

The PDB program adds to these definitions the notion of 'actual object'. The program at each moment is in some state

   S = <H, A>

where

H is a hypertext and
A ∈ S is the name of the actual object, the object that is currently being processed.

The program has the additional axiom

   A ∈ ON

   The current object should be one of the available objects.
And initially, \( A = 'Start' \), each session starts at the start object.

Special operations:

\[
\text{Pick}(S) \in S, \text{ where } S \text{ is any set of names: pick one element from the set.}
\]

\[
\text{GoTo}(\text{Obj}) \text{ has the effect: } \text{Obj} \in \text{ON} \iff A = \text{Obj}.
\]

Provided the object is present, make it the actual object.

PDB offers a number of query operations that may be regarded as being composed of the basic query operations defined above: moving from the actual object to another (making it actual) by way of a (forward) link may be described by:

\[
\text{GoTo}( \text{Pick}( \text{Targets}( \text{Pick}( \text{Forward}(A) ), A )) )
\]

Going backwards over a link is analogous:

\[
\text{GoTo}( \text{Pick}( \text{Sources}( \text{Pick}( \text{Backward}(A) ), A )))
\]

Global (keyword) search may be defined as:

\[
\text{GoTo}( \text{Pick}( \text{KeywFind}(X) ) )
\]

The actual object will be one of the objects that has all elements of \( X \) as keywords. This operation searches the complete hypertext, not only the vicinity of the actual object.
The PDB package

3 Hypertext definition.

As mentioned before, the representation of the structure of a hypertext is duplicated within the PDB system. The extensions file 'OBJECTS.FDB' contains a full description in an interpreter language, while the index database that resides in file 'REFERENCES.FDB' contains a representation in the form of records with pointers for fast access by the computer.

The definition of a PDB hypertext, as recorded in the extensions file, consists of two parts: a configuration part and an object definition part. The configuration part describes the types of contents the hypertext objects may have:

\[
\text{Configuration} \subseteq S \times S \times S
\]

The configuration part contains a table with entries <Kind, Viewer, Editor>. 'Kind', the kind name, is the key to the entry, 'Viewer' is the name of a program for presenting the contents of objects that have the relevant kind, and 'Editor' is the name of the modification program for this kind of objects. 'Viewer' and 'Editor' should be accessible to the file system. If necessary they should be complete path names.

More on configurations in chapter 5.

The object definition part consists of a set of object definitions:

\[
\text{Object definitions} \subseteq S \times P(S) \times P(S \times P(S)) \times [S \times S]
\]

where each object is characterised by a tuple <Name, Keywords, Links, Contents>. Each object is identified by its name (key attribute) and this is the only required component of the object definition.

'Keywords' is the set of keywords of the object,

'Links' defines the links from the object, each link being characterised by a tuple <Name, Targets> and

'Contents' represents the contents, characterised by <Kind, Value>. 'Kind' is the name of the kind of contents and 'Value', a string of bytes of arbitrary length, is its contents.

These definitions are expressed in a specially developed language, the PDB Definition and Update Language.

3.1 Two representations.

We now have two ways to describe the structure of a hypertext, one that gives a global view and one that considers the system as a collection of objects, each with its own specific data:

\[
\text{Hypertext } H = \langle O, R, K \rangle
\]

and:

\[
\text{Object definitions} = \text{Set of } \langle \text{Name}, \text{Keywords}, \text{Links}, \text{Contents} \rangle.
\]

As they both describe the same system, they should be closely related. The description in one
The PDB package

format should be definable in terms of the other. In fact, the axioms of the objects definitions system may be defined in terms of \( H \):

\[
\{ o.\text{Name} \mid o \in \text{Object definitions} \} = \{ o.N \mid o \in H.O \}
\]

The 'Name' attribute of an object description identifies an element of the set of objects of \( H \).

For a given object \( o \in \text{Object definitions} \):

\[
o.\text{Keywords} = \{ k.\text{Name} \mid k \in H.K \land k.R = o.\text{Name} \}
\]

The 'Keywords' attribute corresponds to the subset of the keywords of \( H \) that have the object as a referent.

\[
o.\text{Contents.Kind} = ( p.T \mid p \in H.O \land p.N = o.\text{Name} )
\]

\[
o.\text{Contents.Value} = ( p.C \mid p \in H.O \land p.N = o.\text{Name} )
\]

The kind identification and the value in the 'Contents' field correspond to the type and contents attributes of the object in \( H \).

Further:

\[
\{ l.\text{Name} \mid l \in o.\text{Links} \} = \{ r.N \mid r \in H.R \land r.S = o.\text{Name} \}
\]

and for every \( l \in o.\text{Links} \):

\[
l.\text{Targets} = \{ r.T \mid r \in H.R \land r.N = l.\text{Name} \land r.S = o.\text{Name} \}
\]

Each link \( 'l' \) of \( 'o' \) corresponds to a subrelation of \( H.R \), with \( 'o.l.\text{Name}' \) as a name and \( 'o.\text{Name}' \) as a source.

3.2 Interpreter language.

In essence, a hypertext definition in the PDB language consists of strings grouped together to sets or tuples or sets of tuples or tuples of sets, etc., thus allowing any data structure to be specified. A tuple will be of the form

\[
\text{string1} \{ \text{string2} | \text{string3} \} \text{ string4} | \text{string5} \ldots
\]

where 'string1' is the name of the tuple, 'string2' and 'string4' are the names of components, and 'string3' and 'string5' are the values of the components, which in turn may have an internal structure (e.g. 'string3' may be a tuple specification). The order in which the components appear is immaterial. A set will be specified as

\[
\text{string1} \{ \text{string2} | \text{string3} \ldots \}
\]

where 'string1' is the name of the set and the elements 'string2' and 'string3' may again be structured:

\[
e.g. \text{for string2:} \{ \text{string4} | \text{string5} | \ldots \}
\]
Each structured element may be terminated by a string, which should be equal to the string by which it starts:

```
string1 { string2 | .. } string1
```

Keywords of the hypertext syntax are: 'Comment', 'Config', 'Viewer', 'Editor', 'Object', 'Name', 'Keywords', 'Links', 'Targets', 'Contents', 'Kind', 'Value'. These are not reserved words. They may be used freely outside the context in which the interpreter expects them (e.g. as keywords, link or object names or within contents).

Comments may occur anywhere between the 'Config' and the 'Object' definitions. They may be used to bracket out strings that should not occur in the hypertext.

A full specification of a part of a hypertext looks like:

```
Comment { This part will not be incorporated in the hypertext }
Config { Kind {Text} | Viewer {own} | Editor {see} }
Object { Name{Start} |
  Keywords {start | begin | root } |
  Links { Name {document} | Targets {Hypertext} }
  {Targets {Links | Keywords | Contents} | Name {help} } |
  Contents { Kind {Text} | Value {This is the actual contents of the object.} }
  } Contents
} Object 
```

For a definition of the syntax, see section 3.2.2.

3.2.1 Update language.

The definition and update language allows us to specify the complete structure of a hypertext plus the configuration for the processing of object contents. It also allows the specification of modifications to existing objects, by using as default data those of the existing hypertext, so only explicitly specified data are modified.

Some examples:

```
Object{ Name{Abc} | Keywords{} } 
```

means: "Remove all keywords from object 'Abc'."

```
Object{ Name{Abc} | Links{ Name{pqr} | Targets{Xyz} } |Links }
```

"Add link 'pqr' to object 'Abc', with target 'Xyz'."

If the object already owned the link, 'pqr'‘s target set is changed to {Xyz}.

```
Object{ Name{Abc} | Links{ Name{pqr} | Targets{}} } 
```

"Remove link 'pqr' from object 'Abc'."

Only those attributes (keyword set, one or more links, contents) of the object that are explicitly
mentioned are affected. The others will remain unchanged.

3.2.2 Formal syntax specification.

The interpreter is based on a syntax that facilitates partial interpretation, such that different program parts may interpret different parts of the specification. So there may be several interpreters, each with its own goal, such as a configurations interpreter and an objects interpreter. Interpreters for the contents of an object may also be useful. The only part of the syntax that is common to all interpreters is the interpunction: the symbols '{' and '}' mark, respectively, the beginning and the end of a (possibly nested) block of specifications, and ':' marks the separation of two parts. The general syntax is:

```
construct ::= <empty> | <string> | <structure>
structure ::= <simple set> | <tuple> | <set of struct>
simple set ::= <string> | <simple set> ':' <string>
tuple ::= <named element> | <tuple> ':' <named element>
set of struct ::= '{' <structure> '}' | <set of struct> ':' '{' <structure> '}'
named element ::= <string> '{' <construct> '}' [ <string> ]
string ::= <character> | <string> <character>
character ::= <non-meta letter> | \ <letter>
letter ∈ ASCII-set
non-meta letter ∈ (ASCII-set \ {',', ':', '}'} \ \})
```

where construct "[x]" indicates 0 or 1 times x, "A \ B" indicates set difference of A and B and "{x, y, z}" indicates the set, consisting of x, y, and z.

Leading spaces of strings are discarded, so wherever they are essential, they should be escaped (preceded by one \-symbol). The optional string at the end of a named element should be identical to the one at the front.

Formally a hypertext definition is a tuple, according to the definition above, of 'Config', 'Comment' and 'Object' elements. In contrast to what is normally known as a tuple, each of these element types may occur arbitrarily often in the definition.

The 'Config' elements are sets of 3-tuples with elements 'Kind', 'Viewer' and 'Editor', each of one string construct.

The <construct> part of 'Comment' elements consists of one (possibly long) string.

The 'Object' element, of which there may be very many, must have a 'Name' element and may have 'Keywords', 'Links' and 'Contents' elements. 'Keywords' contents is a set of (name) strings. The 'Links' element contains a set of ('Name', 'Targets') tuples, where 'Targets' contains a (possibly empty) set of strings. And the 'Contents' element is composed of a 'Kind' part and a 'Value' part.

3.3 Use of the extensions file.

The PDB interpreter language serves a dual purpose: it allows flat files, containing a description of (part of) a hypertext to be incorporated in an actual hypertext and it is the medium in which all data of an object are recorded inside the hypertext. As such, it is accessed through the index database that finds the part of the extensions file ('OBJECTS.FOB') where the data on the actual object is stored and presents it to the interpreter that converts it to a data structure, the
current object buffer, that contains all data on the current object. All operations on the current object are performed on that structure. When the operations have changed the data, the index database is updated and the structure is converted back into the PDB language and added to the extensions file.

The same interpreter that reads the extensions file data is used for the conversion of external descriptions into parts of the current hypertext. The procedure is as follows: when an object of the name mentioned in the external description already exists, it is loaded into the data structure mentioned above. Next the description is interpreted as an update of that data (which are empty when the object was not yet present). And finally the data are converted to part of the extensions file, adding them to the index database as well.

3.4 Extensions file operations.

The PDB operations may be divided into two groups, dependent upon the data they use. Some of the operations use the definitions stored in the current object buffer and some use the index database (see next chapter). The operations that are based on the buffer for the current object are:

Keywords(A)
Forward(A)
Targets(L, A)
Contents(A)

and also the part

Pick( Targets( Pick( Forward(A), A ) )

of the Goto operation to an other object using forward links.
4 Index subsystem.

Objects may be accessed by way of links from one object to another, through keywords associated with them, or directly by name. The keywords, link names and object names and their connections are stored in a functional database that provides an index into the extensions file where the contents of objects, together with the other data of each object is stored. The database allows fast access to objects and the extensions file provides the data wanted of each object.

4.1 Data model.

The index of a hypertext is built from the following components:

Sets:

\[ \text{KwCodes, NmCodes} \subseteq \mathbb{N} \]
\[ \text{KeywordNames, ObjectNames, LinkNames} \subseteq S \]

Functions:

\[ \text{KwNameCode: KeywordNames} \rightarrow \text{KwCodes} \]
\[ \text{ObjNameCode: ObjectNames} \rightarrow \text{NmCodes} \]
\[ \text{LkNameCode: LinkNames} \rightarrow \text{NmCodes} \]

Relations:

\[ \text{Keywords} \subseteq \text{KeywordNames} \times \text{ObjectNames} \]
\[ \text{Links} \subseteq \text{LinkNames} \times \text{ObjectNames} \times \text{ObjectNames} \]

In addition to these components, there is an implicit function, supplied by the database system, that links the object name entries to the object definitions in a different file:

\[ \text{ObjNameDefinition: ObjectNames} \rightarrow \text{Objects} \]

'KeywordNames', 'ObjectName' and 'LinkNames' correspond to resp. 'KN', 'ON' and 'RN' from the specification in chapter 2. 'Keywords' and 'Links' correspond to 'H.K' and 'H.R' respectively. 'KwCodes' and 'NmCodes' are codes for keyword names and object and link names, respectively. They are used as hash codes for retrieval of the corresponding names from the database.

4.2 Soundex coding.

PDB maintains two sets of names: keywords and the set of object and link names. In order to find the relevant element of such a set, given the name of the item as a string of characters, the system uses hash tables, where the hashing function is based on a modified soundex algorithm. Thus two names that sound alike are mapped to the same hash code. So, when a user makes a spelling error, the system can retrieve a list of soundalike names, and ask him or her whether the name (s)he needs is among them. The algorithm works as follows:

1. Find the first character that is a letter (A..Z or a..z).
2. Convert each letter to lower case.
3. First digit of hash code:
Keyword codes

Name codes

Keyword names

Object names

Link names

Data Model of PDB Index

'a', 'e', 'i', 'o', 'u', 'y', 'h', 'j', 'w': 0
'b', 'p', 'f', 'v': 1
'c', 'g', 'k', 'q', 's', 'x', 'z': 2
'd', 't': 3
'l': 4
'm', 'n': 5
'r': 6

5.a Code of letter equal to previous code: skip it.
   b Code is 0: skip it.
   c Else add digit to code.
6. If not end of string and code is less than four digits: go to 4.
7. Add zeroes until code is four digits long.

So both 'Eldorado' and 'alter a door' will be coded as '0436' and typing 'alter a door' as a
The PDB package keyword will, when it is not present in the keyword list, yield 'Eldorado', if that is present, as an alternative.

With respect to the original Soundex algorithm [Knuth, 1973] a few letters have been moved to other codes to accommodate Dutch spelling habits. The difference between words starting with a vowel and those that don't is reflected in their code. Other vowels are suppressed, as in the original algorithm.

4.2 The ELDORADO system.

As mentioned in the introduction, PDB is built using the ELDORADO database system. ELDORADO is used to implement functional databases, databases that are built from sets of items linked by functions. Items are primitive entities like numbers or names. Each item may however have an extension, which may be any collection of data that is relevant to the user. The ELDORADO system itself does not use this extension, it only operates on the sets, called categories, and the functions.

ELDORADO offers a number of query and update operations, both for single items and for sets and functions. For instance, it is possible to obtain in one operation the set of all domain elements corresponding to a single given range element for a given function. Furthermore, it has set operators that determine the union of a pair of sets, their intersection or their difference.

The references file in which the database resides holds the structure of the hypertext in machine readable form. That means that references to items, names or codes of objects, links or keywords, are stored in the records of the database in the form of 32-bit identification numbers.

In addition to the extensions file and the references file there is a third file, the locations file, that is a simple table that contains the identification numbers of items, together with their addresses in the reference and extensions files. Due to this file, the frequent modifications during the construction of a hypertext do not lead to extensive updating of pointers all through the system. Change of location of components is only reflected in this file. In the future this file may also serve to implement a network version of PDB, the entries being among others descriptions of remote locations.

4.3 Use of ELDORADO.

The sets mentioned above are implemented as categories in ELDORADO, the functions are ELDORADO functions and the relations are implemented using "ghost categories", sets of elements without a value that serve only as a domain or range for functions. 'Keywords' is such a category, with two functions, one ranging over 'KeywordNames' and one with range 'ObjectNames'. 'Links' has the functions 'From' and 'To', that range over 'ObjectNames', complemented by a function with range 'LinkNames'.

Whenever a given name is to be found in the database, it is first converted into Soundex code. Then the code is located in the database, either in the set 'KwCodes' or in 'NmCodes'. Next the relevant function, either 'KwNameCode' or 'ObjNameCode' or 'LkNameCode' is applied inversely, generating the set of all names with the same code. Finally, all names from this set are compared with the given name, until the name, and thus the location in the database, is found.

The operation: "Find the objects that correspond to a given set of keywords" (KeywFind(X) in
chapter 2) is implemented in ELDORADO in the following way:

Find the first of the given keyword names in the database and apply the inverse of the function (one of the functions that implements the 'Keywords' relation, the one that maps 'Keywords' onto 'KeywordNames'), giving the elements from the relation 'Keywords' that correspond to that name. Apply the function that maps this set to the object names, thus finding the set of objects that have the given name as one of their keywords. Do the same with the second keyword name and intersect the resulting set with the original one. Repeat this for the remaining keyword names. This results in the subset of 'ObjectNames' that corresponds to all objects that have the given set of keyword names as keywords.

4.4 Reference file operations.

The PDB operations that are implemented using the reference file (the PDB index) are KeywFind(X), as described above, and all Goto parts of operations, the parts that find the actual location of an object in the database, given its name. The index subsystem also implements the Goto operation that moves backward over a link:

\[
\text{GoTo( Pick( Sources( Pick( Backward(A)), A )))}
\]
5 The contents subsystem.

Each node in a PDB hypertext may contain some kind of free-format data, like an ASCII or WordPerfect text, a graphics file or program code. To indicate what type of contents the object has, a kind name is associated with the data - no object data without a kind identifier. The PDB program maintains a kinds table, where each kind name is associated with a viewer and an editor that respectively may interpret or modify the data. Viewers may display the data on the screen (for text or picture data) or execute the program code that is contained in the data, or do anything that should be done to activate the contents of the object. Editors may create new data or change existing object data: word processors, paint programs, compilers, etc.

So PDB is truly multi-media in that it may contain any data that may be accessed and used by a computer or its peripherals.

5.1 The configuration table.

The kind of the contents of an object as described in chapter 3, o.Contents.Kind, must occur in a configuration table with entries <Kind, Viewer, Editor> that links viewers and editors to kind identifiers:

∀ o ∈ Object definitions: o.Contents.Kind = empty

The configuration table has the kind names as a key:

∀ c1, c2 ∈ Configuration: c1.Kind = c2.Kind ⇒ c1 = c2

Now each 'Viewer' and each 'Editor' name from the configuration table indicate a program within the system that is accessible to PDB (the processor). These programs may be described as functions:

∀ c ∈ Configuration: Processor(c.Viewer): S → S
∧ Processor(c.Editor): S * S → S,

where S is the set of strings of any length.

The configuration table of a hypertext will name all viewers and all editors that are to be used by that hypertext. In practice, the names indicate the paths where the relevant programs are to be found. When the user selects the option 'modify' when accessing the value data of the current hypertext object, the editor belonging to these data will be started, allowing interactive modification of the current contents.

5.2 Contents handling operations.

The operations of the contents subsystem may be specified as follows:

Display contents of object x:

Attach contents kind 'k' to object x:

PutKind(k, x):

    Precondition: E c ∈ Configuration: c.Kind = k
    Postcondition: x.Contents.Kind = k

Edit contents of object x (Q is a sequence of edit commands):

Modify(Q, x):

    Postcondition: c.Contents.Value = 
                    Processor(c.Editor | c ∈ Configuration ∧ x.Contents.Kind = c.Kind)(Q, C)

Add new entry (kind 'k', viewer 'v', editor 'e') to configuration table:

PutConfig(k, v, e):

    Postcondition: <k, v, e> ∈ Configuration
6 User interaction.

There are several ways in which a user may interact with the PDB system. First, he or she may give one-letter commands (from a set of alternatives displayed on the screen). Next, (s)he may choose an item from a menu by stepping to it and pushing a 'select' button. And finally, he or she may need to type names of files or objects or links or keywords.

The first mode is the fastest, requiring only one key press. This mode is used for most commands, to indicate what's to be done. Selection of one or more items from a menu may indicate objects to go to, links to follow or keywords to use in a keyword search. These actions often also allow addition of items to a menu. Finally, new names or names unknown within the current context should be entered character by character, using a special line editor.

6.1 Menus.

Menus are used, among other things, to implement the 'Pick' operation mentioned in section 2.4: select one element from a set of alternatives given in the menu. In PDB, however, menus also have a broader use. In some menus the user may select more than one element from the list of menu items. There are even cases in which one may add and remove elements of the list. The PDB menu operation is:

\[
\text{DoMenu} : P(S) \times \text{Inputs} \times P(\text{Attr}) \rightarrow P(S)
\]

where:

\[
\text{Inputs} = (S \cup \text{KeyComs})^*
\]

The actions of the user, any sequence of strings and special keystrokes.

\[
\text{KeyComs} = \{\uparrow, \downarrow, \rightarrow, \leftarrow, \text{Home}, \text{End}, \text{PageUp}, \text{PageDown}, \text{Insert}, \text{Delete}, \text{BackSpace}, \text{Esc}, \text{Enter}\}
\]

Special keys for control of menu operations.

\[
\text{Attr} = \{\text{Vertical}, \text{Preselect}, \text{Single}, \text{Add}\}
\]

Attributes used to specify the mode of operation.

DoMenu displays a set of entries (strings) horizontally or vertically (attribute 'Vertical') on the screen, allowing the user to interactively select (keyboard command 'Insert'), deselect (command 'Delete') or add entries (by typing in the relevant string; these entries are selected by default). After the menu is closed the operation delivers the set of selected entries (if closed by 'Enter') or the original input set (if closed by 'Esc').

Axioms:

if \[
M = \text{DoMenu}(M', \text{KeyBln}, A)
\]

then:

\[
\neg(|\text{Preselect}, \text{Single}) \subseteq A)
\]

The attributes 'Preselect' and 'Single' should not occur together.
The PDB package

Single ∈ A ⇒ #M = 1

The 'Single' attribute means that the result set has exactly one element.

¬(Add ∈ A) ⇒ M ⊆ M'

Without the 'Add' attribute no elements may be added to the original set.

Strings are added using the line editor.

6.2 Menu operations.

DoMenu internally uses the following data structure:

MenuState = <M, S, Number, Total, Current>

where:

M is the set of entries, M ⊆ S;
S, the selection, is a mapping of M to {TRUE, FALSE};
S: M → {TRUE, FALSE}
Total ∈ N is the number of elements of the menu.
Number: M → [0 .. Total-1] assigns unique consecutive numbers to the elements of M
Current ∈ S

State axioms:

Number(x) = Number(y) ⇒ x = y
#M = Total
Current ∈ M

Definitions:

Succ(x) =
\[ \begin{cases} 
  x & \text{if Number(x) = Total - 1} \\
  \text{Number}'(Number(x) + 1) & \text{otherwise}
\end{cases} \]

Pred(x) =
\[ \begin{cases} 
  x & \text{if Number(x) = 0} \\
  \text{Number}'(Number(x) - 1) & \text{otherwise}
\end{cases} \]

Initially, after starting DoMenu(M', KeybIn, A):

MenuState.M = M'
∀ i ∈ MenuState.M: (MenuState.S(i) ⇔ (Preselect ∈ MenuState.A))
Current = Number'(Total - 1)

Operations:

The next symbol of KeybIn is

↑: Precondition: c = Current ∧ OK = (Vertical ∈ A ∧ Number(c) ≠ 0)
Postcondition: OK ⇒ Current = Pred(c) ∧ ¬OK ⇒ Current = c
The PDB package

For vertical menu: go to previous item (if any).

\[\downarrow: \text{Precondition: } c = \text{Current} \land \text{OK} = (\text{Vertical} \in A \land \text{Number}(c) \neq \text{Total}-1)\]
\[\text{Postcondition: } \text{OK} \Rightarrow \text{Current} = \text{Succ}(c) \land \neg\text{OK} \Rightarrow \text{Current} = c\]

For vertical menu: go to next item (provided current item is not the last item).

\[\rightarrow: \text{Precondition: } c = \text{Current} \land \text{OK} = (\neg(\text{Vertical} \in A) \land \text{Number}(c) \neq \text{Total}-1)\]
\[\text{Postcondition: } \text{OK} \Rightarrow \text{Current} = \text{Succ}(c) \land \neg\text{OK} \Rightarrow \text{Current} = c\]

Horizontal menu: next item.

\[\leftarrow: \text{Precondition: } c = \text{Current} \land \text{OK} = (\neg(\text{Vertical} \in A) \land \text{Number}(c) \neq 0)\]
\[\text{Postcondition: } \text{OK} \Rightarrow \text{Current} = \text{Pred}(c) \land \neg\text{OK} \Rightarrow \text{Current} = c\]

Horizontal menu: previous item.

Home, PageUp: Postcondition: \text{Current} = \text{Number}'(0)

Go to begin of menu.

End, PageDown: Postcondition: \text{Current} = \text{Number}'(\text{Total} - 1)

Go to end of menu.

Insert: Postcondition: \text{S(Current)} = \text{TRUE}

Add current item to list of selected items.

Delete: Postcondition: \text{S(Current)} = \text{FALSE}

Remove current item from list of selected items.

\text{string } x \text{ of alfanumeric characters, terminated by Enter:}

\[\text{Precondition: } (\text{OK} = \text{Add} \in A) \land t = \text{Total}\]
\[\text{Postcondition: } \text{OK} \Rightarrow (x \in M) \land \text{S}(x)\]
\[\land \text{Total} = t + 1\]
\[\land \text{Current} = x\]
\[\land \text{Number}(\text{Current}) = t\]

If 'Add' in attributes: Add \(x\) to menu and select it.

Esc: terminate with \text{DoMenu} = M'

Quit menu operations, leaving original set unchanged.

Enter: terminate with \text{DoMenu} = \{m \in M \mid \text{S}(m)\}

Leave menu with set of selected items.
6.3 Special menus.

In practical systems, such as PDB, one may encounter several kinds of menus:

1. The single choice menu.
2. The options menu.
3. The interactive directory.

The first two are fixed menus that differ in the number of items that may be selected. The single choice menu often offers a number of alternative operations, one of which will have to be performed as indicated by the user. This may be implemented by Domenu with attribute 'Single'.

The options menu allows several items to be selected (and de-selected). The selected items are often indicated by tick marks. Initially, some items may already be selected. PDB only allows preselection of all items (attribute 'Preselect') or none. Apart from 'Preselect', none of the options may be set.

An interactive directory is a kind of menu that has no fixed length. The user may add and remove items (file names) at will, either by indicating them directly in the directory or by giving a command, like "copy A B" or by "dragging" in a GUI file manager. When using DoMenu this mode is indicated by the 'Add' attribute.

6.4 The set editor.

Sets of links, keywords or objects are updated through menus by deselecting and/or adding items to an original set \( S_1 \). This results in a modified set

\[ S_2 = \text{DoMenu}(S_1, \text{KeybIn}, A). \]

Now the elements to be added and to be removed from the system are found by set difference:

\[
\text{AddSet} = S_2 \setminus S_1 \\
\text{RmveSet} = S_1 \setminus S_2.
\]

Example:

When the user has requested a modification of the list of keywords of the current object, DoMenu presents the current keywords, all being preselected. The user now may add and remove items, creating a new set of keywords. Subsequently the object is updated, not by removing the old set of keywords and adding the new set, but by removing only the 'RmveSet', as indicated above, and adding the 'AddSet'.

Pim Lemmens
7 Conclusions.

In this chapter we briefly summarise how the PDB package attempts to realise the requirements of ease of use, reliability, versatility and extensability.

Ease of use through simple commands and versatile menus.

Ease of use is realised by offering simple one-letter commands for frequently used operations and by a uniform menu structure for all other kinds of operations, both for consulting and for updating a hypertext. Response speed is optimal through exploitation of the possibilities offered by redundant storage in different representations.

Reliability through redundancy.

The dual representation of the hypertext data requires extra storage space, but has two important advantages: not only speed is increased because each operation uses the representation that offers the most efficient execution, also reliability is boosted: system crashes that destroy part of the data may be overcome by restoring one representation using the other.

Versatility through named links and keywords.

The link implementation used by PDB offers more possibilities than the usual schemes: links may be identified by their targets, as in HyperCard, but the connections between objects may also be identified by special names, like "yes" and "no", or "answer 1", "answer 2", .. as responses to a question posed by the object.
Keywords, used for a rough characterisation of objects, serve as the basis for a global search for a specific object.

Extensibility through external viewers and editors.

As the contents of an object may be in any format, and can be used by any computer program, there are no limits to the possibilities. New programs and devices may easily be accomodated without any change to the hypertext program.

Future extensions.

It would be very useful, as shown in section 1.3, to have different views on the same collection of data. Furthermore, ordering nodes and relations in a limited number of categories may also help to get a clear picture of the structure of a hypertext. This may be done by registering node types and relation types in one or more meta-hypertexts. A meta-hypertext would show the different categories of nodes, with their properties (like "ASCII text node", "black and white raster image", etc., c.q. 1:1 relation, or 1:n or m:n, or "connecting node of category A to one of category B"). It could be used to enforce disciplined behaviour of users. The meta-hypertext could easily be stored as part of the hypertext it describes, with 'type' relations between normal nodes and meta-nodes.
References.


The PDB program is meant to be used as an authoring system. It should assist in creating texts, from the collection of rough material to the preparation of the final draft. PDB is a kind of database program of information objects. These objects may be of any kind: texts, drawings, pictures or sounds. The program allows you to quickly record a concise characterisation of the objects the hypertext will contain. It will store pieces of text or images in such a way that they may easily be retrieved and modified. And then it will enable you to add structure, placing the pieces in a mutual context.

Each object, each piece of information in a hypertext will have a unique name. It may be found by that name in the database. An object may be characterised by a number of keywords. It may be retrieved from the database through those keywords. The user may also move among the objects by following the links that connect them.

During an interactive hypertext session, there is always one object, the actual or current object, of which the information is presented on the screen. At the beginning of a session the current object is the start object (always called 'Start'). The information on an object is contained in two different pages: the default page displays the name of the current object, its keywords, and the links from this object to another. This page also shows the main command list. The other page is the contents page showing the contents (text or picture or whatever) of the current object. In addition to these pages there are the menu pages for choosing from, or modifying, certain sets of items.

There are a few things you should remember before starting to use the system:

Commands are single keystrokes. So, if you would like to access the contents of an object, you simply type the character 'c' at the prompt of the main command list.

Menus are vertical or horizontal lists of (object, link or keyword) names. You may choose one of them by moving to it using the cursor keys. Pressing 'Enter' finalises your choice and makes you leave the menu. Pressing 'Esc' returns you to the situation before the menu appeared.

The string "\" in reverse video requires you to push any key in order to view the rest of a text.

The command list on the object info page offers the following commands:

Command 'k' from the main list gives access to the keyword operations.

Command 'I' opens up the world of links.

The 'o' command will allow you to access an object through its (object) name. Command 'o' with object name 'Start' will take you back to where the session began.

'm' gives access to a group of miscellaneous commands.

For more information, see below.

General remarks.

During a PDB session you will often be asked to enter a command in the form of a single
The PDB package

letter, e.g.

object does not exist. c(reate)?

to which you may reply by typing a 'c'. You may, however, also simply press 'Enter' or any other character. In this case no action is performed and the next menu will be presented. Single character commands need not be followed by 'Enter'.

In addition to the commands several other types of menu may be presented. There are menus that offer you a choice of one item and there are menus from which you may choose more than one item. You may even add items to them. More on menus later on.

Names (of keywords, links or objects), like "I/O-procedures", may contain any character, except control characters. Character recognition is case-sensitive, so "Start" is different from "start".

Continuous texts and pictures may be changed using a word processor or a paint program. The way to do it depends on the specific word (or picture) processor used.

Menus.

Menus appear as lists of items, one below the other, or one to the right of the other, of which one is indicated at any time. This item is displayed in reverse video. The chosen item (or items) will be marked by the ' ' symbol right before the item name.

On single choice menus you should make your choice by indicating the item using the up- and down-arrow or right- and left-arrow keys, depending on the kind of menu, or using the 'Home' ('PageUp') or 'End' ('PageDown') keys, which respectively take you to the start or the end of the menu.

There are also menus that allow you to choose more than one item. Sometimes all items presented will already be chosen as the menu appears. Choices are made by indicating an item and pressing 'Insert'. Pressing 'Delete' cancels the choice of the indicated item. You may also add items by simply typing a text. This text will appear as the last item of the menu. Left and right arrow keys, Delete and Backspace keys may all be used for editing while entering new text. Pressing 'Enter' adds the typed text to the menu and automatically chooses it. Pressing any of the other special keys (except 'Home', 'End', 'Right', 'Left', 'Delete' and 'Backspace') erases the text and returns you to the menu.

Pressing the 'Enter' key finalises your choice and makes you leave the menu. Pressing 'Esc' allows you to leave the menu without finalising your choice (all selections are undone.)

Objects.

An object represents a node in the hypertext network. It has several attributes: its name, its contents (a text, a picture or a program), the set of associated keywords and the set of links it owns.

An object may be accessed directly. Its name may be used to fetch it and make it the current object. After typing 'o' the system will ask for an object name. If the name you provide is unknown to the system, it will ask if a new object having that name is to be created. Typing 'c' creates the object and makes it the current object. You will then be asked to supply keywords for it. This set of keywords may be changed later on. You 'll also have to add one or more links.
those of the current object. Using as "find" keywords only "writers" and "English" will give you all English writers from the hypertext, provided they all have "writers" and "English" as keywords.

A keyword may be identical to some link or object name.

Contents of objects.

An object may contain a piece of text or a picture or a program or even a piece of music. There are many possible kinds of objects. In order to distinguish these kinds, the contents of each object has a kind specification, in the form of a kind name. To each kind name belong two processor names: a viewer name and an editor name. Viewer and editor names may be normal operating system command names or program names (without input file parameters). Each time a user wants to examine the contents of an object, the viewer is invoked, with the contents of the object as a parameter. When the user indicates (s)he wants to modify the contents of an object, the editor is invoked. Viewers may be display programs, interpreters or loaders or any other program that does not change the contents of the object. Editors may be text editors, paint programs, compilers, etc.

Miscellaneous operations.

The choice of 'm' from the main list gives access to:

definition file input: used to extend the hypertext by the definitions contained in the file you specify. For more details, see below.

configuration: allows you to specify a viewer and an editor for each specific kind of object data (to be activated through the 'contents' command). More information further on.

garbage collection: first creates a backup of the current hypertext and next compacts it, releasing unused space to the file system. The objects.bak file created may be used to restore the hypertext (by the 'd(efinition file input)' option), in case you want to return to the current version.

quit: leave PDB.

Definition file input.

This choice activates an interpreter for hypertext specifications and procedures for appending to and updating of the hypertext database. This option may be used to enter data from a flat (text) file or to restore a previous version of a hypertext (in this case use 'objects.bak' or a renamed version of 'objects.fdb' for input). One may also enter configuration data through a definition file.

A text file should be processed first to be readable by the interpreter. It should be divided among a number of objects (and comments, for the parts that need not occur in the hypertext) by inserting 'Object' (or 'Config' or 'Comment'), 'Contents', 'Kind' and 'Value' instructions plus brackets, and commands for keywords and links should be added.

The three files used by PDB to store a hypertext (Locations.fdb, References.fdb and Objects.fdb) may be damaged or a hypertext may be destroyed in some other way. In that case, the backup file from a previous garbage collection may be used to restore the hypertext to the state of just before that operation. First the '*.fdb' files should be removed, second new (blank) files should be created using 'newpdb', and then, within the 'pdb' program, the 'd' option of 'm(ore)' should
be chosen. As an input, the file 'Objects.bak' should be specified (or any copy or rename of such a file or of the original 'Objects.fdb' file).

**Interpreter error messages.**

Possible error messages from the interpreter:

0: Unexpected symbol.

1: Illegal keyword in contents specification ("Kind" or "Value" expected).

2: String expected in simple set.

3: Closing string not "Targets" (as it should be).

4: Illegal keyword in link specification.

5: '{' expected in link specification.

6: Closing string not "Links".

7: Closing string not "Value".

8: Closing string not "Contents".

9: Illegal keyword in object specification.

10: '}' or 'l' expected in specification file.

11: Closing string not "Object".

12: System error - notify system developer.

13: Closing string not "Comment".

14: Illegal keyword in hypertext specification.

15: Illegal symbol in hypertext specification.

Sometimes symbol numbers will occur in error messages. These indicate the following symbols:

0: '{'

1: '}'

2: 'l'

3: string of any length

4: end of file

Configuration specifications may give rise to the following messages:

101: Illegal keyword in configuration specification.
102: Closing text not 'Config'.

103: Symbol is not a keyword (such as 'Config', Comment', 'Object').

104: (warning) No configuration in extension file (at the start of a session).

Configurations.

The hypertext contains a table of viewers and editors that is created using the 'configuration' command. First you are invited to enter a name for the kind of the data you are considering (e.g. 'text', 'picture', 'program'), next you should specify which program should be used to display or otherwise animate the data (in the form of an operating system (shell) command), and finally an editor command is required that will allow you to change the data. 'own' as a viewer uses built-in software to display plain ASCII texts. An empty string or 'none' as viewer or editor suppresses viewing or editing, respectively. When, for example, you need to use an editor both for viewing and editing, you'd better suppress the viewing option. Otherwise you may get the false impression during viewing, that the contents may actually be modified. But any modifications are lost when returning to PDB from viewing mode.

About PDB.

PDB was created at the Eindhoven University of Technology as a front-end to the ELDORADO storage system, also developed at that university. It is written in Logitech Modula-2 to run on MS-DOS machines.

Inquiries to the author:

W.J.M. Lemmens
University of Technology
Faculty of Mathematics and Computer Science
Computer Science Group
P.O. Box 513
5600 MB Eindhoven.

The author also welcomes any bug reports, suggestions for improvement and extension, and examples of use of the package.
Appendix B: Description of ELDORADO.

B.1. Introduction.

B.1.1. Database states.

A database contains a (possibly large) set of data items that are uniquely identifiable - no item will occur twice in the database. Many items have connections to other items, according to certain rules. An item may, or may not, be associated to a value object of a certain type. It may be retrieved from the database on account of its data value or because of its connection to other items.

A database state may be described by a 3-tuple:

\[ DB = \langle \text{Obj}, V, \text{Link} \rangle \]

Where \( \text{Obj} \) = a set of indices designating the items of the database.

\( V \) = a (partial) function: indices \( \rightarrow \) values, that assigns a value to the items from \( \text{Dom}(V) \subseteq \text{Obj} \). Not every item needs to have a value.

\( \text{Link} \) = a ternary relation: labels \( \ast \) indices \( \ast \) indices. Each link is characterised (uniquely identifiable) by a label \( l \), a 'from' object \( i_1 \) and a 'to' object \( i_2 \).

\[ \langle l, i_1, i_2 \rangle \in \text{Link} \]

\( \forall k \in \text{Link}: k.i_1 \in \text{Obj} \land k.i_2 \in \text{Obj} \)

An item should be accessible and identifiable. Accessibility means: it should be possible to formulate a query that retrieves the item from the database. Or, in other words: it should either have a value or it should be reachable through a link from another accessible item. Identifiability means: it should in some way be distinguishable from the other items, either by its value or by its connections.

B.1.2 The functional model.

In a functional data model the items are grouped into categories. A category is a set of items that share a number of features: they are associated with the same type of data object, and with the same pattern of connections to other items. Categories are linked by functions. A function is a set of connections among items. Potentially, all members of a category may be connected to items of a category that has a function link to that category.

A database skeleton according to the functional model is a 5-tuple:

\[ FS = \langle \text{CI}, \text{FI}, D, R, T \rangle \]

in which \( \text{CI} \) and \( \text{FI} \) are sets, indicating respectively: category indices and function indices. They identify the functions and the categories in the database.

\( D, R \) and \( T \) are functions:

\( D \), the domain function, associates a domain category from \( \text{CI} \) to every function index from \( \text{FI} \).

\( R \) is the range function of the functions \( \text{FI} \).
The PDB package

T is a function that links each category name to a set of possible values, its type. This set may be empty. In this case the associated category is a so-called 'ghost'-category, of which the elements have no value, only two or more links to elements from other categories.

A database state in the functional model may be described by a tuple:

\[ DB = \langle Obj, V, Links, PC, PF \rangle, \]

where PC and PF correspond to a partitioning of Obj and Link, respectively. PC is a function that assigns a category index (from set CI) to every object. PF is a function that assigns a function index (from FI) to every link.

B.1.3. ELDORADO.

ELDORADO is an implementation of the concepts defined above. Compared to the definition of a functional database, it poses some restrictions, but also offers some extensions. In addition to the conditions mentioned above, ELDORADO requires the elements of a category to be fully ordered, or rather, it requires the elements of each category type to be fully ordered. So for each category a successor function is defined that, given an element from the category, provides the next higher element (if any).

A feature of ELDORADO are the extensions: each item may have some free format data associated with it. These data do not contribute to the accessibility or identifiability of the item, but they may be stored with the item and accessed to serve some purpose in a specific application.

\[ \text{EXT: } DB.\text{Obj} \rightarrow S \text{ is a function that associates an arbitrary data sequence to a number of items from the DB. Actually, it's a partial bijective function.} \]

The ELDORADO system is oriented towards collective operations. Wherever possible, it operates on subsets of categories and functions, not on individual elements. A further feature of the system is its use of indices throughout its operations. Integer, real or string values are only used by the user for input and by the system for presentation of final results.

The ELDORADO system is built in a number of levels, with the storage system at the lowest level. The storage system operates fully at the level of the general database states, described above. It has nothing to do with categories and functions. The procedures of this system will be used to find and update links among items and data and extensions connected to them (chapter 2).

The PDB package does not use the higher level ELDORADO operations, those concerned with categories and functions, so these are not described here. It does use some of the temporary data structures of ELDORADO, however. These may be used in expressions containing database data (chapter 3).

B.2. The storage system.

B.2.1. Introduction: Items and Refs.

All categories of an ELDORADO database, and possibly of various such databases, will be stored in the same files. It would be impractical to open a new file every time a new category is going to be accessed. So the files used will not correspond to the categories created. In fact, even the structure of the files used will not mirror the
structure of the DB in terms of categories and functions.

At a certain level we are not aware of categories and functions in much the same way as, looking through a microscope, a plant is not seen to consist of leaves and stems, but of (more or less differently shaped) cells. In our case these cells correspond to the items of the database. An item may have a value of some type and connections to other items. It may be found through its connections, by way of an other item, or it may be accessed on the basis of its value. For this purpose, there will be an index mechanism that, given a certain value from a specified category, allows us to locate the associated item.

So every item will eventually contain at least one of the following: a value and a number of refs (indices) of other items, where every ref is associated with a certain label at the item itself. If that would not be the case, the item could never be retrieved.

The storage system, which constitutes the lower layer of the ELDORADO package, is only concerned with items, not with categories or functions. It implements a kind of semantic net in which each item is a separate object, uniquely distinguishable from all others, as defined above. Internally, all items are identified by a unique number, their reference (ref). For practical reasons the identifiability of the items is not verified by the system. Accessibility is only verified to a certain degree: ELDORADO only checks whether an item has a value or some connection to an other item.

B.2.2. Files, Records and Addresses.

If we could sufficiently augment the magnification of our microscope, at a certain moment the cells would dissolve into molecules before our eyes. As we have seen, the "cells" of our system, the items, are built from various components. And these components will have their own inner structure. Different components will be stored in different files where each file will contain records of a different type.

The representation of our DB within the computer system is as follows:

\[ \text{DBRepr} = \langle \text{Index, References, Objects, Locations} \rangle \]

The database is stored in four files: an index file for access by way of values, a reference file to store the links among the items, an objects file to find extensions associated with the items and a location file that holds the locations of items within the reference and objects file.

The index file is not used by the PDB package, so we won't go into it any more.

Next we have the reference file. The records of this file will contain values of some category type and references of other items.

\[ \text{References} = \langle \text{Freeloc, Records} \rangle \]

where References.Freeloc points to the location of free space at the end of the file and

\[ \text{References.Records} = \{ \langle \text{id, val, fr, br, el} \rangle \mid \text{id} \in \text{DB.Obj} \} \]

id is the ref of the item itself, val is its value, el the length of the extension associated with the item. val and el are optional elements. They will only be present if the item has a value or an extension, respectively.

\[ \text{fr} = \langle \text{nf, fpl} \rangle, \]
with \( \text{nf} \) = \#fpl the number of forward (functional) references and

\[ \text{fpl} = \text{A set of tuples} \langle \text{l, fp, Ip} \rangle, \] with \( \text{fp} \) indicating the range value associated with the current element for the function concerned, and \( \text{Ip} \) a reference to an other
The PDB package

element from the same category with the same function value.

\( f \) may be considered as a function with domain \([0 .. \text{nf}-1]\) and range \( ID * ID \)

\[ \forall k \in \text{Link}: \exists r \in \text{References.Records}: k.i1 = r.id \wedge k.i2 \in \{f.fp | f \in r.fr.fpl\} \]

\[ \forall x \in \text{References.Records}: \]
\[ \forall f \in x.fr.fpl: \]
\[ f.lp = \text{succ}(x, \{y \in \text{References.Records} | y.fr.fpl(f.l).fp = f.fp\}).id \]

\( \text{br} = <\text{nb}, \text{bpl}>, \)
where \( \text{nb} = \#\text{bpl} \), the number of backward references,
\( \text{bpl} \) = a set of tuples \(<l, \text{bp}>>\) indicating an item from the inverse function
associated with label \( l \). \( 0 \leq l < \text{nb} \).

\[ \forall x \in \text{References.Records}: \]
\[ \forall b \in x.br.bpl: \exists k \in \text{N}: \]
\[ b = \min\{m.id | m \in \text{References.Records} \wedge m.fr.fpl(k).fp = x.id\}, \]

In other words: \( \text{bp} \) points to the beginning of the sorted list of items that have a
specific reference to the current item. There is always a close correspondence between
the forward label \( m.fr.fpl.l \) of some domain category and the backward label \( n.br.bpl.l \)
of the range category of a given function.

An item may also have an entry in the objects file, where the extensions are
stored.

\( \text{Objects} = \{\text{EXT}(i) | i \in \text{DB.Obj}\} \)

The position of items within the files may change as other items are added or
removed and unused filespace is reclaimed. If we want to use their mutual connections
we need to keep track of the items as they move. There is a special file for this
purpose. It contains the locations of the reference and data parts of all the items:

\( \text{Locations} = \{<\text{id}, \text{rloc}, \text{dloc}> | \text{id} \in \text{N}\} \)

and will be updated every time a location is changed by garbage collection or update
operations. So, if the ref (id) of an item is known, it will always be possible to locate
its components (rloc and dloc). Existing items with no extension have a Null dloc
value.

Locations contains more than the current set of indices. It also contains indices
that have once been used, but are removed from the database since. These are
available for use by newly created items. They are characterised by a Null value for
rloc. Their dloc values are used for the construction of a list of free indices: they
point to an other free element.

The data in this storage scheme are intentionally made redundant, e.g. by storing
the item identification and the lengths of the reference lists with every item. This is
done for reasons of reliability and efficiency: if part of the stored data is damaged,
much of it may be retrieved by inspecting the undamaged part. This redundancy
further limits the number of disk accesses needed, especially to the dictionary.

If, for example, the locations file had been destroyed, it could be restored by
going through the reference file, item by item, and noting the locations of the items
and the identifications stored at these locations. Furthermore, the addition of a new
function to a database that already contains a large number of items, does not force
us to change all the items of the domain and range categories, because of added
labels, as each item has its own indication of the number of labels. Only those items that play a role in the new function need to be adapted.

B.2.3. References and Objects Operations.

Operations on the storage system serve three main purposes: update, retrieval and maintenance.

Retrieval operations find attributes of items. They are used e.g. to get a specific pointer from a specific item, or its value or its extension. These operations do not change the state of the system, so there will be no risk of violating the constraints on the data structures.

Update operations create or remove items, they add, change or remove pointers or data in the References file, or they add, change or remove extensions in the Objects file. Secondary effects of these operations are that the Locations files may also be changed.

Maintenance is done in order to keep system performance up or to remove defects. Maintenance operations remove unused items and reclaim patches of storage space that do not contain useful data anymore. They may also repair a damaged file. The contents of the database are not affected by maintenance. Only the location of data on the storage medium may change.

The three groups of operations mentioned above may each be subdivided into a group of operations on links and values (in the Reference file) and a group of operations on extensions (residing in the Object file). As all access to the files is buffered, the relevant records should first be transferred to the buffer before any operations are performed, and after update operations be transferred back to disk to make the changes permanent. A buffer has an area for the links and an area for the data (value and extension) of a given item. There may be any number of buffers active for a given set of files. The user is advised to create at least one for every (meta-) database that shares the same files.

B.2.3.3. Implementation notes.

The pointer fields of the References records contain item identifications. The location of the associated records may be found in the RLoc fields of the location table.

Forward pointers indicate a range item. Backward pointers point to the start of a list of items that have a forward pointer, with a specific label, that points to the item in question. The elements of the list are arranged in ascending order of item identifications. These elements are connected by way of the list pointer with the same label as the forward pointer. The last element of the list has a Null list pointer.

UpdateFL and RemoveFL operate on a higher level than the other procedures mentioned here. Not only will they affect the forward pointer of the specified item, they will also update the backward list, thereby changing backward and connecting pointers, if necessary.

B.2.4. Location file operations.

The location file consists of records that contain two file addresses (32-bit numbers), rloc and dloc, one for the References file and one for the Objects file. The first record (item Zero) has a special function: it holds the pointer to the entry of the first free item. The list of free items is linked through the dloc field, each dloc pointing to the next free item. These items are characterised by a zero rloc field.
Location file operations use a buffer, in the same way as the operations on the Objects and References files. In this case, however, there is only one buffer. The operations on the location file are totally transparent to the user. They are contained in a local module that is only accessible to the storage operations mentioned above.

B.2.5. Maintenance operations.

The removal of an item from the database, or the removal of data from an item, even a change in dimensions of the data record or the addition of a link to a record from the Reference file, each of these occurrences will leave a patch of unused space in the file concerned. Because these patches are normally not reused, the files will have a tendency to grow, even if the database would be shrinking.

Free spaces in the References file, are characterised by the Null item identification left there by PutRef. The Null identification is followed by a CARDINAL indicating the length of the patch. An item record in References that has no pointers to other items and has no data associated with it, is also considered free for reuse. The garbage collector finds these pieces of free space and unites them to larger units, where new or changed records may find the room they need.

At the heart of each garbage collector there is a procedure that processes one record each time it is invoked (ShiftRRec in CleanupRef and ShiftDRec in CleanupData). If the current record is free, it is added to the free space. If it is occupied, it is moved into the free space (if any) found preceding it in the file concerned. In this way the free space is moved upward step by step, mopping up any free patches encountered on the way.

Although the garbage collection process may be interrupted by query and update operations, the procedures mentioned should be indivisible. Any break would jeopardise the integrity of the file. Currently, garbage collection is not implemented as a concurrent process, but it should be in the future, enabling update operations to reclaim any free space collected so far (using procedure Claim).

B.2.5.1. Storage management procedures.

To allow for garbage collection, the References file has been extended by three new variables: ScanPos, FirstHole and HoleLength. ScanPos is used and updated by the ShiftRRec and ShiftDRec procedures, indicating the current position in the file. FirstHole points to the current position of the free space (initially at the end of the file). HoleLength is the length of the free space found so far. It is equal to the length field of the record indicated by FirstHole. As mentioned before, the first record of each file contains a pointer to the end of the data. It is updated to make it equal to FirstHole after each full scan.

Every time garbage collection is invoked it starts at the bottom of the file and works its way upward, record by record, until the top of the file is reached. The free space collected on the way is left there, in the free record that always resides on the top of the file. Initially this record is empty: it only consists of the Null identification and a Length field with zero value. As the file expands it will shrink or move along, dependent on its contents. The value of the first field of the file (FreeLoc) will always correspond to the position of this record.

The whole process of disk storage management is dependent on the shift and the claim procedures. These perform the following actions:
ShiftRRec:

preconditions:
    \( p = \text{ScanPos} \)
    \( f = \text{FirstHole} \)
    \( r = \text{length of record indicated by ScanPos} \)
    \( h = \text{HoleLength} \)

postconditions:
    \[
    \text{FirstHole} = \begin{cases} 
    f, & \text{if } f > p \land \text{Accessible}(p) \\
    f + r, & \text{if } f < p \land \text{Accessible}(p) 
    \end{cases}
    \]
    \[
    \text{HoleLength} = \begin{cases} 
    h, & \text{if } \text{Accessible}(p) \\
    h + r, & \text{if } \neg \text{Accessible}(p) 
    \end{cases}
    \]
    \( \text{ScanPos} = p + r \)

Accessible\((p)\) decides whether the record at position \( p \) is accessible within the database.

Claim\((F, l, p)\):
    input parameter: \( l \)
    input/output parameter: \( F \)
    output parameter: \( p \)

preconditions:
    \( f = F.\text{FirstHole} \)
    \( h = F.\text{HoleLength} \)

postconditions:
    \( F.\text{FirstHole} = f + l \)
    \( F.\text{HoleLength} = h - l \)
    \( p = f \)

For the sake of clarity, this definition is kept incomplete. If the required length \( l \) is larger than the HoleLength then free space is acquired from the end of the file.

When garbage collection is not active, the values of FirstHole and HoleLength will correspond to the FreeLoc pointer and the Length field of the last record, respectively.

The extension file is cleaned up using data from the reference file as it is scanned. The extensions of the items encountered are copied to a new extension file, one by one. The old file is renamed to objects.bak.

B.3. Temporary data structures.

B.3.1. Introduction.

The basic structures of ELDORADO are: Atoms, Sets, Categories, Tfunctions, Functions and Tables. Atoms will be of one of the following types: empty, integer, real, string, ref or bool. Sets are either of type integer, real, string or ref, or they are empty. Tfunctions are functions from ref to some value, whereby this value may be an integer, a real, a string or a ref. Functions may be considered as composed of refs, and tables are sets of named temporary functions.

Atoms, sets, tfunctions and tables will not reside on background memory. They will cease to exist when the program that uses them is terminated. These structures
The PDB package

are represented in memory by an ordered list of elements, together with some asso-
ciated data.

B.3.2. Atoms.

Atoms are single elements of a certain type:

\[
\text{Atom}(X) \Rightarrow \text{Type}(X) \in \{\text{empty, int, real, identifier, ref}\}
\]

\[
\text{Type}(X) = \text{ref} \Rightarrow X \in \text{DB.Obj}
\]

ref atoms represent some element from the database.

Because atom operations are relatively expensive as compared to the collective (set and temporary function) operations, they should be used as scarcely as possible.

B.3.3. Sets and set operations.

A set is characterised by the type of its elements and, if this type is ref, the category that holds them, together with the list of elements itself:

\[
\text{Set}(X) \Rightarrow (\exists y \in \{\text{empty, int, real, identifier, ref}\}: \forall x \in X: \text{Type}(x) = y)
\]

All elements of a set have the same type.

\[
(\text{Type}(X) = \text{set of ref}) \Rightarrow \exists i \in \text{CI}: X \subseteq \text{PC}^\ast(i)
\]

Sets that have been emptied (by SetExtract, SetRetain or SetRemove) may not be reused for elements of a different type (or category, in the case of ref sets) until they are re-created for that purpose by CreaSet.

B.3.3.2. Implementation notes.

A set is implemented as a list of elements, arranged in ascending order. In fact, it is a circular list, where the largest element is connected to the smallest. The set descriptor contains a pointer to this largest element. For list traversal, this requires an extra step before the start, the smallest element, is reached. But when adding elements in ascending order, which is a rather frequent operation, the efficiency is very much larger than in the case where each insert action would have to start at the smallest element. Also, the determination of both the set maximum and minimum are very efficient operations. They are used e.g. in binary set operations (union, intersection, difference) for a first check of whether two sets are disjunct.

\[
\text{Set}: S = <t, \text{cat}, \text{cont}> \quad \text{(Set descriptor)}
\]

\[
\begin{align*}
& t \in \{\text{empty, int, real, identifier, ref}\} \\
& t = \text{ref} \Rightarrow \text{cat} \in \text{CI} \\
& t = \text{empty} \Rightarrow S.\text{cont} = \\
& S.\text{cont} \neq 0 \\
& \Rightarrow S.\text{cont} = <v, V, \text{Next}>, \text{ with Next}: V \rightarrow V \text{ and } v \in V \\
& \wedge (S.\text{cont}.v = \text{max}(V) \wedge S.\text{Next}(v) = \text{min}(S)) \\
& \wedge \forall w \in V: w \neq v \Rightarrow \text{Next}(w) > w
\end{align*}
\]

Tables and temporary functions are not used by PDB so they are not covered here.
Appendix C: PDB definition modules.

C.1 Overview of ELDORADO modules.

The layered structure of ELDORADO is reflected in the connections among modules:

Level 1. EldAlg. General utility module.
Only imports procedures and types from standard modules.

2. Elstorag. Manages References and Objects files.
ElSets. Handles temporary sets.
These modules import only from EldAlg and standard modules.

Imports from EldAlg, ElStorag and ElSets.

Imports from EldAlg, ElStorag, ElSets.

C.1.2 ELDORADO definition modules.

DEFINITION MODULE EldAlg;
(* Last modified: 880925 *)

EXPORT QUALIFIED elid,value,tvalue,identifier,Null,datatype,
IOError,RTError,ELError,ELWaarsch,CmpVal,CmpRef, NullRef;

TYPE
   elid = RECORD
      hi: CARDINAL;
      lo: CARDINAL;
      END;
   identifier = ARRAY[0..19] OF CHAR;
   datatype = (empty, int, rl, wd, ghost);
   value = RECORD
      CASE datatype OF
         int : i: INTEGER |
         rl : r: REAL |
         wd : w: identifier |
         ghost: g: elid |
         empty: e: elid |
      END
      END;
   tvalue = RECORD
      t: datatype;
      v: value;
      END;

VAR Null: elid;

PROCEDURE IOError ( en, ec: CARDINAL);
   (* IO error while reading or writing files program is aborted *)
PROCEDURE RTError (en: CARDINAL);
(* run time error
program is aborted *)
PROCEDURE ELError (en: CARDINAL);
(* Specific ELDORADO error
program is aborted *)
PROCEDURE ELWaarsch (en: CARDINAL);
(* ELDORADO Warning
program will be continued *)
PROCEDURE CmpVal(t: datatype; v1, v2: value): INTEGER;
(* Compares two values (or refs):
CmpVal = -1 => v1 < v2,
CmpVal = 0 => v1 = v2,
CmpVal = 1 => v1 > v2. *)
PROCEDURE CmpRef(v1, v2: elid): INTEGER;
(* Compares two refs:
CmpRef = -1 => v1 < v2,
CmpRef = 0 => v1 = v2,
CmpRef = 1 => v1 > v2. *)
PROCEDURE NullRef(Ref: elid): BOOLEAN;
(* Is ref Null? *)
END EldAlg.

DEFINmION MODULE ElStorage;
(* Made by: W.J.M. Lemmens, Eindhoven University of Technology (TUE).
Latest modification: 1990-01-19. *)
(* Rules for Eldorado Storage System operations:
1) Before any operation can be performed on a new element, it should first be created using NewElem.
2) The lifetime of an element extends from its creation to its removal, by RemoveEl. No two "living" elements will have the same identification el (Type(el) = elid).
3) An element may have zero or one data record. Data records are created by NewData followed by PutData and removed by RemoveData.
4) An element may be removed only if it does not have a data record and there are no references from and to the element. (RemoveEl will check this.)
5) Every element belongs to a specific buffer (TYPE database). During its whole lifetime it should reside in that buffer, and that buffer only, while operations on that element are performed. (This will not be verified by the system.)
6) Operations on reference fields or the value of an element should be preceded by NewElem or GetRef, not followed by NewElem or GetRef for an other element in that buffer.
7) Operations on data records should be preceded by NewData or GetData, not followed by NewData or GetData for an other element in that buffer.
8) NewElem should be followed by UpdateFL or PutRef for that buffer (before
GetRef or another NewElem) in order to make the element permanent. Otherwise
there will be an element identification but no reference record. RemoveEl will
remove both the reference record and the element identification.

9) One or more operations that update the extension (PextDat, AppendExt) should
be finalised first by calling PutData or CommitExt before performing PextDat on
the associated buffer. Failure to do so will result in loss of updates (PextDat -
the data will not be changed) or combination of two extensions in one.

10) Every program that uses these operations should start by initialising the
necessary buffers (by calling IniDB). It should close the files used by calling
CloseRF and CloseDF before termination.

11) Garbage collection will remove inaccessible items: Newly created items or
items with all references and data removed will have disappeared after
CleanupRef.

This module may produce the following messages:

I/O Error 01: Write Failure on location table.
ELDORADO Error 02: Attempt to create new location table while old one
still exists.
ELDORADO Error 03: Attempt to create new reference file while old one
still exists.
I/O Error 04: Write failure while creating new reference file.
I/O Error 05: Write failure on reference file.
I/O Error 06: Read failure on reference file.
I/O Error 07: Write failure while removing reference record.
ELDORADO Error 08: Attempt to create new data file while old one
still exists.
I/O Error 09: Write failure while creating new data file.
I/O Error 10: Write failure on data file.
I/O Error 11: Read failure on data file.
Run-time Error 12: Not enough memory to load existing data record.
Run-time Error 13: Not enough memory available for new data record.
I/O Error 14: Write failure while removing data record.
Run-time Error 15: Not enough memory available for new extension data.
ELDORADO Error 16 (fatal): Attempt to remove item while it still held
a data value or references to other items.
ELDORADO Warning 17: Attempt at garbage collection on an empty file.
Run-time Error 18: No room for database buffer.
ELDORADO Warning 19: Location table not present.
ELDORADO Warning 20: Reference file not present.
ELDORADO Warning 21: Data file not present.
I/O Error 22: Write error while creating free file space.
I/O Error 23: Write error while updating free pointer in reference file.
I/O Error 24: Write Error while updating free pointer in objects file.
ELDORADO Error 25: Cannot find indicated item.
*)

FROM SYSTEM IMPORT ADDRESS, BYTE;
FROM FileSystem IMPORT File;
FROM EldAlg IMPORT elid, tvalue;

EXPORT QUALIFIED database, DB,
IniDB, CreateRF, NewElem, GetRef, GPntFP, GBpList, GPntLP,
The PDB package

UpdateFL, RemoveFL, PutRef, CloseRF, CreateDF, NewData, GetData,
GInfVal, PExtDat, AppendExt, HasExt, CommitExt, GExtDat, PutData,
RemoveData, CloseDF, RemoveEl, CleanupRef, CleanupData, RemoveRF,
RemoveDF;

TYPE database;
    FilePt = POINTER TO File;

VAR DB: database;
(* Basic facts base buffer. *)

PROCEDURE IniDB(VAR Buf: database);
(* Initialises buffer Buf.*)

(* Procedures for creation,
   updating and fetching of reference file and reference file elements: *)

PROCEDURE CreateRF;
(* Creates new reference file, with first 2 words = 0. *)

PROCEDURE RemoveRF;
(* Removes reference and location files ("Referenc.FDB" and "Location.FDB"). *)

PROCEDURE NewElem(Buf: database; VAR el: elid; rf, rb: CARDINAL);
(* Creates a new element, with system-assigned identification el, in database.
   Procedure to be used as an alternative to GetRef.*)

PROCEDURE GetRef(Buf: database; el: elid);
(* Loads record from reference file into buffer.
   Precondition: Record el does exist.
   If the designated record was already present, no operation is performed,
   so possible unsaved updates are conserved.*)

PROCEDURE GPntFP(Buf: database; m: CARDINAL; VAR Ref: elid);
(* Get forward pointer #m.*)

PROCEDURE GPbList(Buf: database; m: CARDINAL; VAR Ref: elid);
(* Obtain pointer to first element of inverse list.*)

PROCEDURE GPntLP(Buf: database; m: CARDINAL; VAR Ref: elid);
(* Ref will indicate next element of inverse list.*)

PROCEDURE UpdateFL(Buf: database;
    el1: elid; rf: CARDINAL; el2: elid; rb: CARDINAL);
(* Adds new link (Function element) from el1 to el2,
in such a way that all pointers are consistent and the order is preserved.
   Returns with el1 in buffer.
   Precondition: If element el1 is to have any value this should be added
   first (using NewData).*)

PROCEDURE RemoveFL(Buf: database; el: elid; rf, rb: CARDINAL);
(* Cut link from el (function #rf). All relevant pointers are updated.
   Returns with el in buffer.*)

PROCEDURE PutRef(Buf: database);
(* Adds reference record to file. If length has been changed, to new location.*)
The PDB package

PROCEDURE CloseRF;
  (* Closes reference file. *)

  (* The following are procedures for creation, 
     updating and fetching of data elements or extensions: *)
PROCEDURE CreateDF;
  (* Creates new data file, with first 2 words = 0. *)
PROCEDURE RemoveDF;
  (* Removes existing data file ("Objects.FDB"). *)
PROCEDURE NewData(Buf: database; el: elid; V: tvalue);
  (* Creates new data record and assigns V to it. *)
PROCEDURE GetData(Buf: database; el: elid); 
  (* Loads dynamic buffer from reference file. 
     If the designated item already occupies the data buffer, no operation 
     is performed, so possible unsaved updates are conserved. *)
PROCEDURE GInfVal(Buf: database; VAR V: tvalue);
  (* V will represent data record (= search argument) of element in buffer. *)
PROCEDURE PExtDat(Buf: database; L: CARDINAL; EP: ADDRESS); 
  (* Add extension data to the data buffer. 
     Performing Pextdat will destroy the original extension data in the 
     buffer, and subsequently in the database data if PutData is performed. *)
PROCEDURE HasExt(Buf: database): BOOLEAN;
  (* Shows whether item in buffer has extension data. *)
PROCEDURE AppendExt(VAR Str: ARRAY OF BYTE; L: CARDINAL); 
  (* Appends string Str to end of objects file. May be used to enter an 
     extension in several pieces (a number of AppendExt calls, followed 
     by CommitExt). A PExtDat call will destroy all non-committed 
     extensions. *)
PROCEDURE CommitExt(Buf: database); 
  (* Makes appended extension permanent and associates it with item in Buf. *)
PROCEDURE GExtDat(Buf: database; VAR L: CARDINAL; VAR EP: ADDRESS); 
  (* Makes available the extension data of buffer element. *)
PROCEDURE PutData(Buf: database); 
  (* Transfers buffer, including data on extensions to reference file. *)
PROCEDURE RemoveData(Buf: database; el: elid); 
  (* Removes data record from item. *)
PROCEDURE CloseDF;
  (* Closes data file. *)
PROCEDURE RemoveEl(Buf: database); 
  (* Removes element from database. 
     Precondition: all references associated with el are Null. *)

50 Pim Lemmens
PROCEDURE CleanupRef;
(* Garbage collection on reference file: one pass through the file. *)

PROCEDURE CleanupData;
(* Garbage collection on objects file: one pass through the file. *)

END ElStorage.

DEFINITION MODULE ElSets;
(* ELORADO set operations.  
   Made by: W.J.M. Lemmens, Eindhoven University of Technology (TUE).  
   Latest modification: '87-11-23.  
   May produce one of the following messages:  
   Eldorado error 200: Insert not allowed for ref sets.  
      201: Type of element to be inserted not compatible with set type.  
      202: Set to be valued is not a ref set.  
      203: Type of set operands in SetAdd not compatible.  
      204: ref Operands in SetAdd not of same category.  
      205: Both operands identical in SetAdd.  
      206: Type of set operands in SetRetain not compatible.  
      207: ref Operands in SetRetain not of same category.  
      208: Both operands identical in SetRetain.  
      209: Type of operands in SetRemove not compatible.  
      210: ref Operands in SetRemove not of same category.  
      211: Both operands identical in SetRemove.  
   Run-time error 212: No room in heap for new set element.  
*)

FROM EldAlg IMPORT elid, datatype, value, tvalue;

EXPORT QUALIFIED setel, setrec, elset, IniSet, CreaSet, Insert, 
RefInsert, SetExtract, Valuate, SetAdd, SetRetain, SetRemove, 
CopySet, EmptySet, SetMin, SetMax;

TYPE setel = POINTER TO settee;
  settee = RECORD
    V: value;
    Next: setel
  END;
  elset = RECORD
    CASE T: datatype OF
      empty, ghost: Cat: elid
    END;
    Elem: setel
  END;

PROCEDURE IniSet(VAR S: elset);
(* The very first operation to be performed, only once, on a set.  
   This should always be done by the user. *)

PROCEDURE CreaSet(VAR S: elset; T: datatype; Cat: elid);
(* Used for the creation of a new empty set in situations where the user needs
to build a set by adding element by element.
Releases memory currently occupied by set. *)

PROCEDURE Insert(VAR S: elset; V: tvalue);
(* This procedure adds an atom (tvalue) to the set. *)

PROCEDURE RefInsert(VAR S: elset; El: elid);
(* This procedure is only meant for internal use by ELDORADO. *)

PROCEDURE SetExtract(VAR S: elset; VAR V: tvalue);
(* This procedure removes an element from the set. *)

PROCEDURE Valuate(VAR S1: elset; S2: elset);
(* Produces a set of values from the database corresponding to the refs in S2. *)

PROCEDURE SetAdd(VAR S1: elset; S2: elset);
(* Union operation: the contents of S2 are added to S1.
Preconditions: S1 different from S2,
Type of S1 equal to type of S2 (or one or both are empty)
Categories of ref sets are equal.*)

PROCEDURE SetRetain(VAR S1: elset; S2: elset);
(* S1 := Intersection( S1, S2 )
Preconditions: S1 different from S2,
Type of S1 equal to type of S2 (or one or both are empty)
Categories of ref sets are equal.*)

PROCEDURE SetRemove(VAR S1: elset; S2: elset);
(* S1 := S1 \ S2
Preconditions: S1 different from S2,
Type of S1 equal to type of S2 (or one or both are empty)
Categories of ref sets are equal.*)

PROCEDURE CopySet(VAR S1: elset; S2: elset);
(* S1 := S2 *

PROCEDURE EmptySet(S: elset): BOOLEAN;
(* Checks whether set is empty. *)

PROCEDURE SetMin(VAR MV: tvalue; S: elset);
(* Yields the set minimum (if any). *)

PROCEDURE SetMax(VAR MV: tvalue; S: elset);
(* Produces the maximum value from the set S. *)

END ElSets.

DEFINITION MODULE ElAtoms;
(* Some operations on atom data.
Made by: W.J.M. Lemmens, Eindhoven University of Technology (TUE).
Latest modification: '87-08-25

May Produce the following messages:

Run-time error 800: Not enough memory available for extension data.

Pim Lemmens
FROM SYSTEM IMPORT ADDRESS;
FROM EldAlg IMPORT elid, tvalue;
FROM E1Storage IMPORT database;
FROM E1Sets IMPORT elset;

EXPORT QUALIFIED AtomVal, AtomExt, ExtAtom, AtomAppl, AtomInv;

PROCEDURE AtomVal(VAR V: tvalue; Buf: database; R: elid);
(* Produces value (if any) of item R from database Buf. *)

PROCEDURE AtomExt(VAR L: CARDINAL; VAR Ext: ADDRESS; Buf: database; R: elid);
(* Produces extension (length L, address Ext) of item R from Buf. *)

PROCEDURE ExtAtom(Buf: database; R: elid; L: CARDINAL; Ext: ADDRESS);
(* Extends item R from database Buf by data at address Ext of length L.
If this item did already have an extension, it will be lost. *)

PROCEDURE AtomAppl(VAR RR: elid; Buf: database; rf: CARDINAL; RD: elid);
(* Function application: RR := range value of function characterised by rf
for domain value RD. *)

PROCEDURE AtomInv(VAR SR: elset; Buf: database; CD: elid; rf, rb: CARDINAL; RR: elid);
(* Inverse function application. SD: result set, RR: range value. Function
characterised by labels rf and rb. *)

END E1Atoms.

DEFINITION MODULE E1Functions;
(* Handles database functions.
Made by: W.J.M. Lemmens, Eindhoven University of Technology.
Latest modification: '87-nov-23.

Messages:
ELDORADO error 401 (fatal): Input set not of ghost (= ref) type.
*)

FROM EldAlg IMPORT elid;
FROM E1Storage IMPORT database;
FROM E1Sets IMPORT elset;

EXPORT QUALIFIED Apply, InvAppl;

PROCEDURE Apply(VAR S2: elset; Buf: database; CR: elid; rf: CARDINAL;
S1: elset);
(* Function application of F (D(F)=S1.Cat, R(F) = CR, forw lbl = rf)
to S1, resulting in S2. *)

PROCEDURE InvAppl(VAR S2: elset; Buf: database; CD: elid; rf, rb: CARDINAL;
S1: elset);
(* Inverse function application of F (D(F) = CD, R(F) = S1.Cat, bkw lbl=rb)
to S1, resulting in S2. *)

Pim Lemmens
The PDB package

END ElFunctions.

C.2 Overview of special PDB modules.

The lower level PDB modules are UserIO, PDBStringBuffer and PDBConfig.

UserIO is used by Interact and PDBStringBuffer is used by PDBScanner.

PDBScanner is used by PDBObjects

C.2.2 PDB definition modules.

DEFINITION MODULE UserIO;
(* Procedures for user interaction.
This implementation is PC and ANSI specific.
Made by: WJM Lemmens, Eindhoven
Latest Modification: 901217

This module contains a number of keyboard and screen handling functions that allow the operation of several kinds of menus. The controlling program receives keyboard codes and string input from the keyboard by way of FromKeyb and as a result may send commands to the screen. There are two kinds of such commands: Simple screen operations like 'one line up' (KeyCom Up), 'one position to the right' (Right), etc. These commands are issued by way of DoDispl. They are supplemented by the procedures CLS, GotoXY and WriteUnsInt. The package also contains a line editor (LineEdit).
*)

EXPORT QUALIFIED KeyCom, CLS, DoDispl, GotoXY, WhereX, WhereY, LineEdit,
FromKeyb, ReadString, Edit, RVOn, RVOff, PutOnScreen,
WriteCard;

TYPE KeyCom = (NoCom, Enter, Up, Down, Right, Left, PgUp, PgDown,
Home, End, Ins, Del, Escape, F1, F2, F3, F4, F5, F6,
F7, F8, F9, F10);
(* Special keys from keyboard and special commands to display. *)

PROCEDURE CLS;
(* Clear screen. *)

PROCEDURE DoDispl(Com: KeyCom); (* Various screen actions. *)

PROCEDURE GotoXY(x, y: CARDINAL); (* Move to indicated spot on screen. *)

PROCEDURE WhereX(): CARDINAL; (* Supplies X position of cursor. *)

PROCEDURE WhereY(): CARDINAL; (* Supplies Y position of cursor. *)
PROCEDURE LineEdit(L: CARDINAL; VAR ItName: ARRAY OF CHAR):
   KeyCom;
(* Insert and edit line of text. *)

PROCEDURE FromKeyb(): KeyCom;
(* Input (special) keys. *)

PROCEDURE ReadString(VAR Str: ARRAY OF CHAR);
(* Input identifier from keyboard. *)

PROCEDURE Edit(VAR Str: ARRAY OF CHAR);
(* Edit string. *)

PROCEDURE RVOn(X0, Y0, XL, YL: CARDINAL);
(* Displays rectangular piece of screen in reverse video. *)

PROCEDURE RVOff(X0, Y0, XL, YL: CARDINAL);
(* Returns rectangular piece of screen to normal video. *)

PROCEDURE PutOnScreen(C: CHAR);
(* Puts C at cursor position. Cursor position and fg/bg colour are not changed.*)

PROCEDURE WriteCard(N: CARDINAL);
(* Writes number N. Uses just as much positions as needed. *)

END UserIO.

DEFINITION MODULE PDBStringBuffer;

FROM SYSTEM IMPORT BYTE;

EXPORT QUALIFIED StringBuf, IniSB, CopyBuf, InByte, OutByte, SBReset, SBErase, EOSB;

TYPE StringBuf;

PROCEDURE IniSB(VAR SB: StringBuf);
(* Creates new string buffer SB. *)

PROCEDURE CopyBuf(OutBuf: StringBuf; Inbuf: StringBuf);
(* Creates string buffer "OutBuf" with contents of "InBuf". *)

PROCEDURE InByte(SB: StringBuf; C: BYTE);
(* Put C in String Buffer. *)
The PDB package

PROCEDURE OutByte(SB: StringBuf; VAR C: BYTE);
(* Get next byte from string buffer. *)

PROCEDURE SBReset(SB: StringBuf);
(* Next OutByte will deliver first character of buffer. *)

PROCEDURE SBErase(SB: StringBuf);
(* Removes contents of string buffer.
Next InByte byte will be first in buffer. *)

PROCEDURE EOSB(SB: StringBuf): BOOLEAN;
(* "Last OutByte delivered last character of buffer SB
and no SBReset has been done on SB since." *)

END PDBStringBuffer.

DEFINITION MODULE PDBRefs;
(*
Hypertext reference file operations.
Latest modification: 900123.

A hypertext consists of any number of items. Every item has an
unique name and may have contents of some sort (e.g. a text, an
image or a program).
Items are accessible by name (operation FindName), through
keywords or through links from other items. For that purpose
every item may have a set of keywords that need, however, not be
exclusively associated to that item (other items may have the
same set). By specifying a set of keywords all items that have
all the keywords from the set may be found (operations FindKw and
KwToItems). The reverse, finding the keywords to a specific item,
is also possible (GetKw).
A link connects one item to an other. A link is uniquely
determined by a 'from' item, a 'to' item and a name. Link names
are not allowed to appear as item names. For a specific item all
links may be found that proceed from that item ('from' links
- operation GetFromLinks). Also all links may be found that lead
to that item ('to' links - GetToLinks).
Items are accessible through links in a number of ways:
1. All items that have a named link as a 'from' link
   (GetFromItems).
2. All items that have a named link as a 'to' link (GetToItems).
3. All items that are connected by a named link from a specific
   item (a subset of 2 - GetLinkFrom).
4. All items that are connected by a named link to a specific
   item (a subset of 1 - GetLinkTo).

Update Operations.

There is also a set of operations to change a hypertext. First
there is one to add a new item of a specific name (AddIt). To
such an item keywords may be associated (AddKw; KwRel) and
dissociated (RmvKw) and links may be added (AddLk; LkRel) and
removed (RmvLk). Link updates may only be performed on links

Pim Lemmens
that originate from the item specified. 'To' links may not be updated.
Of course, items may also be removed (RmvItem).

Except where explicitly stated differently, all operations use references (TYPE elid) as operands. So keywords, links and items are not referred to by name, but by some internal identification. These identifications may be found by the operations FindName (for items and links) or FindKw. If a certain name is not found among the keywords or among the item or link names, procedures RelatedKeywds or RelatedItems will find any soundalikes.
References are also produced by AddKw, AddLk and AddIt.

Often an operation yields a set (of items, links or keywords) as a result. These sets may further be manipulated by ELDORADO set procedures (see ELDORADO manual). Specifically, the names associated to a (ref-) set may be found by Valuate.

*)

FROM EldAlg IMPORT elid, identifier;
(* internal identification type, name type. *)
FROM ElSets IMPORT elset;

EXPORT QUALIFIED FindName, FindKw, IsItem, RelatedItems, RelatedKeywds,
AddIt, AddKw, KwlRel, AddLk, LkRel, RmvKw, RmvLk,
RmvItem, GetKw, Kwtoltems, GetFromItems, GetToItems,
GetFromLinks, GetToLkns, GetLinkFrom, GetLinkTo;

PROCEDURE FindName(VAR Ref: elid; Name: identifier);
(* Find (link or item) name in name table. 'Ref' will be Null if 'Name' not present.*)

PROCEDURE FindKw(VAR Ref: elid; Kw: identifier);
(* Find keyword in keyword table. *)

PROCEDURE IsItem(Ref: elid): BOOLEAN;
(* Determines whether Ref is item index. *)

PROCEDURE RelatedItems(VAR S: elset; Nm: identifier);
(* Finds set of names that sound more or less like Nm. *)

PROCEDURE RelatedKeywds(VAR S: elset; Kw: identifier);
(* Finds set of keywords that sound more or less like Kw. *)

PROCEDURE AddIt(VAR Ref: elid; Nm: identifier);
(* If item name Nm doesn't exist then enter it into name table. Deliver item ref. *)

PROCEDURE AddKw(VAR Ref: elid; Kw: identifier);
(* If Kw non-existent then enter it into DB. Deliver associated ref. *)

PROCEDURE KwlRel(R1, R2: elid);
(* Associate keyword R1 to item R2. *)
The PDB package

PROCEDURE AddLk(VAR Ref: elid; Nm: identifier);
(* If link name Nm doesn't exist then enter it into name table.
   Deliver link ref. *)

PROCEDURE LkRel(RB, RT, RL: elid);
(* Establish link from item RB to item RT through link RL. *)

PROCEDURE RmvKw(BaseRef, KwRef: elid);
(* If keyword is present and connected to item then remove connection between
   item and keyword.
   If keyword does not refer to an item anymore then remove keyword. *)

PROCEDURE RmvLk(BaseRef, LkRef, TgtRef: elid);
(* Remove link LkRef from item BaseRef.
   If link name not in use anymore, remove it. *)

PROCEDURE RmvItem(VAR Ref: elid);
(* Removes item plus associated keywords and links from DB.
   If item still has 'to'-links, no action is performed.
   Ref = Null if successful, Ref is unchanged otherwise. *)

PROCEDURE GetKw(VAR KwSet: elset; Ref: elid);
(* Produces set of keywords (refs) of item Ref. *)

PROCEDURE KwToItems(VAR IS, KS: elset);
(* Finds all items that have all keywords from KS as a keyword.
   Keywords are represented as refs, so first do FindKw and RefInsert
   if necessary. *)

PROCEDURE GetFromItems(VAR ItemSet: elset; LkRef: elid);
(* Produces set of items that have LkRef as a 'from' link. *)

PROCEDURE GetToItems(VAR ItemSet: elset; LkRef: elid);
(* Produces set of items that have LkRef as a 'to' link. *)

PROCEDURE GetFromLinks(VAR LkSet: elset; BaseRef: elid);
(* Produces set of fromlinks of item BaseRef. *)

PROCEDURE GetToLinks(VAR LkSet: elset; BaseRef: elid);
(* Produces set of tolinks of item BaseRef. *)

PROCEDURE GetLinkFrom(VAR SR: elset; BaseRef, LkRef: elid);
(* Finds set of items with link to item BaseRef through link id LkRef. *)

PROCEDURE GetLinkTo(VAR SR: elset; BaseRef, LkRef: elid);
(* Finds set of items with link from item BaseRef through link id LkRef. *)

END PDBRefs.

DEFINITION MODULE Interact;
(* Procedures for user interaction.
   Made by: WJM Lemmens, Eindhoven
   Latest Modification: 901217 *)
The PDB package

FROM Eisets IMPORT elset;
FROM EldAlg IMPORT tvalue;
FROM UserIO IMPORT KeyCom;

EXPORT QUALIFIED Menu, Display, DoMenu, SetModify, SetEdit,
     Choice;

TYPE Menu;

PROCEDURE Display(VAR S: elset; Vertical: BOOLEAN;
                    XT, YT, XW, YW: CARDINAL);
     (* Displays set S in menu format. *)

PROCEDURE DoMenu(VAR S: elset;
                 Vertical, AddIt, SingleCh, Preselect: BOOLEAN;
                 XT, YT, XW, YW: CARDINAL): KeyCom;
     (* Executes menu. 'Vertical' indicates whether the menu should be
        presented vertically.
        'AddIt' indicates whether items may be added,
        'Single' whether the choice should exist of one item only,
        'Preselect' whether S will be the default selection
        (Users may however add or remove items from the original selection).
        When return value is 'complete', the menu is executed properly,
        when 'unfinished', no choice was made.
        'failed' indicates something went wrong.
     *)

PROCEDURE SetModify(VAR S: elset;
                     Vertical, SelAll: BOOLEAN;
                     Header: ARRAY OF CHAR);
     (* Interactive updating of set S.
        SelAll initiates all elements as selected. *)

PROCEDURE SetEdit(VAR AS, RS, IS: elset;
                   Vertical: BOOLEAN;
                   Header: ARRAY OF CHAR);
     (* Allows modification of a set;
        delivers added (AS) and removed (RS) elements. *)

PROCEDURE Choice(VAR V: tvalue; VAR S: elset;
                 Vertical: BOOLEAN;
                 Header: ARRAY OF CHAR);
     (* Interactively choose an element from a set of refs by
        indicating its value. *)

END Interact.

DEFINITION MODULE PDBScanner;

(* Procedures for copying and interpreting files of objects format.
   Procedures may be used for backup and restore of hypertexts, for
   updates from a flat file and for output of one-dimensional texts.

Latest update: 900912.

59 Pim Lemmens
In error messages from interpreters that use this package, sometimes symbol numbers occur. These are:

0: '(['
1: '])'
2: 'l'
3: string of any length
4: end of file.

*)

FROM SYSTEM IMPORT ADDRESS;
FROM EldAlg IMPORT identifier;
FROM PDBStringBuffer IMPORT StringBuf;

EXPORT QUALIFIED Symbol, Sym, GetSym, Input, Skip, IdSym, InError,
Move2Id, Move2CBuf, ScanFile, FileScan, ScanMemory,
PutStr;

TYPE Symbol = (OpenBrace, CloseBrace, Separation, StrSym, EndSym);

VAR Sym: Symbol;

PROCEDURE GetSym;
    (* Lexical analyser. *)

PROCEDURE Input(S: Symbol);
    (* Verifies whether symbol is as expected. If not, no action is performed.
     * Will lift interpreter "back in the rails" if necessary and possible.*)

PROCEDURE Skip;
    (* Skips current component. *)

PROCEDURE IdSym(Id: ARRAY OF CHAR): BOOLEAN;
    (* Checks whether most recently read string is equal to Id. *)

PROCEDURE InError(EN: CARDINAL);
    (* Displays interpreter error message. *)

PROCEDURE Move2Id(V AR Str: identifier);
    (* Copy string from stringbuf to identifier Str. *)

PROCEDURE Move2CBuf(CB: StringBuf);
    (* Copy contents from string buffer to contents buffer. *)

PROCEDURE ScanFile(FileName: identifier): BOOLEAN;
    (* Start of interpretation of file 'FileName'. *)

PROCEDURE FileScan;
    (* Continue (after ScanMemory) scan of file, previously opened by
     * ScanFile. *)

PROCEDURE ScanMemory(EA: ADDRESS; L: CARDINAL);
    (* Start of interpretation of code in RAM. *)

Pim Lemmens
PROCEDURE PutStr(S: ARRAY OF CHAR);
(* Not really a scanner procedure: Outputs S to extension file.
   '\n' is interpreted as CR/LF *)

END PDBScanner.

DEFINITION MODULE PDBConfig;

(* PDB configuration manager. Allows specification of handlers for
the contents of different kinds of objects.

Latest update: 901008.

Each kind of object may be processed by a different set of
programs like word processors or paint programs. This module
offers means of defining objects kinds by specifying
interactively kind name plus display handler plus
modification handler for each kind. It also has possibilities
for input of configuration data through configuration files.

Error messages:

101: Illegal keyword in configuration specification.
102: Closing text not 'Config'.
103: Symbol is not a keyword (such as 'Config', 'Comment', 'Object').
104: (warning) No configuration in extension file.
*)

FROM EldAlg IMPORT identifier;

EXPORT QUALIFIED ResId, ConfResult, AddConf, RmvConf, InputConf,
    ConfMenu, DisplayProc, ModifyProc;

TYPE ResId = (success, full, absent);

VAR ConfResult: ResId (* Result of last config operation *);

PROCEDURE AddConf(KId, DisplId, ModId: identifier);
(* Add kind 'KId' with display program 'DisplId' and modification
program 'ModId'. *)

PROCEDURE RmvConf(KId: identifier);
(* Removes kind 'KId' from configuration table. *)

PROCEDURE InputConf;
(* Reads configurations from current input medium (viz. PDBObjec.HtxtIn).*)

PROCEDURE ConfMenu;
(* Interactive update of configuration table. *)

PROCEDURE DisplayProc(V AR DP: identifier; KId: identifier);
(* Shows display processor of 'KId'. * )
PROCEDURE ModifyProc(VAR MP: identifier; KId: identifier);
(* Shows modification processor of 'KId'. *)

END PDBConfig.

DEFINITION MODULE PDBObjects;

(* Module for hypertext objects file operations.

Latest modification: 901217.

This module contains an interpreter for hypertext specifications
and procedures for appending to and updating of the PDB objects
file. This file will itself contain a description of the hypertext
in question in interpretable form.

The interpreter is based on a syntax that facilitates partial
interpretation, such that different program parts may interprete
different parts of the specification. So there may be several
interpreters, each with its own goal. The only part of the syntax
that is common to all interpreters is the interpunction: The
symbols '{' and '}' mark, respectively, the beginning and the end
of a (possibly nested) block of specifications, and '|' marks the
separation of two parts.

Possible error messages from the interpreter:

1: Illegal keyword in contents specification
   ("Kind" or "Value" expected).
2: String expected in simple set.
3: Closing string not "Targets" (as it should be).
4: Illegal keyword in link specification.
5: '{' expected in link specification.
6: Closing string not "Links".
7: Closing string not "Value".
8: Closing string not "Contents".
9: Illegal keyword in object specification.
10: '}' or '|' expected in specification file.
11: Closing string not "Object".
12: System error - notify system developer.
13: Closing string not "Comment".
14: Illegal keyword in hypertext specification.
15: Illegal symbol in hypertext specification.

*)

FROM EldAlg IMPORT identifier, elid;
FROM ElSets IMPORT elset;

EXPORT QUALIFIED GetRefdObj, GetNamedObj, CurrObjName, CurrObjRef,
   HtxtIn, GetConts, PutConts, ShowKwds, ShowLinks,
   ShowTargets, Show, KwdsIn, LkIn, FixObj;
The PDB package

PROCEDURE GetRefdObj(ItRef: elid);
(* Object identified by Ref gets current object. *)

PROCEDURE GetNamedObj(Id: identifier);
(* Object 'Id' gets current object.
  If 'Id' does not exist it is created. *)

PROCEDURE CurrObjName(VAR CName: identifier);
(* Shows current object name. *)

PROCEDURE CurrObjRef(VAR CRef: elid);
(* Delivers reference (internal id) of current object. *)

PROCEDURE HtxtIn(FileName: identifier);
(* Update current hypertext by specification from 'FileName'.
  Existing objects with the same name as objects from the
  specification are overwritten. *)

PROCEDURE GetConts(Kind: identifier; FileName: identifier);
(* Add contents from 'FileName' to current object as kind 'Kind'. Any
  meta-characters in the file will automatically be escaped.
  Existing contents will be lost. *)

PROCEDURE PutConts(VAR Kind: identifier; FileName: identifier);
(* Put contents of current object in file 'FileName', skipping any
  escape characters. 'Kind' will be set to the proper value. *)

PROCEDURE ShowKwds(VAR KS: elset);
(* Presents keywords (strings) of current object. *)

PROCEDURE ShowLinks(VAR LS: elset);
(* Presents link names associated with current object. *)

PROCEDURE ShowTargets(Lk: identifier; VAR TS: elset);
(* Presents targets of link Lk of current object. *)

PROCEDURE Show(VAR KindId: identifier);
(* Display the contents of current object on the video screen. *)

PROCEDURE KwdsIn(KS: elset);
(* Enters keywords KS in current object buffer. *)

PROCEDURE LkIn(LN: identifier; Tg: elset);
(* Updates current object with link LN having targets Tg. *)

PROCEDURE FixObj;
(* Finalise update: Add all data of current object to the objects
  and reference files. Without FixObj call there will be no
  persistent updates of keywords and contents. *)

END PDBObjects.
In this series appeared:

89/1 E.Zs.Lepoeter-Molnar
Reconstruction of a 3-D surface from its normal vectors.

89/2 R.H. Mak
P.Struik

89/3 H.M.M. Ten Eikelder
C. Hemerik
Some category theoretical properties related to a model for a polymorphic lambda-calculus.

89/4 J.Zwiers
W.P. de Roever
Compositionality and modularity in process specification and design: A trace-state based approach.

89/5 Wei Chen
T.Verhoeff
J.T.Udding

89/6 T.Verhoeff
Networks of Communicating Processes and their (De-)Composition.

89/7 P.Struik
A systematic design of a parallel program for Dirichlet convolution.

89/8 E.H.L.Aarts
A.E.Eiben
K.M. van Hee
A general theory of genetic algorithms.

89/9 K.M. van Hee
P.M.P. Rambags
Discrete event systems: Dynamic versus static topology.

89/10 S.Ramesh
A new efficient implementation of CSP with output guards.

89/11 S.Ramesh
Algebraic specification and implementation of infinite processes.

89/12 A.T.M.Aerts
K.M. van Hee
A concise formal framework for data modeling.

89/13 A.T.M.Aerts
K.M. van Hee
M.W.H. Hesen
A program generator for simulated annealing problems.

89/14 H.C.Haesen
ELDA, data manipulatie taal.

89/15 J.S.C.P. van der Woude
Optimal segmentations.

89/16 A.T.M.Aerts
K.M. van Hee
Towards a framework for comparing data models.

89/17 M.J. van Diepen
K.M. van Hee
A formal semantics for Z and the link between Z and the relational algebra.
Formal methods and tools for the development of distributed and real time systems, p. 17.

Dynamic process creation in high-level Petri nets, pp. 19.

Foundations of Compositional Program Refinement - safety properties -, p. 38.

Decomposition of delay-insensitive circuits, p. 25.

On the delay-sensitivity of gate networks, p. 23.


A logic for one-pass, one-attributed grammars, p. 14.

Receptive Process Theory, p. 16.

Combining the functional and the relational model, p. 15.

A formal semantics for Z and the link between Z and the relational algebra, p. 30. (Revised version of CSNotes 89/17).

A proof system for process creation, p. 84.

A proof theory for a sequential version of POOL, p. 110.

Proving termination of Parallel Programs, p. 7.

A proof system for the language POOL, p. 70.

Compositionality in the temporal logic of concurrent systems, p. 17.

A fully abstract model for concurrent logic languages, p. 23.

On the asynchronous nature of communication in logic languages: a fully abstract model based on sequences, p. 29.
<table>
<thead>
<tr>
<th>Date</th>
<th>Authors</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>90/19</td>
<td>M.M. de Brouwer, P.A.C. Verkoulen</td>
<td>Two Case Studies in ExSpect</td>
<td>24</td>
</tr>
<tr>
<td>90/20</td>
<td>M. Rem</td>
<td>The Nature of Delay-Insensitive Computing</td>
<td>18</td>
</tr>
<tr>
<td>90/21</td>
<td>K.M. van Hee, P.A.C. Verkoulen</td>
<td>Data, Process and Behaviour Modelling in an integrated specification framework</td>
<td>37</td>
</tr>
<tr>
<td>91/01</td>
<td>D. Alstein</td>
<td>Dynamic Reconfiguration in Distributed Hard Real-Time Systems</td>
<td>14</td>
</tr>
<tr>
<td>91/02</td>
<td>R.P. Nederpelt, H.C.M. de Swart</td>
<td>Implication. A survey of the different logical analyses &quot;if...,then...&quot;</td>
<td>26</td>
</tr>
<tr>
<td>91/03</td>
<td>J.P. Katoen, L.A.M. Schoenmakers</td>
<td>Parallel Programs for the Recognition of $P$-invariant Segments</td>
<td>16</td>
</tr>
<tr>
<td>91/05</td>
<td>D. de Reus</td>
<td>An Implementation Model for GOOD</td>
<td>18</td>
</tr>
<tr>
<td>91/06</td>
<td>K.M. van Hee</td>
<td>SPECIFICATIEMETHODEN, een overzicht</td>
<td>20</td>
</tr>
<tr>
<td>91/07</td>
<td>E. Poll</td>
<td>CPO-models for second order lambda calculus with recursive types and subtyping</td>
<td></td>
</tr>
<tr>
<td>91/08</td>
<td>H. Schepers</td>
<td>Terminology and Paradigms for Fault Tolerance</td>
<td>25</td>
</tr>
<tr>
<td>91/09</td>
<td>W.M.P. v.d. Aalst</td>
<td>Interval Timed Petri Nets and their analysis</td>
<td>53</td>
</tr>
<tr>
<td>91/10</td>
<td>R.C. Backhouse, P.J. de Bruin, P. Hoogendijk</td>
<td>POLYNOMIAL RELATORS, p. 52</td>
<td></td>
</tr>
<tr>
<td>91/11</td>
<td>R.C. Backhouse, P.J. de Bruin, G. Malcolm</td>
<td>Relational Catamorphism, p. 31</td>
<td></td>
</tr>
<tr>
<td></td>
<td>E. Voermans, J. van der Woude</td>
<td></td>
<td></td>
</tr>
<tr>
<td>91/12</td>
<td>E. van der Sluis</td>
<td>A parallel local search algorithm for the travelling salesman problem</td>
<td>12</td>
</tr>
<tr>
<td>91/13</td>
<td>F. Rietman</td>
<td>A note on Extensionality</td>
<td>21</td>
</tr>
<tr>
<td>91/14</td>
<td>P. Lemmens</td>
<td>The PDB Hypermedia Package. Why and how it was built</td>
<td>63</td>
</tr>
</tbody>
</table>