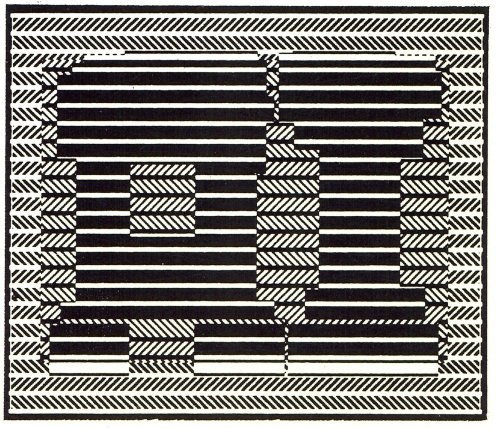
VRIJE UNIVERSITEIT BRUSSEL

ARTIFICIAL INTELLIGENCE LABORATORY

Kris Van Marcke

THE USE AND IMPLEMENTATION OF THE REPRESENTATION LANGUAGE KRS

technical report 88_2



Kris Van Marcke

THE USE AND IMPLEMENTATION OF THE REPRESENTATION LANGUAGE KRS

technical report 88_2

Vrije Universiteit Brussel
Pleinlaan 2
1050 Brussels

THE USE AND IMPLEMENTATION OF THE REPRESENTATION LANGUAGE KRS

Kris Van Marcke

Promotor: Prof. L. Steels
Datum: april 1988

Proefschrift ingediend tot het
bekomen van het diploma van
Doctor in de Wetenschappen

Abstract.

This report is about *Artificial Intelligence programming.* It discusses the representation language KRS, which combines innovative techniques for representation and computation with an object oriented programming style. The report explains the implementation of KRS and the techniques which were developed for this implementation. Some of these techniques are extrapolated to a broader application domain. The report supports KRS usage by means of a short tutorial, a reference manual and several large examples illustrating specific KRS programming styles, for example for data-modeling, data-driven programming and knowledge-based programming.

Keywords:

Artificial Intelligence, Artificial Intelligence Programming, Knowledge Representation, KRS, Representation Languages, Object-Oriented Languages.

for my father

PREFACE.

The KRS project started back in 1983 based on a proposal by Luc Steels. An object-oriented message-passing language was proposed as a good basis for knowledge representation. This language, currently referred to as KRS-83, is defined in [Steels84].

Many of the concepts introduced or applied in KRS-83 namely, *multiple formalisms, open-endedness, object-orientedness* and *message-passing*, are currently quite popular in representation tool system technology. KRS-83 is thus in a way an ancestor of systems like KEE [Intellicorp84] and ART [Clayton85].

KRS-83 was implemented in ZETALISP making use of FLAVORS [Weinreb81]. It was used in several research projects, the largest of which is described by [Zeigler86]. The KRS-83 team included, besides Luc Steels and myself, also Peter Van Damme and Eric Van Releghem.

Around newyear 1985, several fundamental shortcomings of the KRS-83 architecture led to the development of a new KRS. First of all, the system was not completely open-ended in the way it was foreseen. Extensions were possible but not without quite a bit of low level FLAVOR programming. The second major problem was the inflexibility of the class-based inheritance architecture. It became clear that, especially for A.I. programming, a more flexible architecture was required. These issues are by now more widely recognized (see e.g. [Lieberman86]).

KRS was rebuilt from scratch in February 1985, keeping nothing of its ancestor language KRS-83 except for the original objectives and the experience of what went wrong. To solve the open-endedness problem, the new version found inspiration in representation language language technology [Greiner80, Haase86], which introduced the idea to define the components of a language in terms of itself. KRS also adopted the more flexible prototype-based inheritance mechanism [Lieberman86].

The actual design of KRS involved a very tied collaboration between Luc Steels and myself for several months, during which many partial implementations were made to evaluate our ideas. Viviane Jonckers, Pattie Maes, Walter Van de Velde and several external visitors such as Mark Eisenstadt, Ernst Schatz, Michel Tueni, and Stefano

Cerri gave many constructive comments.

By the end of 1985, after a visit to Michel Tueni in Paris, I designed the KRS interface. Katrien Lesage, Mieke de Winter and in a later stage also Peter Strickx made substantial contributions to its implementation.

KRS-83 developments were supported by Esprit Pilot Project no. 82 (Intelligent WorkStation). The KRS kernel was developed within Esprit-1 project no. 440 (Message Passing Architectures and Description Systems). The development of KRS libraries and partly of the interface have been funded by Esprit-1 project no. 82 (the continuation of the pilot project mentioned above) [Van Marcke87b].

KRS is currently being used in many research and development activities. The latter include include development of expert systems for technical diagnosis and for automatic traffic control. KRS is also used for simulation and for office modeling.

Within our laboratory, KRS is used as representation language in all research projects spanning the following areas: second generation expert systems [Steels85], reflection [Maes87], learning and problem solving [Van de Velde88], conceptual programming [Jonckers87] and natural language processing [Daelemans87].

The first aim of this work is to communicate the KRS know-how which exists in our laboratory. It also serves as an illustration of the different programming styles KRS usage has led to. This way it supports current and future KRS users.

The second contribution of this work is the description of the KRS implementation. Our motivations for presenting a detailed account of this implementation vary. First of all, the basic KRS mechanisms are not the result of one sudden bright idea. They are the result of a long history of subsequent ideas and experimental implementations. This way, they grew together towards an adequate design and an operational system. As such, the implementation is not just an implementation of a language-design. It is the tool by which the language was constructed and a proof of the integrity of the resulting design.

Another aspect is the existing interest in the current and previous KRS implementations. Several people spread out over Europe have, sometimes to our surprise, studied in great detail the subsequent KRS implementations. We witnessed two recurring motivations for this. Some people needed to understand low level KRS implementation aspects in order to translate KRS to the LISP-dialects they use. This way, multiple versions were ported to LE-LISP, INTERLISP and COMMONLISP. Other people studied the implementation to acquire a better understanding of the KRS mechanisms. People outside our laboratory have indeed been using KRS in research-projects and doctoral theses, without direct support from those who invented KRS and before adequate documentation was written.

Finally, we think that the KRS implementation contributes to the current technology of implementing object-oriented systems, programming languages and interpreters.

ACKNOWLEDGEMENT

All the following people contributed in many ways to the realization of this work.

My supervisor Luc Steels introduced me to Artificial Intelligence and Knowledge Representation. He spent lots of his time in creating and maintaining a research laboratory where it is fun to work. Moreover, I had the privelege of cooperating with Luc very intensively. During this period, I learned what it means doing experimental research, an experience which still influences my way of thinking. Luc also had an important impact on the structure, format and contents of this work.

Many (day and night hour) discussions with Walter Van de Velde have had a great impact on all levels of this work, including the KRS design and implementation and the organization of this dissertation. Walter also proofread the whole of this text, parts of it many times.

Pattie Maes and Viviane Jonckers have always been open to discuss problems of any kind. In the early KRS days, they actively participated in many KRS discussions, via which they are co-responsible for its current shape. Pattie also proofread parts of this text and some related papers.

Several people have been helpful with the implementation of the KRS kernel and (especially) its interface: Filip Rademakers, Katrien Lesage, Mieke de Winter, Peter Strickx and Eric Wybouw.

Walter Daelemans and Peter Strickx also proofread parts of this text.

All other members of the AI-Lab play each an important role in maintaining an optimal environment for scientific work. I want to mention especially Gina and Karina. All of the problems they daily solve(d) directly enable our working under optimal conditions.

Two people who have been following the KRS developments from the very beginning are Ernst Schatz (Siemens AG, Germany) and Michel Tueni (Bull Transac, France). Many KRS design and implementation discussions with them have broadened my view.

Ken Haase and Stefano Cerri have, during the periods they visited our laboratory, always been open for discussions. Ken also read some preliminary parts of this text.

I am very grateful to Karina and Inke, for their love and understanding. I finally want to thank my family, for their support (of different kinds) and their interest.

This research has been supported by Esprit projects P440 "Message Passing Architectures and Description Systems" and P82 : "Intelligent Work-Station".

TABLE OF CONTENTS.

INTRODUCTION.

1. Domain.

Artificial Intelligence (A.I.) is an experimental science. A.I. problems are mostly ill-defined and proposed theories are often too complex to be actually verified. What remains is the practice of A.I., which is writing programs intended to reflect our theories. If the program works, we gain an understanding of how the theory works. If the program fails (or if we fail to program the theory), then we learn about the problems with the theory.

A programming environment is thus the A.I. researcher's workbench. It poses more or less the same requirements as for traditional programming, but with an special emphasis on flexibility and powerful control structures. These are certainly also important in traditional programming, but are there overshadowed by concerns such as efficiency, modularity, maintainability and portability. We think the following two reasons cause this different emphasis:

- Neither the architecture nor the behavior of an A.I. program are outlined in advance. The program is set up as an experiment from which one hopes to learn both the program's architecture and its ultimate behavior.
- The program's behavior is not fixed. A.I. fields such as machine learning and computational reflection study programs whose behavior evolves during execution time.

Nevertheless, *A.I. programming technology* has been a fruitful source of inspiration to computer science. Languages like LOGO, LISP and PROLOG, or programming methodologies like list processing, data-driven programming or data-level-parallelism have found response outside the A.I. community.

2. Possible Strategies.

A.I. programming tools are situated in four categories: A.I. programming languages, standalone knowledge representation systems, representation languages and representation tool systems.

A.I. PROGRAMMING LANGUAGES

An *A.I. programming language* is a full-fledged programming language which is developed with A.I. applications in mind. It is meant to be general purpose, high-level and efficient in order to confront traditional high level programming languages. In addition, it must emphasize flexibility and powerful control structures.

The traditional A.I. programming languages are LISP and PROLOG. Other examples are SCHEME [Sussman75, Rees86], OPS5 [Brownston85] and the ACTOR languages [Hewitt76]. More recently, the possibility to use parallelism has been investigated, leading to for example CONCURRENT PROLOG [Shapiro83], *LISP [Machines86] and CMLISP [Hillis85].

Obviously, work along this line is a necessity. New languages must be developed to make better use of existing technology. There are however many reasons not to come out with a new programming language every three years. People refuse to learn new programming languages over and over again, existing programming skills get lost, existing software is difficult to integrate with new software, etc. Moreover, new languages will not necessarily support better the tasks of the A.I. programmer. There is a broad spectrum of techniques the A.I. programmer wants to make use of. It is unfeasible and even undesirable to include this whole spectrum within a single programming language.

STANDALONE KNOWLEDGE REPRESENTATION SYSTEMS

A *standalone knowledge representation system* is a program dedicated to the development of knowledge-based systems. It predefines the formalism to be used. This is a combination of a representation framework and a reasoning strategy on top of this. It is a tool for a knowledge engineer who does not need to have programming experience. He just provides the knowledge the program requires in the appropriate format.

Currently commercialized knowledge representation systems include OMEGA [Hewitt80] and KL-ONE [Brachman85]. Other historically important systems are KRL [Bobrow77], NETL [Fahlman] and EMYCIN [Van Melle81].

The major advantage of a knowledge representation system is that it is immediately usable by novice users (experienced though in the problem domain). Using the system requires understanding the mechanism of the formalism on which it is based. It requires no additional programming besides encoding the knowledge to which the program must have access. An important drawback is that all formalisms developed thus far turned out to be only applicable within loose but rather small boundaries. Most knowledge representation researchers now agree that no general purpose formalism will ever be found.

REPRESENTATION LANGUAGES

A *representation language* is a library built on top of an existing A.I. programming language for which it provides facilities in any of the following areas: *representation*, *deduction* and *control*.

- Facilities in the *representation* area consist of data structures which allow the flexible representation of facts, based on a knowledge representation formalism. Among the best known formalisms are production rules, frames, and semantic networks.
- Addressing the *deduction* area leads to the development of deductive mechanisms. A deductive mechanism automatically operates on the knowledge at hand, either to retrieve new facts or to solve a particular problem.
- Effort in the *control* area involves increasing power and flexibility of control-structures. More complex control-structures are required to program more idiosyncratic behavior. At the same time, they must preserve program readability and maintainability. Those may possibly be based on good intuitions rather than on a profound understanding of the control-flow.

There is a large tradition of representation languages built on top of LISP. Historically important ones in this tradition are PLANNER [Sussman71], CONNIVER [Sussman72], AMORD [de Kleer77], LOOPS [Bobrow81] and FRL [Roberts77].

Representation languages on top of PROLOG are usually implemented via special meta-interpreters. Enhanced meta-interpreters can embed new control strategies [Gallaire82, Pereira82], extend the logic with new useful constructs [Furukawa84, Kauffmann86] or define analysis tools to provide typical expert systems [Bowen84, Sterling84].

The major advantage of a representation language is that it is applicable to a wide range of problems. An A.I. programming library anticipates the level problem, i.e. the gap between the relatively low level of programming languages and the high level of the tasks to be programmed. An important drawback is that, to effectively use the library, the programmer must have extensive knowledge of different sorts: (i) knowledge about the underlying programming language, (ii) knowledge about how the added features function, (iii) knowledge about how those features integrate with the underlying language and (iv) knowledge about when to use which of them. Therefore, unlike knowledge representation systems, representation languages can only be effectively used by experienced A.I. programmers.

REPRESENTATION TOOL SYSTEMS

A *representation tool system* is an integration of different packages, each of which contains an individual knowledge representation system. They are integrated such

that there can be transfer of knowledge between all packages. *Open-ended* knowledge representation tool systems can be augmented with new packages.

The representation tool system approach has been the most popular in the last decenium. The concept of an open-ended multiple formalism system originated around 1984 for example in [Steels84] and in [Ferber84]. Many representation tool systems have been developed since. The best known examples are KEE [Intellicorp84], ART [Clayton85], NEXPERT [Data86] and BABYLON [Di Primio85].

The fundamental motivation for combining multiple formalism is to broaden the range of applicability. Representation tool systems turn out to be useful for a large variety of A.I. applications. This explains the commercial success of systems like KEE.

There are however still several disadvantages. An obvious handicap is that to effectively use the system, considerable A.I. programming experience is required. The programmer must have detailed knowledge about the advantages and disadvantages of each of the included packages in order to be able to choose the appropriate one for his tasks. A second problem is that real open-ended systems do not yet exist. An experienced A.I. programmer however wants to be able to adapt his environment to his specific needs by adding new tools or by tailoring existing ones. This turns out to be hard and often impossible.

3. The KRS Strategy.

KRS is a representation language which provides a limited set of features to augment its implementation language LISP.

A first option KRS takes is not to hide the underlying language, but to augment it, thereby providing additional functionality for the LISP programmer. Four motivations underly this strategy.

- We do not want to reinvent the computational capacities of LISP. Instead we want features of both languages to collaborate. To the computational capacities of LISP, KRS adds better representation capacities and a better structural organization of programs.
- By preserving the underlying programming language, we validate existing programming skills. Experienced LISP programmers do not see their programming skill devaluate. Instead, LISP experience can be even better employed.
- KRS can be placed on top of different LISP implementations. When KRS is for example ported to a LE-LISP environment, the specific LE-LISP characteristics become available.
- Existing LISP software can be integrated in new KRS programs. Hence the KRS programmer has access to whatever the LISP world has access to. KRS applications have for example been written which used the Symbolics mail-system and

through this also the UNIX mail-system [Van Marcke87b].

There are two reasons to choose LISP as the underlying language. First, KRS is developed to better promote LISP both as an A.I. programming language and as a general purpose programming language. This follows from the LISP tradition in our laboratory but also from our proper conviction that LISP does not get the appreciation it deserves as a general purpose programming language. Second, KRS is built on top of LISP because of LISP's incrementality. The interactive nature of LISP fits best in the KRS design. In particular the representation axiom of KRS provides an ingenious and flexible interface to LISP (cfr. chapter one).

The KRS language provides the following facilities: (i) frame-based data structures, (ii) single inheritance, (iii) advanced scoping mechanism, (iv) algorithmic concepts, (v) lazy evaluation and (vi) data-dependencies.

This particular set of tools was chosen for the following reasons:

- The set of basic tools must be small, such that it can be completely captured by the user.
- It is not an ad hoc collection of tools. Each of them has a specific role and this role is essential for the functioning of the whole.
- The tools which KRS provides are complementary to the reasoning capacities of LISP. More specifically, they better support data modeling and data organization.
- KRS provides a minimal kernel on top of which additional tools can be easily modeled.
- KRS is designed to be a workbench for an A.I. professional. The tools KRS provides should not be trivial. They must give support where it is required, i.e. support which is not easy to build from scratch every time.

4. Evaluation and Continuation.

This work is a milestone for the ongoing knowledge representation activities within our laboratory. At this point, we think it is important to make the following observations.

- KRS has had several working implementations since the summer of 1985. Currently, implementations run with acceptable efficiency on a variety of machines, including Symbolics, Sun, Mac-II and Xerox. The KRS implementation has proven very modular and easily portable. Indeed, multiple KRS implementations have been ported to different LISP dialects (ZETALISP, COMMON-LISP, LE-LISP, INTERLISP) by people inside and outside our laboratory.
- The KRS mechanisms have proven their adequacy in a variety of projects in many European research laboratories. KRS is used in several domains including

office modeling, technical diagnosis, simulation, natural language, and music. KRS has also been used in many bachelor and doctoral theses.

- Components of the KRS implementation (such as FPPD) can be extracted from KRS and independently used for other purposes. With respect to the object-oriented programming community, the KRS solutions are relevant for ongoing debates such as on inheritance or scoping, and are occasionally ahead of current technology such as in the use of caching and data-dependencies.

This work is a milestone but not necessarily an endpoint. The work can be continued in several directions.

One problem with the current KRS environment is the lack of debugging and explanation facilities. For KRS to be appreciated when offered to a larger audience, it should be studied which aspects are difficult to understand and how KRS itself could support this understanding.

One of our research goals is to adapt KRS to better support the development of large A.I. programs. This consists of building in tools for knowledge acquisition and learning.

KRS facilities support knowledge-based programming at the symbolic level. Currently, A.I. brings to bear subsymbolic programming by tuning complex dynamical processes [Steels87]. It can be investigated whether KRS can provide facilities for activities of the kind and how they can be integrated with symbolic representations.

5. Overview.

This text consists of three chapters providing a manual, a description of the implementation and an extended illustration of the usage of KRS.

THE KRS MANUAL

Chapter one contains a short KRS tutorial (based on [Van Marcke88a]) and the KRS manual. The tutorial is very elementary and may be skipped by readers who know about KRS. A complete tutorial can be found in [Steels86]. The reference manual describes KRS release 3.0.

A KRS IMPLEMENTATION

Chapter two describes the implementation of KRS release 3.0. It is a full COMMONLISP implementation which runs on Symbolics, Sun and Mac-II.

KRS USAGE

Chapter three provides four extended examples of KRS usage. Each of them illustrates a specific KRS programming style. The first example illustrated direct data-modeling. There is a direct mapping between the objects in the problem domain and the corresponding KRS concepts. The second example illustrates the use of KRS to build more advanced A.I. programming tools. A classification

tree is modeled and applied in the office domain. Example three gives a new view on data-driven programming. It introduces demons, which are procedures to be triggered by the occurrence of a particular event. The example proposes a new mechanism to trigger demons based on the data-dependency network. Example four implements production rules. It illustrates the implementation of existing knowledge representation formalisms in KRS.

Chapter One

THE KRS MANUAL.

INTRODUCTION.

This manual describes the Symbolics™ Common LISP implementation of KRS release 3.0. KRS is a representation language developed at the laboratory for Artificial Intelligence of the "Vrije Universiteit Brussel". The manual assumes preliminary knowledge of Common LISP. It starts with a short KRS tutorial based on [Van Marcke88a].

A full KRS tutorial is given in [Steels86]. The Common Lisp standard is described in [Steele84]. For more information on the implementation of KRS, see chapter two. [Steels88] clarifies the *representation of meaning* idea which is at the basis of the KRS interpreter.

STRUCTURE

Section one gives a short KRS tutorial. It can be skipped by readers who are familiar with KRS.

Section two describes the KRS concept-graph. It discusses concepts, subjects, how concepts represent entities, how concepts are defined and how they are organized in inheritance hierarchies and retrieval networks.

Section three describes the KRS interpreter. It clarifies the role of lazy evaluation. It discusses how requests to the concept-graph can be formulated. Further, it elaborates on two inference mechanisms: referent computation and inheritance. Finally the role of meta-interpreters and the use from within LISP are explained.

Section four describes the KRS interface, as it is currently implemented on the Symbolics™ LISP-machine.

Section five elaborates on errors which can occur, and on the existing debugging mechanisms.

Section six describes the standard libraries.

CONVENTIONS

Descriptions of functions, macros, concepts and subjects use formats which are inspired by the LISP-machine manual formats:

Foo *x (y (a y)) z* *[concept]*

explanation ...

Describes the concept Foo. The optional subject descriptions x y and z describe those subjects which need to be specified when instances of the concept Foo are made. Sometimes the default description of a subject is added, e.g. the default description of the filler of the y-subject is "(a y)".

bar *?arg* *[subject of FOO]*

explanation ...

describes the bar-subject of Foo. The optional concept-variable "?arg" indicates that the subject has one argument named *?arg*.

foo *x y z* *[function]*

explanation ...

describes the function foo with its argument-list (x y z).

foo *x y z* *[macro]*

explanation ...

describes the macro foo with its argument-list (x y z).

The examples often use left and right arrows at the beginning of a line:

```
--> (foo ....)
<-- bar
```

A right arrow indicates that the form which follows is evaluated. A left arrow indicates that the value which follows is returned by the evaluation.

TABLE OF CONTENTS

1. A SHORT TUTORIAL.

1.1. Introduction.

KRS is a representation language supporting knowledge based programming. It provides facilities in three areas:

REPRESENTATION

The concept-graph is the primitive for representation in KRS. It is a large network, the nodes of which are called concepts, the links subjects. Each concept represents a particular entity in the representation domain.

The concept-graph is fully reflective. This means that each single part of the graph, i.e. each concept and each subject, is itself explicitly represented by a unique concept. All those reflective concepts are of course themselves nodes of the concept-graph. This results in a virtually infinite network.

The description of the concept-graph is supported by a concept-language. The concept-language has two important characteristics. First it is interpreted such that the concept-graph is constructed in a lazy way, i.e. only the parts of the graph which are actually visited are ever contructed. This lazy construction strategy has two major advantages. (i) It allows us to define vast libraries without having them consume all the memory, and (ii) circular parts of the concept-graph are easy described. One disadvantage is that it is difficult to have a global overview of all concepts which exist.

The second important characteristic of the concept-language is that it is lexically scoped. This gives all concepts described by the same description in the concept-language easy access to one another. This way, the descriptions define so the speek clusters within the concept-graph, i.e. clusters of concepts wich can easily refer to one another.

DEDUCTION

Deduction in KRS is based on two mechanisms: referent computation and inheritance.

Referent computation consists of the computation of a concept's referent out of its definition. It is defined by the representation axiom [Steels88] which is inspired by intensional logic [Carnap56]. The representation axiom states that a concept's referent is equal to the result of the evaluation of the referent of the concept's definition.

One immediately notices that the representation axiom is recursive. This allows referent computation to chain a few levels deep until it bottoms out on some cached referent. A cached referent is a referent who has been computed before and whose result was stored for later re-use.

Inheritance in KRS is the process of augmenting a concept's description by copying the subjects of the concept's type into the concept. A concept's type is also a concept. It is attached to the concept with a subject: the type-subject.

KRS inheritance is in essence a very simple single inheritance mechanism. It is prototype-based, meaning that a concept can inherit from any other concept.

The combination of inheritance with the lexical scope of descriptions leads to one of the most expressive constructs in the KRS language. Normally a description determines (via lexical scope) how the concepts it describes refer to one another. With inheritance however, two descriptions interfere. References which make use of the lexical scope in the parent description are when inherited interpreted such that they make sense in the inheriting context.

CONTROL

Control in KRS is mostly hierarchical. A request is a query for a particular concept in the concept-graph. When a computation sends a request, it waits until it receives a reply. Innovative in the control-flow is the caching mechanism. The system replies to requests from the user. While doing this, it minimizes the required computation by caching results where possible. Those cached results are used as long as they remain valid. They end being valid when something changes which was used for their computation. At that time they are automatically withdrawn by KRS's build-in consistency maintenance mechanism.

1.2. The KRS concept System.

1.2.1. Concepts and Subjects.

KRS is an object-oriented language for representation. All entities, relevant for either the applications or the KRS system itself are represented by means of *concepts*. Users communicate with their application by sending *requests* to concepts. Three example concepts are listed below.

```
<john>  <three>  <person>
```

Subjects are named unidirectional relations that associate a concept with other concepts. This way they attach all issues that rise about a topic to the corresponding concept. The *subject-name* is the name of the relation, and the *filler* is the concept the subject points to. The *owner* of a subject is the concept the subject belongs to.

The following description is called a *defconcept-description*. It defines a concept with name John and with three associated subjects "type", "wife" and "birthyear".

```
(defconcept JOHN
  (a person
    (wife mary)
    (birthyear [number 1961])))
```

The subjects "wife" and "birthyear" are defined explicitly, while the subject "type" is extracted from the first part of the concept-description: "a person". The owner of all three subjects is the concept John. The fillers of the subjects "type", "wife" and "birthyear" are the concept with name Person, the concept with name Mary and some concept representing the lisp-number 1961 respectively (figure 1). There is a variety of possibilities to describe a subject's filler. The filler of the wife-subject for example is described by a concept-name.

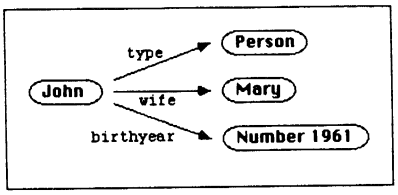


Fig 1: Concepts and Subjects.

We will refer to the filler of the wife-subject of the concept with name John with the more intuitive expression "the wife of John".

A request is lisp-form which returns the filler of a subject. The following requests can be sent to the concept John:

```
--> (>> type of John)
<-- <person>

--> (>> wife of John)
<-- <mary>

--> (>> birthyear of John)
<-- <number 1961>
```

Requests can be serially connected. The following two requests have the same result.

```
--> (>> type of (>> wife of John))
<-- <person>

--> (>> type wife of john)
<-- <person>
```

In the description of Mary below, the filler of the birthyear-subject is described using a request.

```
(defconcept MARY
  (a person
    (husband john)
    (birthyear (>> birthyear of john))))
```

Another way to describe a subject's filler is by a new concept-description in terms of type and subject-list, as illustrated with the father-subject of George.

```
(defconcept GEORGE
  (a person
    (birthyear [number 1965])
    (father (a person
              (birthyear [number 1945])))))
```

Hence, the request

```
(>> birthyear father of george)
```

returns the concept

```
<number 1945>
```

A request without subject-names is often used to access a concept given its name. For example the request below returns the concept with name John.

```
(>> of john)
```

1.2.2. Types and Inheritance.

Anticipating the single versus multiple-inheritance controversy in the object-oriented programming community, we chose to incorporate a simple single-inheritance mechanism, the main purpose of which is to obtain intuitive comprehensible descriptions. By making this choice, we deliberately give up particular facilities like inheriting from multiple orthogonal components. However, KRS shows (i) that carefully designed strict inheritance-hierarchies may contribute to the soundness of the representation, and (ii) that their expressive power comes close to and sometimes extends the power of bulky multiple-inheritance mechanisms [Van Marcke88b].

The KRS inheritance-hierarchy is determined by the type-subjects. This hierarchy is therefore also called the *type-tree*. A concept inherits all subjects from its type, except those it overrides. When the type is not explicitly defined it is added by the KRS parser. The default-type is the top of the type-tree Summum-Genus.

```
(defconcept COLOR)

-->  (>> type of color)
<--  <summum-genus>
```

If we describe the concept Person like below, the concepts John and Mary inherit the subject "father" from Person, but not the subject "birthyear".

```
(defconcept PERSON
  (birthyear (a number))
  (father (a person
            (adult-p true))))

-->  (>> birthyear of john)
<--  <number 1961>

-->  (>> father of john)
<--  <person #1>[1]
```

Notice that also the following requests can be correctly interpreted:

```
-->  (>> adult-p father of john)
<--  <true>

-->  (>> adult-p father father of john)
<--  <true>

-->  (>> adult-p father ... father of john)
<--  <true>
```

1.2.3. Relative-Requests and Lexical Scope.

When describing a generic method in an OOPL, i.e. a method which will be inherited or used by multiple objects, one must have a way to access the *client* of the operation. Traditionally, the client is the object which received the message. Sometimes, it is dynamically bound to the special variable SELF (see the Smalltalk blue book [Goldberg83].) That is why references to the client are also called self-references.

The KRS lexical scope mechanism decouples self-references from the object that received the message. The client or the context of a KRS request, is the concept being described by the global description the request is part of.

A *relative-request* is a request in which the last two elements (e.g. *of john*) are omited. The concept to which the request must be directed is dynamically determined using the lexical scope mechanism.

The birthyear-subject's filler in the changed version of the concept Mary below is defined by a relative-request. The birthyear of Mary is equal to the birthyear of the husband of some concept, which has to be pinned dynamically. This concept is called the *context* of the relative-request.

[1] The sharp-sign followed by a number is used in a concept's default pname if it is a concept with no name. The first part of such a pname is the name of one of the ancestors in the type-tree.

```
(defconcept MARY
  (a person
    (husband john)
    (birthyear (>> birthyear husband))))
```

The context of a relative-request is the *lexical scope* of the description it is part of. The lexical scope of a non-inherited description is the outermost description the current description is part of. In our example the description

```
(>> birthyear husband)
```

is part of the defconcept-description

```
(defconcept MARY ...)
```

and hence the lexical-scope is the concept Mary. Therefore if we ask for the birthyear of Mary, the context of the relative-request is this lexical scope, i.e. the concept Mary. Consequently, the birhtyear of the husband of Mary is used.

If a concept X inherits a description d from a concept Y, and Y is the lexical scope of the description d, then X becomes the lexical scope of the inherited description d. In the following example, the concept Anne inherits the birthyear-subject from Mary. Mary is the lexical scope of the description of the birthyear-subject, thus Anne becomes the lexical scope of the inherited birthyear-subject. Hence, when we ask for the birthyear of Anne, the context of the relative-request is the new lexical scope, i.e. the concept Anne. This way, the birthyear of the husband of Anne is taken.

```
(defconcept ANNE
  (a mary
    (husband george)))

-->  (>> birthyear of anne)
<--  <number 1965>
```

Lexical scope delimits clusters of concepts within the concept-graph, i.e. clusters of concepts with the same lexical scope. The lexical scope allows the concept to access its environment, i.e. concepts in the same cluster. The user can tune the size of the clusters by giving more or less nested descriptions. A cluster can describe a very complex or highly structured combination, or one small detail. Later we will see that a cluster can even be about one individual subject.

Analogous to requests without subject-name, a relative-request without subject-name denotes the current context. A relative-request without subject-name has the form:

```
(>>)
```

1.3. The Representation Axiom.

The meaning of a KRS concept, or what a concept is about, is explicitly represented within the system [Steels88]. Inspired by work in intensional logic [Carnap56], the representation of a concept's meaning consists of two parts: the *referent* and the *definition*. Those parts are connected with each other by means of the *representation axiom*.

1.3.1. Referents.

The referent of a concept is what a concept represents. For example the referent of the concept John is the person John, the referent of the concept Mary is the person Mary, the referent of the concept Person is a prototypical Person, etc. Some concepts may have a referent which is accessible within the machine we are working with. For example the concept Three, may have the lisp-number 3 as referent. Further, there can be a concept About-John, which has the concept John as referent. Such a concept could for example contain information about when the concept John was created, or on what file its description can be found, etc.

When the referent is accessible, it can be accessed with a request:

```
--> (>> referent of three)
<-- 3

--> (>> referent of about-john)
<-- <john>
```

This may give the impression that a referent is like an ordinary subject, which is only partially true. The most important difference between a referent and another subject is that a referent can never be inherited. When a referent of a concept is requested, and there is no explicit referent yet, it is not inherited but computed (see section 1.3.3).

Special-Instance-Descriptions are used to define a concept with a predefined LISP-referent. The concept-description

```
[number 1961]
```

defines a concept with type the concept Number and with referent the LISP-number 1961. Similarly the description

```
[form (+ 1 1)]
```

describes a concept with type Form and with referent the LISP-form (+ 1 1). Those concepts are called *data-concepts*.

1.3.2. Definitions.

The other aspect of representation is the definition of a concept. In contradiction with the referent, the definition of a concept is defined with an ordinary subject, which may be inherited. The *definition* describes the way to compute the concept's referent. Therefore, a definition must be a specialization of the concept Form or of one of Form's subtypes.

In the following defconcept-description we define an age-subject for the concept Person.

```
(defconcept PERSON
  (birthyear (a number))
  (age (a number
         (definition
           [form (- (>> referent of current-year)
                    (>> referent birthyear))]))))
```

We can now execute the following requests:

```
-->  (>> age of john)
<--  <number #2>

-->  (>> definition age of john)
<--  <form (- (>> referent of current-year) (>> referent birthyear))>

-->  (>> referent definition of john)
<--  (- (>> referent of current-year) (>> referent birthyear))
```

1.3.3. Referent Computation.

We already mentioned that when the unknown referent of a concept is requested, it is computed. To compute a concept's referent, we use the representation axiom:

> **Representation axiom:** *The referent of a concept C is the result of the evaluation of the referent of the definition of C.*

Thus, to compute a concept's referent, the KRS interpreter first tries to find the concept's definition (which may be inherited), then it takes the referent of that definition (which may require a recursive referent computation process) and finally this referent is evaluated. Hence:

```
(>> referent of C)  =  (>> eval referent definition of C)
```

So to compute the referent of the age of John, we get the following (assuming that the concept Current-Year has as referent the lisp-number 1987) (see also figure 2):

```
-->  (>> referent age of john)
=    (>> eval referent definition age of john)
=    (>> eval referent of <form (- ...)>)
=    (>> eval of (- (>> referent of current-year)
                    (>> referent birthyear)))
;;   Evaluation of (- (>> referent of current-year)(>> referent birthyear))
;;   with context <john>
<--  26
```

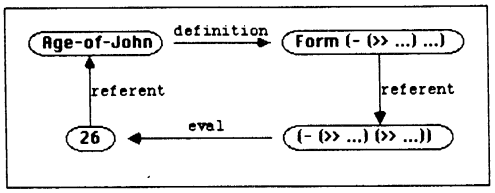


Fig 2: The representation axiom.

1.3.4. Caching and Consistency Maintenance.

Referents are computed the first time they are requested. After its computation, the referent is stored such that it can be re-used in the future. This technique is called caching. Caching brings an important speed-up, and allows to use meta-concepts (as will be shown further).

Caching is implemented on top of a data-dependency mechanism FPPD [Van Marcke86]. The data-dependencies are used to keep the KRS-world consistent. Whenever something changes, all cachings affected by that change are withdrawn. They will be properly recomputed the next time they are needed.

To illustrate this, we define the concept Current-Year as a counter.

```
(defconcept CURRENT-YEAR
  (a counter
    (initial-value [number 1987])))
```

A counter has a subject "increment" that can be used to increment the counter's referent. The concept Counter is described in detail in the manual (section 6.5.2).

Figure 3a shows the initial states of the concepts John and Mary. Local subjects which are marked :failed indicate that they have not been used yet.

Figure 3b shows the state of these concepts, after a request for the referent of the age of Mary. Mary's birthyear, Mary's husband and John's birthyear are asked for while computing Mary's age.

Figure 3c finally shows what happens after we increment Current-Year. The referent of the age of Mary has been withdrawn (pname has changed), because it depended on the former referent of Current-Year.

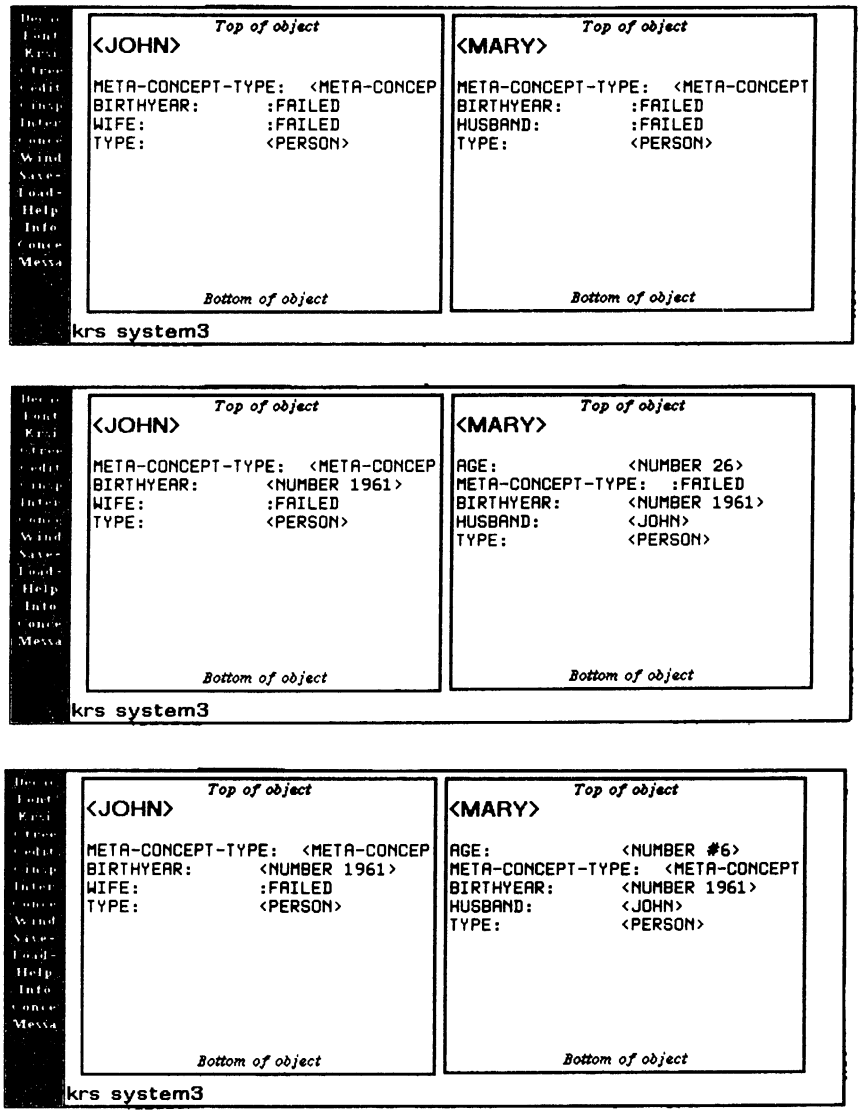


Fig 3: Consistency maintenance.

1.4. Meta-Concepts.

Meta-Concepts are concepts about other concepts. Or differently stated, a meta-concept is a concept with referent another concept. Meta-concepts are frequently used by the KRS-interpreter to make the KRS mechanism explicit. They are accessible and redefinable by the KRS user.

1.4.1. Subject-Concepts.

Subjects were defined before as named unidirectional relations. Internally those subjects are represented by KRS concepts, inheriting from the concept Subject. To facilitate the explanations, we will refer to those concepts as 'subject-concepts', while we

will keep using the term 'subject' like before.

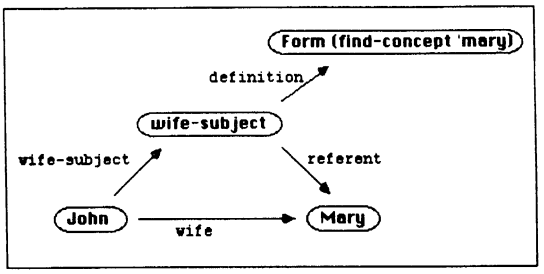


Fig 4: Subject-concepts.

The filler of a subject is the referent of the corresponding subject-concept. Like with ordinary concept's, this referent is computed the first time it is needed; in this case the first time the subjects filler is requested. The KRS language parser only stores the definition of the subject such that the referent can be properly computed the first time it is asked for. The wife-subject of John (figure 4) for example is defined by the KRS parser like:

```
(a subject
  (subject-name [symbol wife])
  (definition
    [form (find-concept 'mary)]))
```

Notice that subject-concept's have subject's of their own. This again results (virtually) in an infinite series of concepts.

Subject-concepts automatically provide a lazy interpreter. The subject-referents are computed the first time they are needed instead of when they are defined. A lazy interpreter gives all kinds of advantages. Circular structures for example can straightforwardly be defined. Take the descriptions of John and Mary, which contain cross pointing subjects "wife" and "husband":

```
(defconcept JOHN
  (a person
    (wife mary)
    (birthyear [number 1961])))
```

```
(defconcept MARY
  (a person
    (husband john)
    (birthyear (>> birthyear husband))))
```

With an eager interpreter, to create the concept John the concept Mary must be defined first and vice versa.

The fact that subject-relations make use of referent computation also allows us to benefit from the caching and consistency-maintenance mechanism. Once a relation is used, it is established via caching and it will remain so until the consistency-maintenance mechanism detects a possible inconsistency. Hence, the integrity of the KRS world is automatically preserved.

Explicit subject-concept's have the possibility of inheriting information from the

concept Subject. Further, the special symbol 'handler' in a request denotes that the subject-concept itself is needed, instead of the subject's filler:

```
-->  (>> handler age of john)
<--  <subject #3>

-->  (>> definition handler age of john)
<--  <form (a number ...)>

-->  (>> subject-name handler age of john)
<--  <symbol age>

-->  (>> subject-of handler age of john)
<--  <john>
```

A last important consequence is the ability to make use of the full referent computation mechanism to define subject-relations. For example, by defining the subject-concepts explicitly, arbitrary lisp-form can be used to compute a subject's filler. The adult-p-subject of Person illustrates this[2]:

```
(defconcept PERSON
  (birthyear (a number))
  (age (a number
         (definition
           [form (- (>> referent of current-year)
                    (>> referent birthyear))])))
  ((adult-p
     (a subject
       (definition
         [form (if (>= (>> referent age) 21)
                   (>> of true)
                   (>> of false))])))))
```

A subject-concept having no definition may inherit one. The concept Synonym-Subject described below, is a specialization of the concept Subject. It contains a definition which translates a request for the filler of a subject of this kind to a request for the filler of another subject.

```
(defconcept SYNONYM-SUBJECT
  (a subject
    (synonym-for (a symbol))
    (definition
      (a form
        (definition
           [form '(>> ,(>> referent synonym-for) of ,(>> subject-of))])))))
```

We can now explicitly define the partner-subject-concept of Person as a synonym-subject. Its synonym-for inherits from Symbol and has as referent either the LISP-symbol 'wife' or 'husband' depending on the sex of the person.

---

[2] The two brackets in front of the symbol adult-p are caused by the fact that in an explicit subject-description, the subject-name from the implicit description is replaced by a more complex description, consisting of the subject-name and the subject-concept description. Thus: "adult-p" becomes "(adult-p (a subject ...))".

```
(defconcept PERSON
  ...
  (sex (a sex))
  ((partner
    (a synonym-subject
      (synonym-for
        (a symbol
          (definition [form (if (eq (>> sex) (>> of male))
                                'wife
                                'husband)])))))
```

To find the partner of John, the subject-concept is first inherited from Person. Then, the subject-concept's referent needs to be computed. Therefore the partner-subject of John inherits the definition from Synonym-Subject. This definition is described with

```
(a form
  (definition [form '(>> ,(>> referent synonym-for) of ,(>> subject-of))]))
```

To find the partner-subject's referent, the referent of this definition must be evaluated, but this referent is not known yet. So the referent of the definition of the partner-subject is recursively computed, by evaluating the referent of the definition of the definition, which is the lisp-form

```
'(>> ,(>> referent synonym-for) of ,(>> subject-of))
```

Before evaluating this form we need to determine the context of the relative requests. The lexical scope of this description is the concept Synonym-Subject so the context is the concept who is inheriting this description from Synonym-Subject or the partner-subject-concept of John.

Thus, the referent of the definition of the partner-subject-concept is the result of the evaluation of

```
'(>> ,(>> referent synonym-for of <synonym-subject #4>)
     of ,(>> subject-of of <synonym-subject #4>))
```

or (assuming that the sex of John is Male)

```
(>> wife of John)
```

So finally the evaluation of this form returns the partner of John (figure 5). A more cryptic trace of the computation is given below. Notice that in this trace, there are still some shortcuts made.

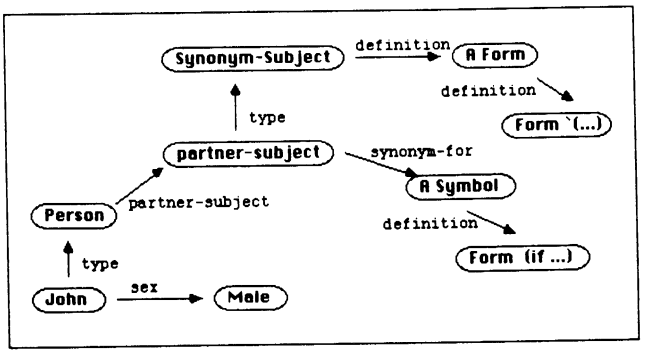


Fig 5: A synonym-subject.

```
-->  (>> partner of john)
=    (>> referent of <synonym-subject #4>)
=    (>> eval referent definition of <synonym-subject #4>)
=    (>> eval referent of <form #5>)
=    (>> eval eval referent definition of <form #5>)
=    (>> eval eval referent of <form #6>)
=    (>> eval eval of '(>> ,(>> referent synonym-for) of ,(>> subject-of)))
;;   Evaluation of '(>> ,(>> referent synonym-for) of ,(>> subject-of))
;;   with context  <synonym-subject #4>
=    (>> eval of (>> wife of john))
;;   Evaluation of (>> wife of john))
<--  <mary>
```

This example shows that although there is only one simple inheritance rule applied, one single request can still result in multiple orthogonal inherited subjects. This is the case because the inheritance mechanism is carefully chosen such that it benefits from other characteristics of the KRS concept-graph. Essential in this whole mechanism is the explicitness of subject-concepts, the recursiveness of the representation axiom, the use of lexical scope for the computation of the contexts, the uniformity of the concept-graph and the mechanisms that work on it, and even the caching and consistency maintenance system to make it run efficiently.

1.4.2. Concept-Names.

A *concept-name* plays the same role for a named concept as a subject-concept for its filler. When a concept is defined with a defconcept-description, that concept is not yet created. Instead a concept-name-concept is created, with as definition a form to create the concept. For example the defconcept-description that described the concept John returns a concept-name that can be retrieved with the following request:

```
-->  (>> handler of john)
<--  <john name>
```

The first time the concept John is actually needed, the referent of this concept-name is computed and cached.

```
--> (>> referent handler of john)
<-- <john>
```

Concept-names have two purposes. First they maintain the lazy-evaluation idea. A second important purpose is to make sure that the consistency-maintenance mechanism is complete. Suppose we have the following definitions:

```
(defconcept RED
  (a number))

(defconcept MY-CAR
  (a car
    (color red)))
```

The first time we ask for the color of My-Car, the concept Red is cached and returned. If we now change the concept Red, for example in

```
(defconcept RED
  (a color))
```

the cached color of My-Car must be withdrawn. Since the consistency mechanism works between referents and definitions, this happens if the concept Red is itself the referent of some other concept, i.e. the result of the evaluation of some definition.

1.4.3. Meta-Interpreters.

KRS has been subject of an extensive study on Computational Reflection [Maes87], parts of which are incorporated in the current running version.

Every-concept has a unique meta-interpreter which can be retrieved via the subject "meta-interpreter". The referent of the meta-interpreter of a concept is again the original concept:

```
--> (>> meta-interpreter of john)
<-- <meta-interpreter #4>

--> (>> referent meta-interpreter of john)
<-- <john>
```

A first purpose of a concept's meta-interpreter is to contain static knowledge about the concept itself. It contains for example a concept's context, pname, concept-name, etc.

A second purpose is to determine parts of the dynamic behavior of the concept. These parts can be overridden by the user. In the current version, control of the interpretation process is rather limited. An object's pname is specified in its meta-interpreter. Data-concepts for example have a specialized pname such that if their referent is cached it becomes part of the pname. A number without a cached referent is printed like

```
<number #7>
```

When the referent is cached the concept is printed like

```
<number 26>
```

Currently the meta-interpreter also contains information about how to parse special-instance-descriptions. The function to parse the description

```
[number 26]
```

is part of the meta-interpreter of the concept Number. Those functions can be defined by the user.

There exists an experimental implementation (3-KRS) in which the dynamic control of the meta-interpreters in significantly extended.

2. THE CONCEPT-GRAPH.

2.1. Overview.

This section is about the KRS concept-graph. It discusses the following aspects:

- *Concepts* are nodes of the concept-graph representing an entity in the application domain.
- *Subjects* are named unidirectional relations between pairs of concepts. They constitute the arcs of the concept-graph.
- The *representation axiom* is a constraint between two special-purpose subjects of a concepts: its referent-subject and its definition-subject. It defines how to compute the former out of the latter.
- All concepts have a special-purpose-subject called "type". Type-subjects classify concepts in a type-tree, which is used for *inheritance*.

2.2. Concepts.

2.2.1. Introduction

A *concept* is a node of the concept-graph, representing an entity in the representation domain. This entity is called the concept's *referent*. We distinguish three kinds of concepts depending on the kind of referent they have. *External-concepts* represent an external entity, i.e. a (real or abstract) entity in some external domain. *Data-concepts* represent LISP-objects like numbers, strings, lists, programs, etc. Finally *meta-concepts* represent another concept.

Some concepts have a name. They are called *named* concepts. Others, called *anonymous* concepts, have not.

2.2.2. Accessing a Concept.

A named concept can be accessed with a request (1) or with the function Find-Concept (2).

```
--> (>> of john)                                                    1
<-- <john>


--> (find-concept 'john)                                            2
<-- <john>
```

An anonymous concept can sometimes be accessed by a request (3), i.e. when it is the filler of a subject).

```
-->  (>> father of john)                                    3
<--  <person #25>
```

find-concept *concept-name* *[function]*

returns the concept whose name is *concept-name*. If it does not exist, an error occurs.

SEE ALSO

requests: section 2.3.3.

2.2.3. Defining a Named Concept.

A *defconcept-description* is a description of a named concept, using the macro Defconcept.

defconcept *concept-name concept-description* *[macro]*

stores *concept-description* as the description of the concept named *concept-name*. The description is not interpreted now. It will be interpreted the first time the concept is used. *Concept-name* is a LISP-symbol. *Concept-description* is one of the allowed concept-descriptions. An overview of those is given in section 2.2.6.

EXAMPLES

```
(defconcept PERSON)

(defconcept JOHN
  (a person))
```

SEE ALSO

concept-descriptions: section 2.2.6.; concept-name-concepts: section 2.2.4. lazy evaluation: section 3.2.

2.2.4. Concept-Name-Concepts.

A *concept-name-concept* is a concept created by a defconcept-description. The concept described by that defconcept-description is the referent of the concept-name-concept. The type of a concept-name-concept is the concept Concept-Name.

EXAMPLE

(4) shows the description of the concept-name-concept for the concept John (5). The function of the defconcept-macro is to create this concept-name-concept and to store it on the plist of the LISP-symbol "john".

```
(a concept-name                                                    4
  (lisp-name [symbol john])
  (definition [form (a person)]))


(defconcept JOHN                                                   5
  (a person))
```

The concept Concept-Name is the type of all concept-name-concepts.

Concept-Name *[concept]*

a concept. The type of Concept-Name is the concept Handler.

lisp-name *[subject of Concept-Name]*

the lisp-name of a concept-name-concept is a concept whose type is the concept Symbol and whose referent is the LISP-symbol the concept-name-concept is attached to.

A concept-name-concept can explicitly be accessed by putting the LISP-symbol "handler" in front of the "of" in a request.

EXAMPLES

```
--> (>> handler of john)
<-- <john name>

--> (>> referent definition handler of john)
<-- (a person)
```

SEE ALSO

indefinite-descriptions: section 2.2.8.; lazy evaluation: section 3.2.

2.2.5. Printed Representations.

The printed representation of a concept is customizable.

The default printed representation of a named concept consists of the name of the concept between the triangular brackets "<" and ">" (6).

```
<john>                                                             6
```

The default printed representation of an anonymous concept consists of a LISP-symbol and a unique identifier, also between the triangular brackets "<" and ">" (7). The LISP-symbol is the concept's *concept-name-of-type*. This is the name of the concept's type if this is a named concept. Otherwise, it is the concept-name-of-type of this concept. The unique identifier is a LISP-symbol starting with a sharp-sign (#) and followed by an integer. This identifier is called the concept's *concept-id*.

```
<person #23>                                                    7
```

The printed representation of a concept-name-concept consists of its lisp-name and the LISP-symbol "name" between the brackets "<" and ">" (8).

```
<john name>                                                     8
```

The default printed representation of a subject-concept consists of three elements. The first is a concatenation of the subject's name and the LISP-symbol "-subject". The second is the LISP-symbol "of". The third is the subject-concept's owner (9).

```
<type-subject of <john>>                                        9
```

The default pname of an anonymous data-concept is variant. If its referent is cached, it consists of the concept's concept-name-of-type and the referent. It it is not cached, it is printed like an ordinary anonymous concept (10).

```
<number 109>                                                    10
```

SEE ALSO

printed representations: section 3.5.4. subject-concepts: section 2.3.2.; data-concepts: section 2.6.; caching: section 3.4.2.

2.2.6. Concept-Descriptions.

This paragraph gives an overview of concept-descriptions. These can be used where ever a concept-description is required, e.g. within a defconcept-description.

CONCEPT-CONCEPT-DESCRIPTION

A concept is a valid concept-description. It describes itself.

CONCEPT-NAME-DESCRIPTION

EXAMPLE

```
person
```

A concept-name is a valid concept-description. It describes the concept with that name.

SEE

section 2.2.3

CONCEPT-STRUCTURE-DESCRIPTION

EXAMPLE

```
(type person)
(size tall)
((best-friend
   (a subject
     (definition [form (arbitrary-LISP-form ...)]))))
```

A concept-structure is a sequence of subject-descriptions. It describes a new concept with subjects as described by the individual subject-descriptions.

SEE

sections 2.3.1 and 2.3.4

INDEFINITE-DESCRIPTION

EXAMPLE

```
(a person
  (size tall)
  ((best-friend
     (a subject
       (definition [form (arbitrary-LISP-form ...)])))))
```

Type-subjects play an important role within the concept-graph. An indefinite-description reformulates a concept-structure-description such that the role of the type is better emphasized. The indefinite-description in the example is equivalent to the example of a concept-structure-description given above. If both are used in the same description, the one that textually occurs last is taken.

SEE

section 2.2.8.

(RELATIVE-) DEFINITE-DESCRIPTION

EXAMPLE

```
(>> age of john)
```

A (relative-) definite-description describes a concept by means of a (relative-) request. It describes the concept the (relative-) request returns.

SEE

section 2.3.3.

SPECIAL-INSTANCE-DESCRIPTION

EXAMPLE

```
[number 1988]
```

A special-instance description is a customizable description, i.e. for each concept-name that occurs as first element of a special-instance-description, the user can define how the description is to be parsed. It is mostly used to describe data-concepts which have an initial referent, as in the example above.

SEE
section 3.5.5.

CONCEPT-VARIABLE-DESCRIPTION

EXAMPLE

```
?child
```

A concept-variable-description describes a concept by means of a concept-variable. To be valid, it must lexically occur within the scope of the variable.

SEE
section 3.6.1.

TILDE-DESCRIPTION

EXAMPLE

```
~foo
```

A concept-description can be a tilde-expression. The tilde-expression must evaluate to either a concept or a concept-name.

SEE
section 3.6.2.

2.2.7. Functions to Parse a Concept-Description.

The functions Parse-Concept-Description and Parse-Concept-Descriptions can be used to find the concept a description describes.

Parse-Concept-Description *description ks::distance ks::krs-env ks::lisp-env* *[function]*

given a concept-description, this function returns the concept it describes. All other arguments contain context information and must have exactly the same names in the function call as they have in the function definition.

Parse-Concept-Descriptions *descriptions ks::distance ks::krs-env ks::lisp-env* *[function]*

given a list of concept-descriptions this function returns the list of concepts they describe. All other arguments contain context information and must have exactly the same names in the function call as they have in the function definition.

EXAMPLE

```
--> (parse-concept-description 'john)
<-- <john>

--> (parse-concept-descriptions '(john (a person) [number 1]))
<-- (<john> <person #12> <number 1>)
```

2.2.8. Defining an Anonymous Concept.

Type-subjects play a special role within the KRS concept-graph. Therefore, a syntax which emphasizes the role of the type-subject improves the readability of KRS-descriptions.

An *indefinite-description* is a concept-description starting with one of the keywords "a" or "an", followed by a description of the concept's type, which is in turn followed by an optional concept-structure, describing the rest of the subjects.

Indefinite-descriptions are interpreted by the macros "a" or "an". Those can be used to create an anonymous concept, i.e. a concept without a name.

A *type-description &rest concept-structure* *[macro]*

returns a concept with type the concept described by *type-description*, and with additional subjects the subjects described in *concept-structure*.

AN *type-description &rest concept-structure* *[macro]*

The function "An" is a synonym for the function "A".

EXAMPLES

```
(defconcept RED                                                    11
  (a color))

(defconcept JOHN
  (a person
    (father (a person
              (color-of-hair red)))))
```

2.3. Subjects.

2.3.1. Introduction.

A subject is a labeled unidirectional link between two concepts. Given two concepts A and B and a subject with label L going from A to B, we say that A is the *owner* of the subject, that B is the *filler* of the subject, and that L is the *name* (or *subject-name*) of the subject.

> A *concept-structure-description* is a sequence of subject-descriptions, each of them describing a subject. In its simpliest form, a subject-description consists of a subject-name and a subject-filler-description.

The subject-name is a LISP-symbol, the subject-filler-description is a new concept-description. A subject-description describes a subject whose name is that particular subject-name and whose filler is the concept described by the subject-filler-description.

EXAMPLE

```
(defconcept JOHN                                                    12
  (type person)
  (father
    (type person)
    (color-of-hair red)))
```

The concept-structure-description describing the concept John (12) contains two subject-descriptions. The filler of the type-subject is described by the concept-name-description "person". The filler of the father-subject is again described by a concept-structure-description, describing a new concept with two subjects "type" and "color-of-hair" (figure 6).

SEE ALSO

subject-descriptions: section 2.3.4.

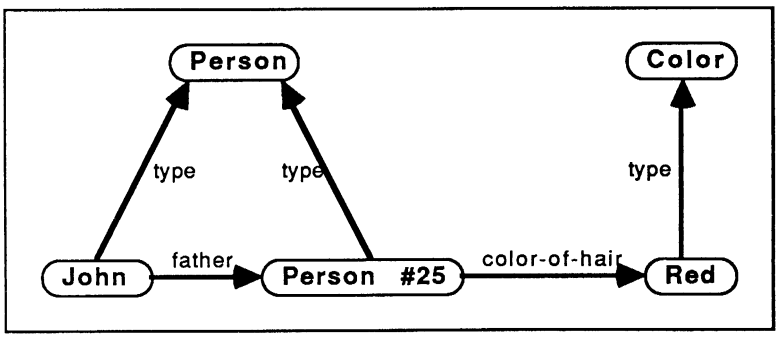


Fig 6: A part of the concept-graph.

SEE ALSO
the type-tree: section 2.5.1.

2.3.2. Subject-Concepts.

Subjects are themselves represented as concepts. We call a concept representing a subject a *subject-concept*. The referent of the definition of a subject-concept contains the description of its filler. The referent of the subject-concept is the subject's filler (figure 7).

EXAMPLE

```
(defconcept JOHN                                                    13
  (type person))


(a subject                                                          14
  (subject-name [symbol type])
  (owner john)
  (definition
    [form (find-concept 'person)]))
```

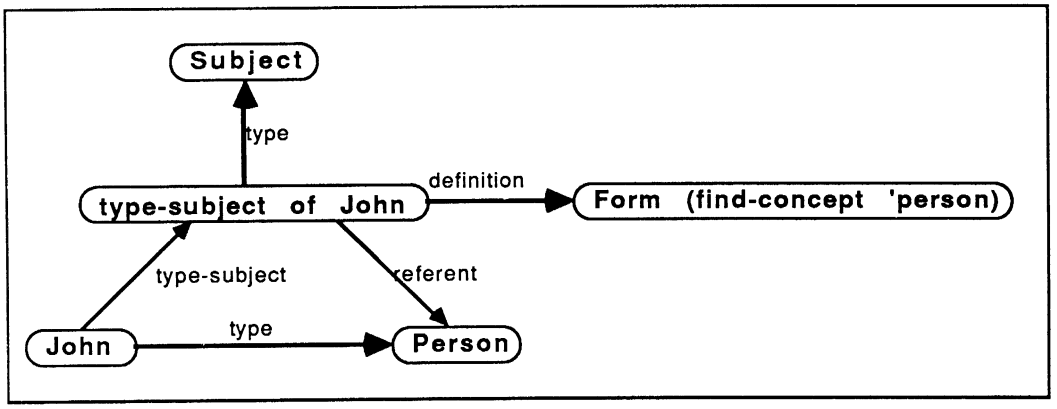


Fig 7: The concept-graph with explicit subject-concepts.

(14) shows the description of the subject-concept representing the type-subject of the concept John (13) (figure 8).

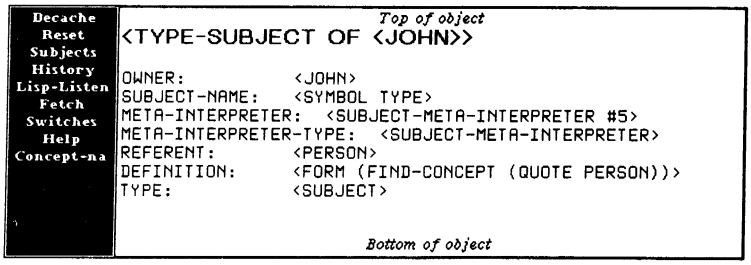


Fig 8: A subject-concept.

A subject-concept can be explicitly requested by using the LISP-symbol "handler" in front of the subject-name in a request.

EXAMPLES

```
-->  (>> handler friend of john)
<--  <friend-subject of <john>>

-->  (>> definition handler friend of john)
<--  <form (find-concept 'person)>
```

SEE ALSO

lazy evaluation: section 3.2.; referent-computation: section 3.4.

2.3.3. Retrieving Subject-Fillers.

A request is a query. It gets as input a concept-description and a series of subject-names.

>> *@subject-name-sequence &optional OF concept-description* *[macro]*

If *subject-name-sequence* is empty, the concept described by *concept-description* is returned. If there is only one subject-name, the filler of that subject of the concept described by *concept-description* is returned. A request with more than one subject-name is a concatenation of requests, i.e. it is equivalent to the result of a new request with as subject-name-sequence only the first element of the original subject-name-sequence, and with as concept-description the original request without this first element of its subject-name-sequence.

```
(>> subject-name-1 ... subject-name-n of concept-description)
≋  (>> subject-name-1  of
         (>> subject-name-2 ... subject-name-n of concept-description))
```

The argument OF must always be bound to the symbol OF. It separates the subject-name sequence from the concept-description.

EXAMPLES

```
-->  (>> of john)
<--  <john>

-->  (>> type of john)
<--  <person>

-->  (>> color-of-hair father of john)
≋    (>> color-of-hair of (>> father of john))
<--  <red>

-->  (>> type color-of-hair father of john)
<--  <color>
```

A *definite-description* is a concept-description formulated by means of a request. It is a description for the concept which is returned by the evaluation of the request.

EXAMPLE

```
(defconcept MARY                                                    15
  (a person
    (father (>> father of john))))
```

In (15), the filler of the father-subject of the concept Mary is described by a definite-description. So it is equal to the filler of the father-subject of John.

The concept-description within a request is optional. When omitted, the concept is determined dynamically, using the lexical-scope rule. Such a request is called a *relative-request.*

> A concept-description by means of a relative-request is called a *relative-definite-description.*

Subject-names in a request need not to be given as a LISP-symbol. They can be described by a new request. This internal request computes the subject-name dynamically. It must therefore return a LISP-symbol.

EXAMPLE

```
(defconcept PERSON
  (partner (>> (>> referent partner-subject-name)))

(defconcept JOHN                          (defconcept MARY
  (a person                                 (a person
    (partner-subject-name [symbol wife])      (partner-subject-name [symbol husband])
    (wife mary)))                             (husband john)))
```

SEE ALSO

lexical-scope: section 2.5.3.; subject-arguments: section 3.3.3.

2.3.4. Explicit Subjects.

Subject-concepts, implicitly created by the parser when interpreting subject-expressions, are in no way different from ordinary concepts. They have a type, the concept Subject, and a definition. Also their referent is computed using the standard representation axiom.

This opens a possibility for describing subject-concepts explicitly, which has considerable advantages:

- A user can give an arbitrary LISP-form to compute the subject's filler. Hence, the filler must not be described by one of the existing concept-descriptions, but can be the result of an arbitrary complex LISP-computation.
- Additional subjects can be defined for the subject-concept, meaning that the subject-concept obtains additional information.
- The full KRS-mechanisms can be used to compute the subject's filler, including inheritance.

The first element of a subject-description describes the subject-concept, the second element describes the subject's filler. For a simple subject-description (see section 2.3.1.), the former is just a LISP-symbol, and the latter is a full concept-description. For a subject-description describing a subject-concept explicitly, the first part is unfolded into a new couple, consisting of the subject's name (a LISP-symbol) and a description of the subject-concept (a concept-description). The second part, i.e. the subject-filler-description is omitted, because it is now inherently part of the subject-concept-description.

In practice, this means that an implicit subject-description starts with one bracket, while an explicit subject-description starts with two brackets.

EXAMPLE

```
(defconcept PERSON                                                    16
  (father (a person)))


(defconcept PERSON                                                    17
  ((father
     (a subject
       (definition [form (a person)])))))
```

The description of the father-subject in (16) is equivalent to its description in (17). In the former, the subject is implicit, in the latter it is explicit.

Explicit subject descriptions are particularly useful when the filler of a subject is the result of an arbitrary complex LISP-computation.

EXAMPLE

```
(defconcept LIST-OF-PERSONS
  (a concept-list
    (element-type person)
    ((oldest
      (a subject
        (definition
          [form (first (sort (>> referent)
                             #'(clambda (?person-1 ?person-2)
                                 (>> true-p
                                     (smaller-than (>> age of ?person-1))
                                     age of ?person-2))))])))))))
```

REMARK

Any subject-concept must be a subtype of the concept Subject.

Subject *[concept]*

a concept, ancestor of all subject-concepts in the type-tree.

owner *[subject of Subject]*

returns the owner of a subject.

subject-name *[subject of Subject]*

returns an instance of Symbol whose referent is the name of the subject.

EXAMPLES

```
--> (>> owner handler father of john)
<-- <john>

--> (>> subject-name handler father of john)
<-- <symbol father>
```

2.3.5. Adding Subjects to Existing Concepts.

The macro Defsubject adds a new implicit subject to a concept. The macro Define-Subject adds a new explicit subject.

defsubject *subject-name OF concept-description filler-description* *[macro]*

adds a subject with name *subject-name* and filler the concept described by *filler-description*, to the concept described by *concept-description*. The argument OF is a dummy argument, and must always be bound to the LISP-symbol "of". The result is *subject-name*.

define-subject *subject-name OF concept-description subject-concept-description* *[macro]*

adds a subject-concept with name *subject-name* and described by *subject-concept-description*, to the concept described by *concept-description*. The argument OF is a dummy argument, and must always be bound to the LISP-symbol "of". The result is *subject-name*.

EXAMPLES

```
(defsubject birthday of john
  (a date
    (day ten)
    (month march)
    (year [year 1961])))


(define-subject birthday of john
  (a subject
    (definition
      [form (a date
              (day ten)
              (month march)
              (year [year 1961]))])))
```

REMARK

The lexical-scope of subject defined by a defsubject or a define-subject is the same as the scope of the concept they are added to. For example, the relative-definite-description "(>> wife father)" in the subject-description below retrieves the wife of

the father of John, as it is interpreted in the lexical scope of the concept John.

```
(defsubject mother of john                                          18
  (>> wife father))
```

SEE ALSO
lexical scope: section 2.5.3.

2.4. The Representation Axiom.

2.4.1. Referents.

The referent of a concept is the entity the concept represents. For example, the referent of the concept Red might be the color red, the referent of the concept Person might be some prototypical person, and the referent of the concept John might be the person named "John". Concepts whose referent is a LISP-object are called *data-concepts.*

EXAMPLES

```
[number 1988]                                                       19
[list (a b c)]
[form (* 12 12)]
```

The descriptions in (19) are called *special-instance-descriptions.* They describe a data-concept together with its referent.

The first example in (19) describes a concept whose type is the concept Number and whose referent is the LISP-number 1988. The second describes a concept whose type is the concept List and whose referent is a LISP-list with elements the symbols a, b and c. The third example describes a concept whose type is the concept Form and whose referent is the LISP-form (* 12 12). (19) illustrates an application of a more general mechanism for defining concepts with a special-instance-description. The full mechanism is explained in section 3.5.5.

The referent of a concept can be retrieved with a request (20), or with the function Get-Referent (21).

```
--> (>> referent of [list (a b c)])                                 20
<-- (a b c)


--> (get-referent [list (a b c)])                                   21
<-- (a b c)
```

get-referent *concept* *[function]*

computes and returns the referent of the given concept.

get-referent-if-cached *concept* *[function]*

returns the referent if it was cached before. If not, it returns the LISP-symbol :failed.

SEE ALSO

special-instance-creators: section 3.5.5.; data-concepts: section 2.6.; requests: section 2.3.3.; caching: section 3.4.2.

2.4.2. Definitions.

The *representation axiom* is a constraint between the definition and the referent of a concept:

> *The referent of a concept is the result of the evaluation of the referent of the concept's definition (figure 9).*

In order for this constraint to be satisfied, a concept's definition (= the filler of its definition-subject) must be a concept whose referent is an evaluatable LISP-form. For this reason, it usually is a subtype of the concept Form.

(22) shows the description of the concept Two. The definition of Two is a concept whose type is the concept Form and whose referent is the LISP-form "(+ 1 1)" (23).

```
(defconcept TWO                                                    22
  (a number
    (definition [form (+ 1 1)])))


-->  (>> definition of two)                                        23
<--  <form (+ 1 1)>

-->  (>> referent definition of two)
<--  (+ 1 1)

-->  (>> type definition of two)
<--  <form>

-->  (>> referent of two)
<--  2
```

From the representation axiom follows that the description of the concept Two given in (22) is equivalent to the description given in (24).

```
(defconcept TWO                                                    24
  (a number
    (definition [form (+ 1 1)])
    (referent (>> eval referent definition))))
```

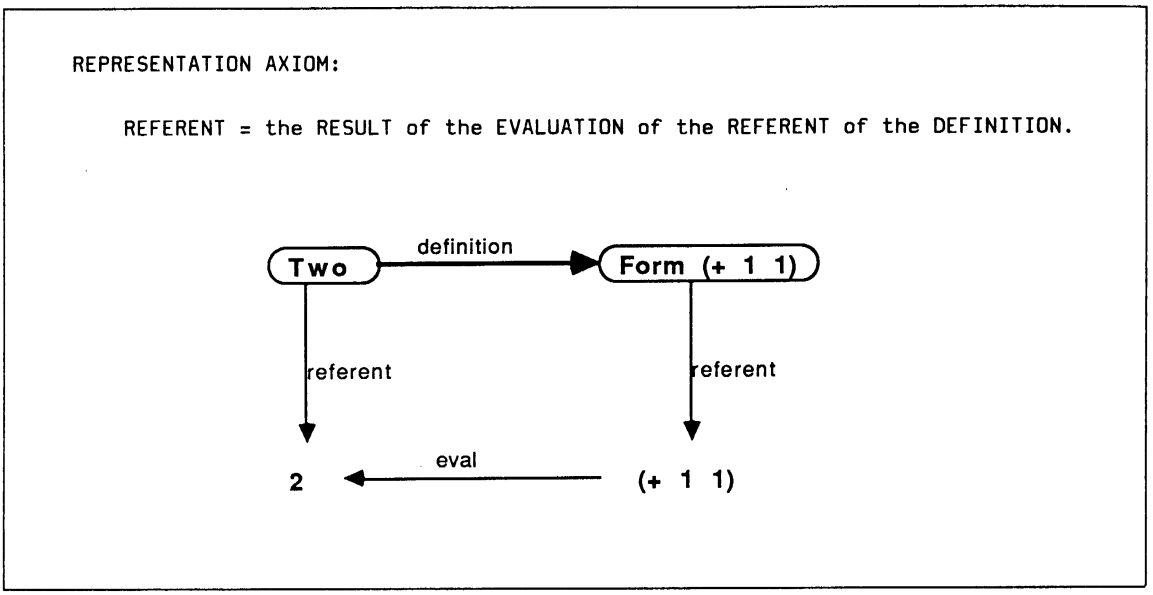


Fig 9: The Representation Axiom.

SEE ALSO

referent computation: section 3.4.; subtype: section 2.5.1.

2.5. Inheritance.

2.5.1. The Type-Tree

Every concept has a type-subject. When it is not explicitly part of the concept's description, it is implicitly inferred by the KRS interpreter. For top-level descriptions, the default type is the concept Summum-Genus. This way Summum-Genus is the top of a tree, called the *type-tree*, of which all concepts are an element (figure 10). The type of Summum-Genus is Summum-Genus itself.

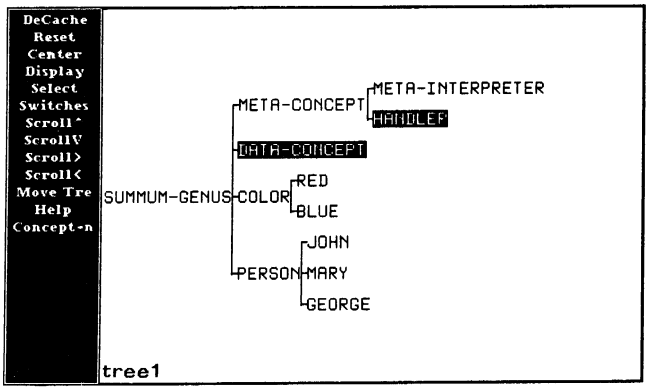


Fig 10: A part of the type-tree.

We say that a concept Foo is a *subtype* of a concept Bar, if the type of Foo is either Bar or a subtype of Bar. Instead of subtype, also the terms *specialization* or *instance* are occasionally used, depending on the semantic interpretation of the specific case.

2.5.2. Basic Inheritance Mechanism.

The type-tree is used to guide a very simple inheritance strategy:

A concept inherits all subjects of its type, except those it locally overrides.

EXAMPLE

```
(defconcept PERSON                                                    25
  (color-of-hair brown))


(defconcept JOHN                                                      26
  (a person
    (father (a person
              (color-of-hair red)))))
```

Given the description of person shown in (25), the concept John (26) inherits a color-of-hair-subject (27) while the father of John doesn't (28).

```
-->  (>> color-of-hair of john)                                       27
<--  <brown>


-->  (>> color-of-hair father of john)                               28
<--  <red>
```

An *inherited subject* of the concept X is every subject with owner X whose description was inherited from the type of X. A *local subject* of a concept X is every subject of X which is not inherited.

SEE ALSO
inheritance: section 3.3.2.

2.5.3. Lexical Scope.

Relative-requests (see section 2.3.3) are requests without concept descriptions (29). For a relative-request, the concept to which a query is directed is determined dynamically, using the lexical scope mechanism.

```
(>> age father)                                                       29
```

Relative-requests play the role of *self-references* in object-oriented languages [Goldberg83]. In OOP-terms, a self-reference during the response to a message sent to an object X, is an additional message to the object X. This additional message either obtains more information about X or performs part of the task. When objects share

code, for example via inheritance, there must be a way to access the object to which the message was sent. This object is called the *context* of the operation.

Similarly, a relative-request during the computation of the filler of a subject is a reference to the context of the computation. However, this is where the similarity stops. The context of a KRS computation is **not** the owner of the subject whose filler is being computed. Instead, the context is determined by means of the lexical-scope rule.

> *The lexical-scope of a description d, is the concept described by the global (outermost) description the description d is a part of.*
>
> *If there is no inheritance, the lexical-scope (or context) of a relative-request is the lexical-scope of this request.*
>
> *Whenever a concept A inherits a subject-description from a concept B, and B is the lexical-scope of relative-requests in the original subject-description, then A becomes the lexical-scope of the inherited relative requests.*

EXAMPLE

In (30), there are two relative-definite-descriptions, twice the form "(>> owner)". Both of them have as lexical scope the concept Car.

```
(defconcept CAR                                                    30
  (owner (a person))
  (driver (>> owner))
  (insurance-contract
    (a contract
      (company (an insurance-company))
      (insured (>> owner)))))
```

When computing for example the insured of the insurance-contract of the concept Car, the context of the relative-request "(>> owner)" is its lexical-scope Car. Hence, the owner of Car is returned (31) (figure 11).

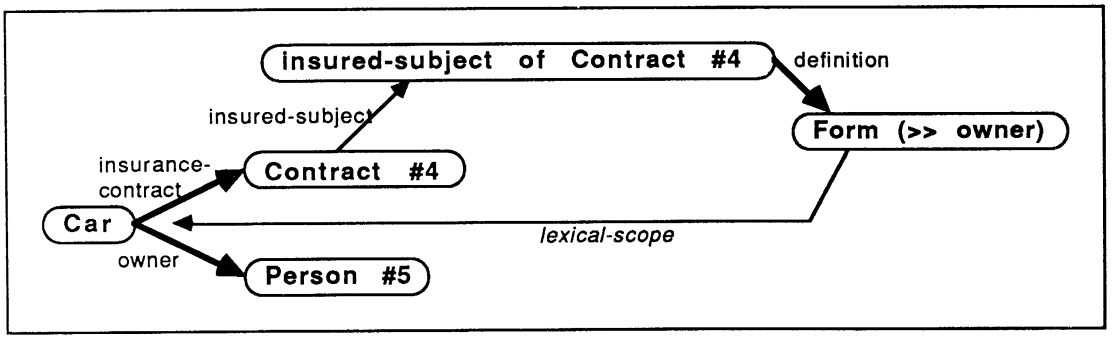


Fig 11: Without inheritance, context = lexical-scope.

```
--> (>> owner of car)                                              31
<-- <person #5>

--> (>> driver of car)
<-- <person #5>

--> (>> insured insurance-contract of car)
<-- <person #5>
```

When the insured of the insurance-contract of Johns-Car (32) is requested, the concept Johns-Car inherits the subject-description of the insurance-contract-subject from Car. At this time, Car is the lexical-scope for the relative-request "(>> owner)" within this inherited description. Hence, Johns-Car becomes the lexical-scope for the inherited relative-request (figure 12).

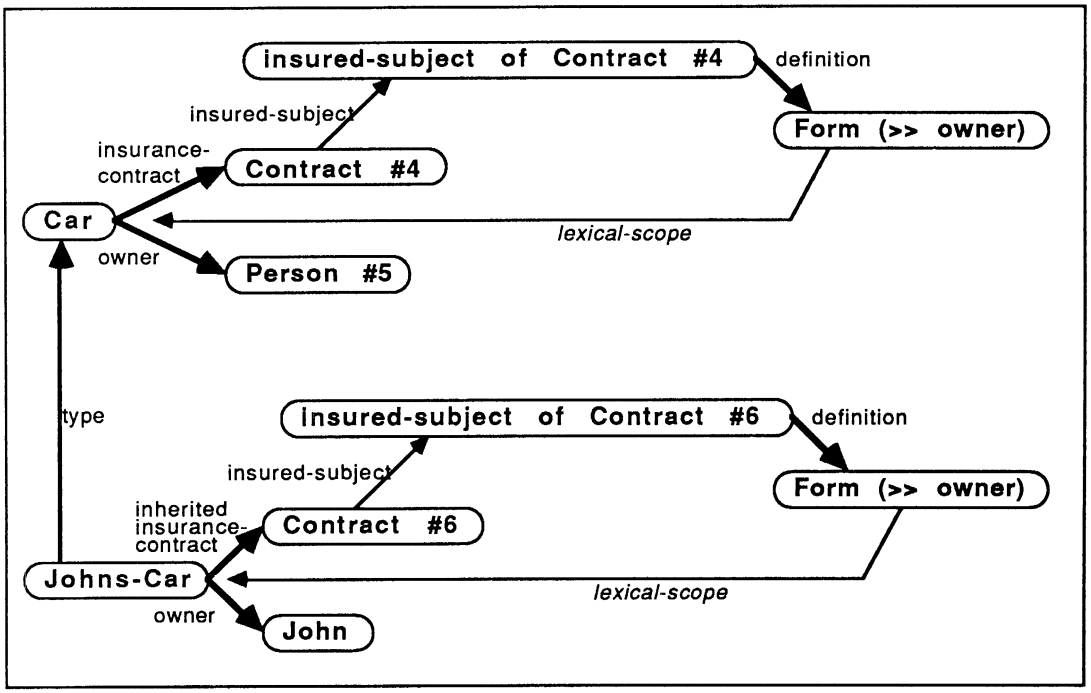


Fig 12: The computation of the context with inheritance.

```
(defconcept JOHNS-CAR                                              32
  (a car
    (owner john)))


--> (>> owner of johns-car)                                        33
<-- <john>

--> (>> driver of johns-car)
<-- <john>

--> (>> insured insurance-contract of johns-car)
<-- <john>
```

EXAMPLE

```
(defconcept PERSON                                                    34
  (car (a car
         (owner (>>))))))

(defconcept GEORGE
  (a person))


-->  (>> driver car of george)                                        35
<--  <george>

-->  (>> owner car of george)
<--  <george>

-->  (>> insured insurance-contract car of george)
<--  <george>
```

REMARK

The relative-definite-description in (36) describes the current context.

```
(>>)                                                                  36
```

REMARK

Since a defconcept determines the lexical-scope, a defconcept-description becomes a basic clustering operator. A defconcept determines a cluster of concepts within the concept-graph. All concepts in this cluster can access one another without having explicit subjects therefore [Van Marcke87a].

2.5.4. Inheritance for Subject-Concepts.

Subject-concepts are full fledged KRS concepts. Hence, to compute its referent, a subject-concept can for example use inheritance.

EXAMPLE

The salary-subject of Person (37) is described explicitly as a subtype of the concept Delegating-Subject. The definition of Delegating-Subject (38) delegates the request to the subject-concept's delegate-to. For the salary-subject of Person, this is the person's profession.

```
(defconcept PERSON                                                    37
  (profession (a profession))
  ((salary (a delegating-subject
             (delegate-to (>> profession)))))


(defconcept DELEGATING-SUBJECT                                        38
  (a subject
    (delegate-to)
    (definition [form (>> (>> referent subject-name) delegate-to)])))
```

```
(defconcept JOHN                          (defconcept TEACHER                    39
  (a person                                 (a teacher
   (profession teacher)))                     (salary low)))


-->  (>> salary of john)                                                         40
<--  <low>
```

REMARK

Notice the power of lexical-scope in this example. The two relative-requests in the definition of Delegating-Subject are self-references to individual subject-concepts. Hence, the scope of this defconcept-description is very narrow, i.e. it goes about things as small as an individual subject-concept.

SEE ALSO

The subject-library: section 6.4.

2.5.5. The Default Type of a Concept.

The default type of a concept is the type the KRS interpreter gives to the concept if the user has not specified a type-subject for it. For top-level descriptions, the default type is Summum-Genus. For a concept-structure describing a subject-concept, its default type is the subject-concept with the same name whose owner is the type of the owner of the new subject-concept. Similary, for a concept-structure describing a subject's filler, its default type is the filler of the subject with the same name and whose owner is the type of the owner of the subject whose filler is being described. We say that a type inferred in this way and the type of the enclosing concept are *parallel* (figure 13).

EXAMPLES

```
(defconcept RETIRED-PERSON                                                      41
  (a person
    ((salary
       (delegate-to (>> ex-profession))))))

-->  (>> handler salary of retired-person)
<--  <salary-subject of <retired-person>>

-->  (>> type handler salary of retired-person)
<--  <salary-subject of <person>>


(defconcept JOHNS-CAR                                                           42
  (a car
    (owner john)
    (insurance-contract
      (company AG))))
```

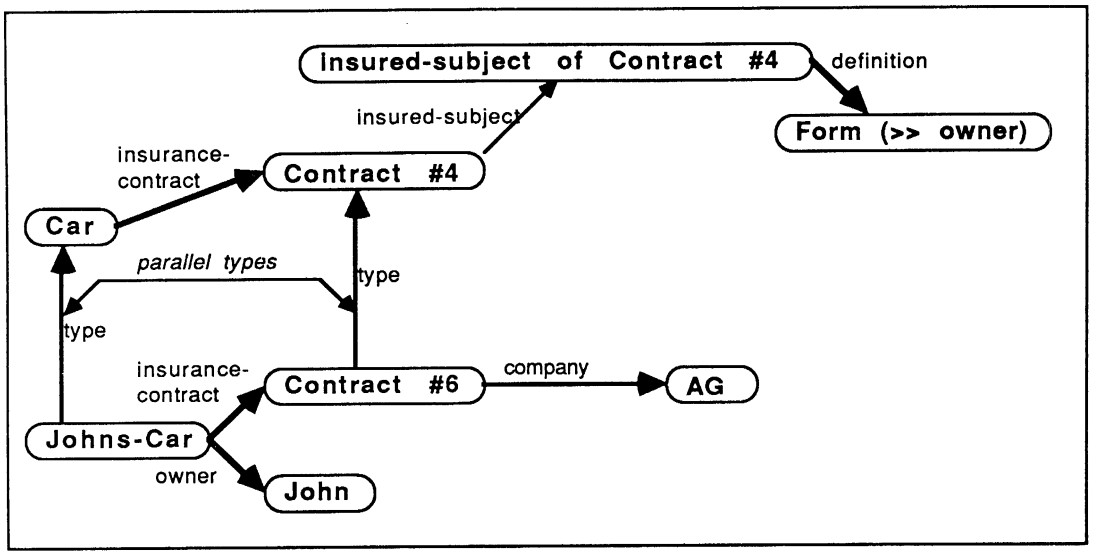


Fig 13: Parallel types.

2.5.6. Additive Inheritance.

Additive inheritance is a mechanism to augment the description of an inherited subject, without overriding it. This is often useful to change a part of an inherited description without having to entirely redefine it.

Parallel types give additive inheritance in the following way. The insurance-contract of Johns-Car (42) is a concept which has both the subjects defined for it in the description of Johns-car as the subjects defined for it in the description inherited from Car. For example the company-subject is defined locally (43), while the insured-subject comes from the description of insurance-contract inherited from Car (44).

```
--> (>> company insurance-contract of johns-car)                    43
<-- <ag>

--> (>> insured insurance-contract of johns-car)                    44
<-- <john>
```

REMARK

Notice again the importance of lexical-scope in this example. The insurance-contract of Johns-Car inherits its insured-subject from the insurance-contract of Car. Within the description of this subject there is a relative-request "(>> owner)" (30). The lexical-scope of this request is the concept Car. Hence, the concept Johns-Car inherits this description from Car and is thus the lexical scope of the inherited relative-definite-description.[3]

[3] This is what makes lexical-scope so difficult to explain and to implement. Luckily, what concerns lexical-scope, it is so that intuive models can easily be developed and prove more

2.6. Data-Concepts.

2.6.1. Introduction.

Data-Concepts are concepts whose referent is a LISP-object. Predefined data-concept types are Number, Form, List, Symbol, Boolean, Concept-List, Sequence.

SEE ALSO
data-concept-library: section 6.2.

2.6.2. Data-Concept Referents.

A data-concept's referent can be predefined or computable by means of the representation axiom. (45) shows a data-concept whose referent is predefined.

```
[number 3]                                                          45
```

The age of a person (46) is a data-concept whose referent is computed making use of its definition.

```
(defconcept PERSON                                                  46
  (birthyear (a number))
  (age (a number
         (definition [form (- (>> referent of current-year)
                              (>> referent birthyear))]))))

(defconcept CURRENT-YEAR
  [number 1988])

(defconcept JOHN
  (a person
    (birthyear [number 1950])))


-->  (>> referent age of john)                                      47
<--  38
```

REMARK
When a data-concept is described with a special-instance-description and the description of the referent starts with a tilde (~), then the referent of the data-concept is the result of the LISP-evaluation of this tilde-expression.

EXAMPLE

---

adequate than a profound understanding of the mechanism.

```
(defun MAKE-PERSON (age)
  #~(a person
       (age [number ~age])))

-->  (clet ((?person (make-person 35))) (>> age of ?person))
<--  <number 35>
```

SEE ALSO

tilde-expressions: section 3.6.2.

2.6.3. Encapsulating LISP-Objects with Data-Concepts.

Encapsulating LISP-objects with data-concepts is in general a good technique to improve the representational soundness and the expressive power of an application.

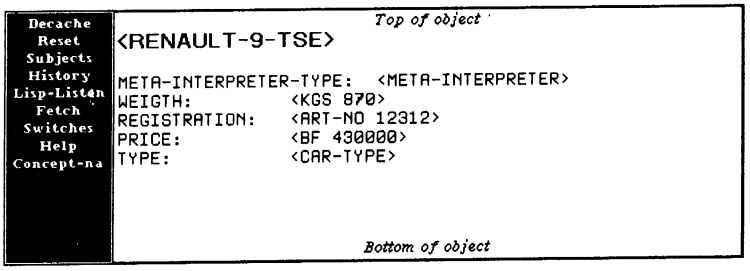


Fig 14: Encapsulating LISP-numbers with data-concepts.

EXAMPLE

A LISP-number has no built-in meaning. It may represent the abstract notion of a number, or a representation of a character, or an error message, etc. With data-concepts, we can specialize how we want numbers to be used. This does not only improve the soundness of the representation, but also the readability of the code (48, 49). In addition, the expressive power is augmented because it allows the user to define different subjects for numbers playing a different role.

```
(defconcept RENAULT-9-TSE                                    48
  (a car-type
    (price [number 430000])
    (registration [number 12312])
    (weight [number 870])))
```

49

```
(defconcept RENAULT-9-TSE
  (a car-type
    (price [bf 430000])
    (registration [art-no 12312])
    (weight [kgs 870])))

(defconcept BF
  (a number))

(defconcept ART-NO
  (a number))

(defconcept KGS
  (a number))
```

3. THE KRS INTERPRETER.

3.1. Overview.

This section defines the KRS interpreter. The following topics are discussed:

- *Lazy evaluation* is a technique for postponing the creation of structure as long as possible.
- Interpretation of requests takes place in two steps. First the requested subject must be retrieved, second the subject's filler must be found. The former step can possibly call on inheritance, the latter is referent computation.
- *Computation of referents* happens on the basis of definitions as predicted by the representation axiom. It is explained how acceptable efficiency is obtained by caching computed referents for possible re-use. It is also explained how caching plays a vital role in the lazy evaluation strategy. Finally, the need for and role of consistency maintenance is contemplated.
- *Meta interpreters* partly steer the behavior of the concept. They are inspired by Pattie Maes' work on computational reflection [Maes87].
- *Concept-variables and tilde-expressions are two kinds of descriptions which facilitate the communication between LISP-programs and KRS-descriptions.*

3.2. Lazy Evaluation.

3.2.1. Introduction.

The KRS concept-graph is constructed in a lazy way. The creation of structure is postponed as long as possible, i.e. until the first time it is requested.

- This lazy construction strategy is entirely based on the lazy computation of referents. A referent is computed the first time it is requested. Once computed, it is cached, so that it needs not to be recomputed the next time it is requested.
- Lazy evaluation is reduced to the lazy computation of referents of concept-name-concepts and subject-concepts. The purpose of both kinds of concepts is to remember a description of a concept. This description is stored as an executable LISP-form in the definition of the concept-name-concept or subject-concept. Hence, computing their referent involves evaluating this LISP-form which creates or retrieves the concept in question.

The major advantage of this technique is that we automatically benefit of consistency maintenance. In particular, consistency maintenance, which assures that all cached referents are consistent with the definitions they were computed from also assures that the concept-graph remains consistent with the KRS expressions by which it was described. When concept expressions are modified, the concept-graph is

automatically updated.

3.2.2. Encoding Concept-Descriptions in Definitions.

This section describes how concept-descriptions are encoded within LISP-forms, as is required for the lazy evaluation strategy.

CONCEPT-CONCEPT-DESCRIPTIONS

EXAMPLE

```
<person>
```

A concept-concept-description is encoded in a lisp-form that quotes the described concept.

GENERATED FORM

```
(quote <person>)
```

CONCEPT-NAME-DESCRIPTION

EXAMPLE

```
person
```

A concept-name-description describes a concept by its name. The constructed form uses the function Find-Concept (section 2.2.2).

GENERATED FORM

```
(find-concept 'person)
```

CONCEPT-STRUCTURE-DESCRIPTION

EXAMPLE

```
(size tall)
(father john)
```

A concept-structure is a sequence of subject-descriptions. They are passed as arguments to the function Make-Instance-Form-Description. The first argument of this function is the default-type, the second argument is the list of additional subject-descriptions.

GENERATED FORM

```
(make-instance-from-description
  (form to compute the default type)
  '((size tall) (father john))
  additional context information)
```

INDEFINITE-DESCRIPTION

EXAMPLE

```
(a person
  (size tall)
  (father john))
```

An indefinite-description is an evaluatable LISP-form and is thus kept as it is (section 2.2.8).

GENERATED FORM

```
(a person
  (size tall)
  (father john))
```

(RELATIVE-) DEFINITE-DESCRIPTION

EXAMPLE

```
(>> size of john)
```

A (relative-) definite-description is an evaluatable LISP-form and is thus kept as it is (section 2.2.6).

GENERATED FORM

```
(>> size of john)
```

SPECIAL-INSTANCE-DESCRIPTION

EXAMPLE

```
[number 1988]
```

A special-instance-description is transformed by the reader macro "[" into a LISP-form to call on the function which is defined in the meta-interpreter (section 2.2.6).

GENERATED FORM

```
(funcall (>> referent special-instance-creator meta-interpreter of number)
         'number '(1988) ...)
```

CONCEPT-VARIABLE-DESCRIPTION

EXAMPLE

```
?foo
```

A concept-variable is transformed by the reader-macro "?" into a LISP-form to select the requested variable out of the environment of concept-variable bindings (section 2.2.6).

GENERATED FORM

```
(krs-variable-value 'foo krs-env)
```

TILDE-DESCRIPTION

EXAMPLE

```
~foo
```

A tilde-description is transformed by the reader-macro "~" into a LISP-form to select the result of this description out of a previously generated environment (section 2.2.6).

GENERATED FORM

```
(lisp-eval 'foo lisp-env)
```

SEE ALSO
concept-descriptions: section 2.2.6.

3.3. Requests.

3.3.1. Handling Standard Requests.

A standard request is a query for the filler of a subject, given a concept and a subject-name. The standard mechanism is first to select the subject-concept with that name within the concept's memory, and second to take this subject-concept's referent (50).

```
--> (>> size of john)                                          50

    --> select size-subject of <john>
    <-- <size-subject of john>

    (>> referent of <size-subject of <john>>)

<-- <tall>
```

There are three exceptions to this. First, if it is a request for a referent, it is

handled as explained in section 3.4. Second, when it is a request with argument bindings, it is handled as explained in section 3.3.3. Third, if the subject-name is a new request, this new request is first executed within the same scope. This inner request returns the name of the subject to retrieve.

SEE ALSO
subjects-arguments: section 3.3.3.; dynamically computed subject-names: section 2.3.3.

3.3.2. Requests for Inherited Subjects.

When a subject-concept is searched for a particular concept, it might be part of the concept's own description, or it might be inherited. For example the size-subject of the concept John is local, its color-of-hair-subject is inherited (52) (figure 15).

```
(defconcept PERSON                                                    51
  (size average-length)
  (color-of-hair brown))


(defconcept JOHN                                                      52
  (a person
    (size tall)))
```

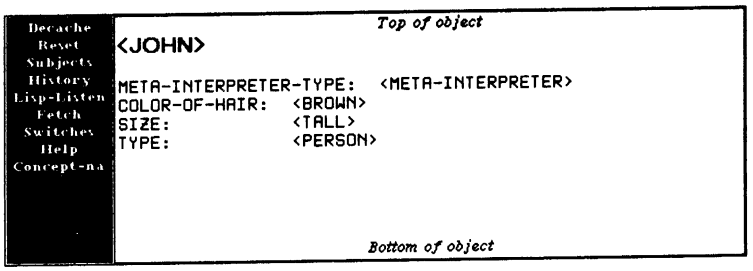


Fig 15: Local and inherited subjects of the concept John.

To inherit a subject for a concept X, the type-tree is climbed (first take the type of X, then the type of the type of X, etc.) until a parent is found which has a subject with that name.

The subject found this way is not used as such, but it is duplicated. This way, the original subject-concept and the inherited subject-concept represent two different subjects but with the same description (53).

```
--> (>> handler color-of-hair of person)                    53
<-- <color-of-hair-subject of <person>>

--> (>> handler color-of-hair of john)
<-- <color-of-hair-subject of <john>>

--> (>> definition handler color-of-hair of person)
<-- <form (find-concept 'brown)>

--> (>> definition handler color-of-hair of john)
<-- <form (find-concept 'brown)>
```

Duplicating an original subject is done by using the inheritance mechanism recursively. The inherited subject-concept is created as a new empty concept, whose type is the original subject-concept (54). This way, the inheriting subject-concept inherits itself all subjects from the original subject-concept (figure 16).

```
--> (>> type handler color-of-hair of john)                 54
<-- <color-of-hair-subject of <person>>
```

The duplicated subject-concept is stored within the local memory of the inheriting concept. The next time this subject will be needed, it will function as a local subject.

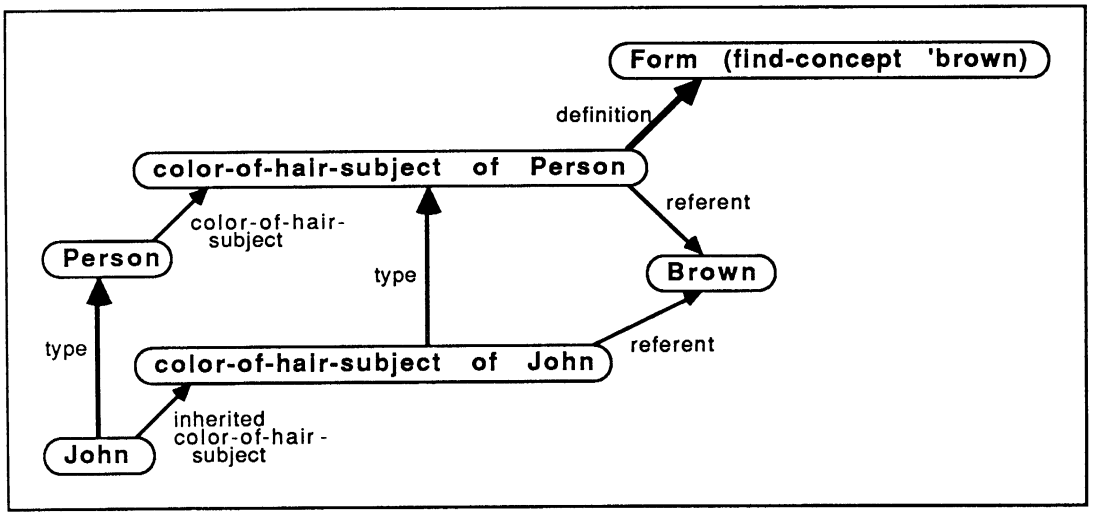


Fig 16: An inherited subject-concept inherits from the original.

3.3.3. Requests for Subjects with Arguments.

Subject-arguments are introduced to support an object-oriented programming style. A subject with arguments relates a concept to a set of others, i.e. it has a set of fillers. One such filler is selected by additional argument-bindings in the request.

The subject-concept for a subject with arguments has an arguments-subject. Its filler is a subtype of the concept Args, whose referent is a list of concept-variables.

In a request for the filler of a subject with arguments, the subject-name is a list. The first element in this list is the name of the subject, the others describe the concepts which will be bound to the subject's arguments.

Before evaluating the referent of the definition of a subject with arguments, the concept-variables mentioned above are bound to the concepts described by the descriptions in the request.

EXAMPLE

```
(defconcept NUMBER
  (a data-concept
    ((add
       (a subject
         (arguments [args ?add-what])
         (definition
           [form (a number
                   (definition
                     [form (+ (>> referent)
                              (>> referent of ?add-what))])])))))))

(defconcept GEORGE
  (a person
    (age [number 22])))

(defconcept JOHN
  (a person
    (age [number 25])))

-->  (>> (add (>> age of john)) age of george)
<--  <number 47>
```

REMARK

The filler of a subject with arguments **is never cached** (because it depends on the arguments). However, when a subject's filler is described by a request using subjects with argument, this subject's filler is cached after its first computation. In (55), the first time the age of Mary is requested, it is cached.

```
(defconcept MARY                                                    55
  (a person
    (age (>> (add (>> age of john)) age of george))))
```

SEE ALSO

caching: section 3.4.2.; concept-variables: section 3.6.1.

3.4. Referent Computation.

3.4.1. The Representation Axiom.

The *representation axiom* defines a constraint between a concept's referent and definition:

> *The referent of a concept is the result of the evaluation of the referent of the concept's definition.*

The representation axiom is introduced in section 2.4. The KRS interpreter uses the representation axiom to deduce the referent of a concept, if it is not known yet.

The referent of the age of John for example, is the result of the evaluation of the definition of the age of John (57).

```
(defconcept PERSON                                                    56
  (age (a number
         (definition
           [form (- (>> referent of current-year) (>> referent birthyear))])))))

(defconcept JOHN
  (a person
    (birthyear [year 1961])))


-->  (>> referent age of john)                                        57
     (>> referent of <number #25>)
     (>> eval referent definition of <number #25>)
     (eval '(- (>> referent of current-year) (>> referent birthyear)))
<--  27
```

Once the referent is known (cached for example), it is returned immediately (58).

```
-->  (>> referent age of john)                                        58
<--  27
```

3.4.2. Caching and Consistency Maintenance.

Referents are computed in a lazy way, i.e. upon their first request. When computed, a referent is cached. The next time the referent is requested, the cached result is returned without recomputation. Caching is not only an efficiency technique. It is also essential for the lazy evaluation strategy.

A cached referent however is not guaranteed to remain valid. When something changes which was used for computing the referent, the cached value may be no longer in compliance with the definition it was computed from. At that time, the cached value is withdrawn such that the referent can be recomputed the next time it is requested.

EXAMPLE

Figure 17/a shows a definition of the age of a person. After the computation of the referent of the age of John, it is cached. The figure shows both the concepts John and the age of John. Figure 17/b shows what happens when the concept Current-Year is redefined. The cached referent of the age of John is withdrawn. All other cached values (subject-fillers) remain valid. Figure 17/c shows the state after recomputing the referent of the age of John.

Changes which can cause cached referents to be withdrawn are: redefining existing concepts with defconcept, adding subjects with defsubject or define-subject, or mutating mutating-subjects.

REMARK

There is one important case in which the referent is **not cached** after its computation. Whenever a computation uses something which it changes afterwards, the result will not be cached.

3.4.3. How Referent Computation Bottoms Out.

The representation axiom defines a recursive mechanism to compute a concept's referent: to compute a referent we also need the referent of a definition.

There are two mechanisms to stop the recursion: *caching* and *predefined referents*. A cached referent stops the computation, because it avoids computation. There are also two kinds of predefined referents. First, the referent of a data-concept defined with a special-instance-description (59) is stored directly when the concept is created.

```
[number 1988]                                                    59
```

Second, referents of definitions created by the KRS parser to entail a concept-description (either within a concept-name-concept or within a subject-concept) are stored when the expression is parsed. For example when the defconcept-description (60) is evaluated, a concept-name-concept is created. The referent of the definition of this concept (the definition is shown in (61)) is predefined, and will thus stop the recursion.

```
(defconcept JOHN                                                 60
  (a person
    (birthyear [year 1961])))


-->  (>> definition handler of john)                             61
<--  <form (a person (birthyear [year 1961]))>
```

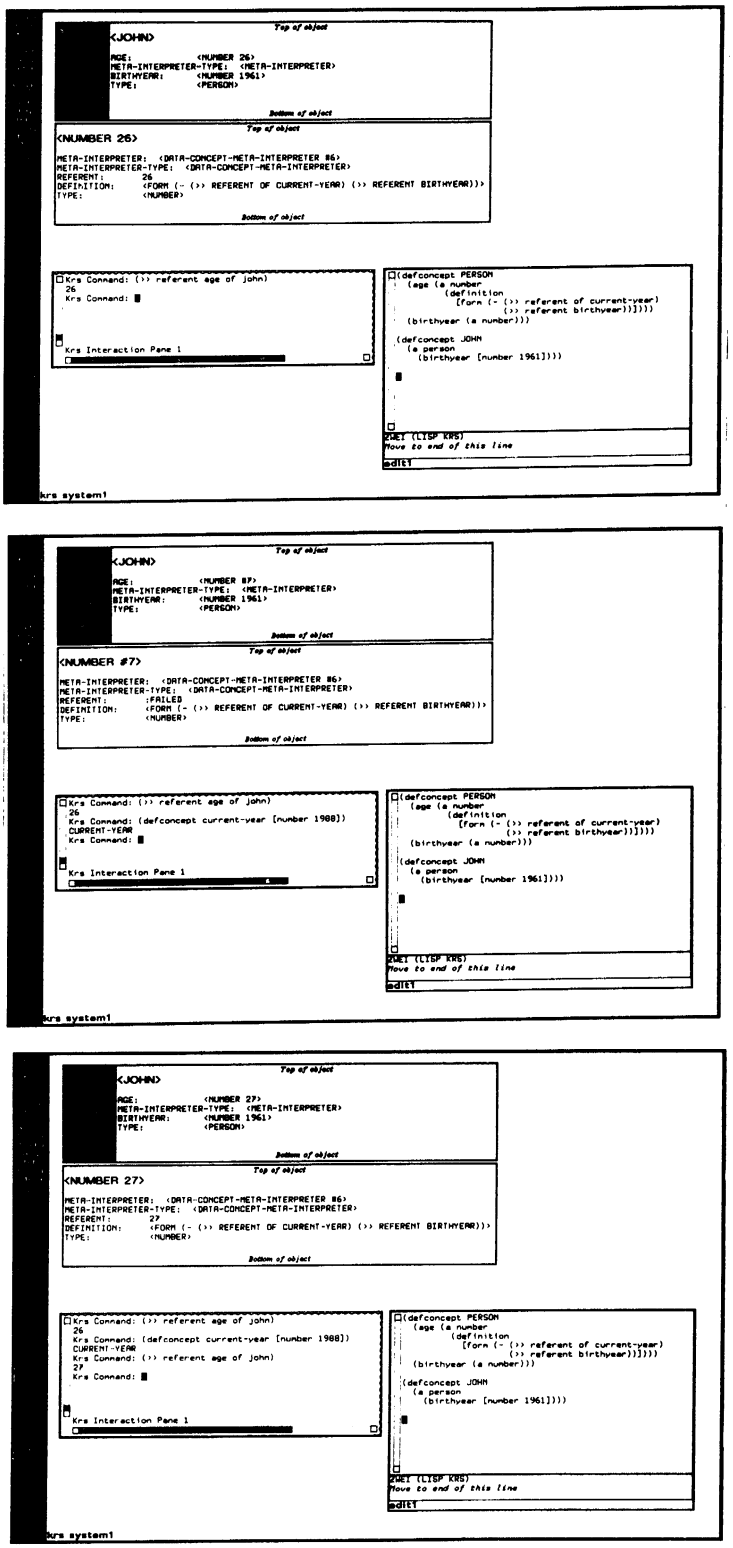


Fig 17: Consistency maintenance.

3.4.4. Algorithmic Concepts.

The major implication of defining the representation axiom recursively is that it supports a dynamic generation of definitions. Instead of directly providing a LISP-form to compute a subject's filler (which is similar to providing a method to respond to a message in OOP), it is now possible to circumscribe the method at a conceptual level, and let the actual LISP-form be generated from it.

A *conceptual definition* is a definition in terms of a more abstract computation concept. The referent of a conceptual definition is dynamically generated by evaluating the referent of the definition of the conceptual definition.

EXAMPLE

The definition of the age of Person (62) is described using the concept Subtraction. The referent of this definition is generated by evaluating the referent of the definition inherited from Subtraction.

```
(defconcept PERSON                                                    62
  (father
    (a person
      (size tall)))
  (birthyear (a number))
  (age (a number
         (definition
           (a subtraction
             (subtract-from current-year)
             (subtract-what (>> birthyear)))))))
```

The concept Subtraction is defined in (63). It has subjects subtract-from and subtract-what, both filled by a number. The referent is a LISP-form which computes the difference between the two numbers.

```
(defconcept SUBTRACTION                                               63
  (a form
    (subtract-from (a number))
    (subtract-what (a number))
    (definition
      [form '(- (>> referent of ,(>> subtract-from))
                (>> referent of ,(>> subtract-what)))])))
```

If the concept John is defined as in (64), this gives the results shown in (65).

```
(defconcept JOHN                                                      64
  (a person
    (birthyear [number 1961])))

(defconcept CURRENT-YEAR
  [number 1988])
```

```
-->  (>> referent definition age of john)                            65
<--  (- (>> referent of <current-year>)
        (>> referent of <number 1961>))

-->  (>> referent age of john)
<--  27
```

The caching mechanism makes that the new definition of age is not much slower than the LISP counterpart given in (66).

```
[form (- (>> referent of current-year) (>> referent birthyear))]          66
```

An important advantage of abstract computation concepts is that their structure can serve for other purposes than for the computation of result. The KRS program can use the structure of the definition to obtain more information about the computation and/or about the subject.

REMARK
It is important that the referent of a computation concept is a LISP-form which does not contain relative-requests any more. Otherwise, these will most likely be executed in the wrong context.

SEE ALSO
cliche-library: section 6.3.

3.5. Meta-Interpreters.

3.5.1. Meta-Interpreter and Meta-Interpreter-Type.

Every concept has a meta-interpreter and a meta-interpreter-type, both of them are concepts. A meta-interpreter-type is typically shared by many concepts, while a concept's meta-interpreter is unique.

A concept's meta-interpreter-type prescribes aspects of the behavior of the concept and its specializations. A concept's meta-interpreter represents the concept. The referent of a concept's meta-interpreter is the concept itself, its type is the concept's meta-interpreter-type (figure 18).

EXAMPLE

```
--> (>> meta-interpreter of john)                                         67
<-- <meta-interpreter #78>

--> (>> meta-interpreter-type of john)
<-- <meta-interpreter>

--> (>> referent meta-interpreter of john)
<-- <john>

--> (>> type meta-interpreter of john)
<-- (meta-interpreter>
```

One typically tunes the behavior of concepts by specializing their meta-interpreter-type. The system includes a few predefined meta-interpreter-types. The concept Meta-Interpreter is the default meta-interpreter-type and thus reflects the default behavior of a concept. The concept Data-Concept-Meta-Interpreter reflects the default

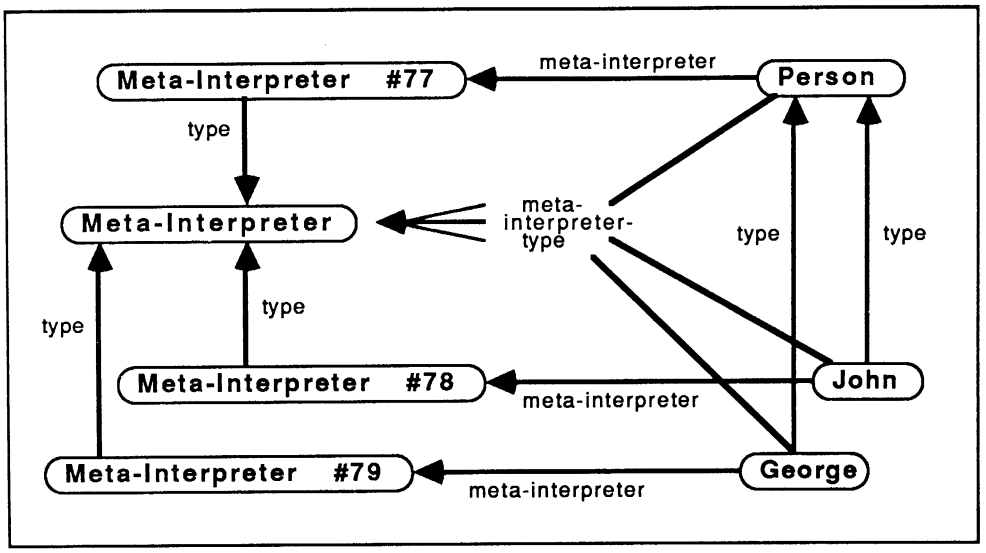


Fig 18: Meta-interpreter's and meta-interpreter-type's.

behavior of all data-concepts.

REMARK
One should not tailor the behavior of a concept by giving it a new meta-interpreter-subject.

3.5.2. The Concept Meta-Interpreter.

The concept Meta-Interpreter represents the default interpreter for concepts. Currently it determines the concept's printed representation and the way special-instance-descriptions are interpreted. In addition, it contains a lot of static knowledge about the concept, like the concept-name, the context, etc.

Meta-Interpreter *[concept]*

the default meta-interpreter-type.

concept-id *[subject of Meta-Interpreter]*

filler is a symbol whose referent is the concept-id of the concept. A concept's concept-id is the identifier used in the default pname of anonymous concepts, e.g. "#56".

concept-name-of-type *[subject of Meta-Interpreter]*

filler is a symbol whose referent is the name of the concept's first named ancestor in the type-tree.

subject-list *[subject of Meta-Interpreter]*

filler is a list whose referent is a LISP-list containing the subject-names of the concept's subjects.

inherited-subject-list *[subject of Meta-Interpreter]*

filler is a list whose referent is a LISP-list containing the subject-names of the concept's inherited subjects.

local-subject-list *[subject of Meta-Interpreter]*

filler is a list whose referent is a LISP-list containing the subject-names of the concept's local subjects.

concept-name *[subject of Meta-Interpreter]*

a Symbol. Its referent is the name of the concept, if it has one. Otherwise it is nil.

context *[subject of Meta-Interpreter]*

the context of a meta-interpreter is the lexical scope of all relative-requests that textually occur within the description of the context.

3.5.3. The Concept Data-Concept-Meta-Interpreter.

Data-Concept-Meta-Interpreter *[concept]*

the default meta-interpreter-type for data-concepts. Its type is the concept Meta-Interpreter.

cached-referent-p *[subject of Data-Concept-Meta-Interpreter]*

a boolean whose referent is T of nil, depending whether the referent of the concept is cached or not.

computable-referent-p *[subject of Data-Concept-Meta-Interpreter]*

a boolean whose referent is T if it succeeds to compute the concept's referent, and nil otherwise. If it succeeds, the referent will be cached. Possible reasons for not succeeding are that the concept has no definition or an error occurred during the evaluation of the definition.

3.5.4. Pnames

The pname-subject of a meta-interpreter determines the printed representation of the concept. It is a subtype of List. Its referent is a list[4] of LISP-objects to be printed between the brackets "<" and ">".

[4] It is a list instead of a string because a list is easier to manipulate.

pname *[subject of Meta-Interpreter]*

describes the default pname of a concept. It is defined as illustrated in (68).

```
(pname                                                                    68
  (a list
    (definition
      [form (let* ((concept-name (>> referent concept-name))
                   (type-name
                     (or concept-name
                         (>> referent concept-name-of-type))))
              (if concept-name
                  (list concept-name)
                  (list type-name (>> referent concept-id))))])))
```

pname *[subject of Data-Concept-Meta-Interpreter]*

describes the default pname of a data-concept. Is defined as illustrated in (69).

```
(pname                                                                    69
  (a list
    (definition
      [form (let* ((concept-name (>> referent concept-name))
                   (type-name
                     (or concept-name
                         (>> referent concept-name-of-type))))
              (cond
                (concept-name
                 (list concept-name))
                ((>> referent cached-referent-p)
                 (list type-name (>> referent referent)))
                (t (list type-name (>> referent concept-id)))))])))
```

Using this facility, it becomes possible to tailor the printed representation of a concept. An interesting usage of this is to change the pname of a data-concept such that the referent is always computed. This is done by the following subject-description (70). A data-concept whose meta-interpreter has this pname-subject will automatically try to compute its referent when it is printed.

```
(pname                                                                    70
  (a list
    (definition
      [form (let* ((concept-name (>> referent concept-name))
                   (type-name
                      (or concept-name
                          (>> referent concept-name-of-type))))
              (cond
               (concept-name
                 (list concept-name))
               ((>> referent computable-referent-p)
                (list type-name (>> referent referent)))
               (t (list type-name
                        (>> referent concept-id)))))])))
```

SEE ALSO

eager-data-concept-meta-interpreter: section 6.2.5.

3.5.5. Special-Instance-Creator.

A special-instance-description exists of a concept-name followed by additional information between the rectangular brackets "[" and "]". The special-instance-creator of the meta-interpreter of the concept with that name is a function which is used to retrieve or to create the described concept. For example the description in (71) is interpreted by (72). It allows the user to manipulate the parsing of special-instance-descriptions.

A meta-interpreter's special-instance-creator is a subtype of Function. Its referent is a LISP-function which must have the following argument-list:

```
(type info ks::distance ks::krs-env ks::lisp-env)
```

The names of the first and second may be changed, the others NOT. When this function is called, the argument type is bound to the name of the concept whose special-instance-creator is called. The second argument is bound to a list containing all elements occurring after the concept-name and before the right rectangular bracket (]) in the special-instance-description. All other arguments pass context information.

EXAMPLE

```
[foo 1 2 3 4]                                                              71


≈ (funcall (>> referent special-instance-creator meta-interpreter of foo)  72
           'foo
           '(1 2 3 4)
           ...)
```

EXAMPLE

Example (73) shows the description of the concept Sequence-Meta-Interpreter. It is the meta-interpreter-type of the concept Sequence, which represents concepts whose referent is a list of concepts. This concept is described in the data-concept library (see section 6.2.3.).

```
(defconcept SEQUENCE-META-INTERPRETER                                      73
  (a meta-interpreter
    (special-instance-creator sequence-sic)))

(defconcept SEQUENCE-SIC
  [function sequence-sic])

(defun SEQUENCE-SIC (type info ks::distance ks::KRS-env ks::lisp-env)
  #~(a ~type
      (definition [form (parse-concept-descriptions ~info
                          ks::distance ks::krs-env ks::lisp-env)])))
```

The concept Special-Instance-Creator is a specialization of Function. Its meta-interpreter has itself a special-instance-creator to facilitate the definition of special-instance-creators. The example in (74) defines the same special-instance-creator as in

(73) making use of the concept Special-Instance-Creator. The concept-variable ?type is bound to the concept whose special-instance-creator is being used, the concept-variable ?info is bound to a subtype of List whose referent is the list of objects occurring in the special-instance-description.

```
(special-instance-creator                                                    74
  [special-instance-creator (?type ?info)
     (a ?type
       (definition [form (parse-concept-descriptions (>> referent of ?info)
                           ks::distance ks::krs-env ks::lisp-env)]))])
```

REMARK

For efficiency reasons, we prefer the first solution to the second.

3.5.6. The Use of Special-Instance-Creator and Pname-subjects.

```
(defconcept PRODUCTION
  (meta-interpreter-type production-meta-interpreter)
  (condition (a function))
  (action (a function)))


(defconcept PRODUCTION-META-INTERPRETER
  (a meta-interpreter
    (special-instance-creator production-sic)
    (pname (a list
             (definition
               [form (list (or (>> referent concept-name)
                               (>> referent concept-name-of-type))
                           (>> referent condition)
                           (>> referent action))])))))


(defconcept PRODUCTION-SIC
  [function production-sic])


(defun PRODUCTION-SIC (type info ks::distance ks::krs-env ks::lisp-env)
  #~(a ~type
      (condition [form ~(first info)])
      (action [form ~(second info)])))


;;; Alternative solution:
; (defconcept PRODUCTION-SIC
;   [special-instance-creator (?type ?info)
;     (a ?type
;       (condition
;         (a form
;           (definition [form (first (>> referent of ?info))])))
;       (action
;         (a form
;           (definition [form (second (>> referent of ?info))])))])


(defconcept IF-FALSE
  (a production))
```

```
--> [if-false (>> starts car of ?case)
              (>> (try ?case) of start-problem-rule-base)]
<-- <if-false (>> starts car of ?case) (>> (try ?case) of start-problem-rule-base))>
```

3.6. Interfacing with LISP-code.

3.6.1. Concept-Variables.

A *concept-variable* is a concept-description, which is temporarily bound to a particular concept, i.e. during the execution of a piece of code.

A concept-variable consists of a LISP-symbol starting with a question-mark (75). Concept-variable bindings can be bound with one of the macros *clet*, *clet**, *cdolist* or *clambda*. It can be used as LISP-variable as well as as concept-description.

```
?child                                                          75
```

clet *((concept-variable-1 form-1) ...) &rest body* *[macro]*

All concept-variables are bound to the concept which results from the evaluation of the corresponding form, during the execution of *body*.

clet* *((concept-variable-1 form-1) ...) &rest body* *[macro]*

All concept-variables are bound to the concept which results from the evaluation of the corresponding form, during the execution of *body*, and during the execution of all the forms describing bindings for the remaining variables.

cdolist *(concept-variable concept-list) &rest body* *[macro]*

body is executed *n* times where *n* is the length of the referent of *concept-list*, while *concept-variable* is bound to the successive elements of the referent of *concept-list*. The LISP-function *return* ends the iteration.

clambda *(concept-variable-1 ...) &rest body* *[macro]*

creates a function which binds all the concept-variables to its input arguments, before executing *body*.

EXAMPLES

```
--> (clet ((?child (a person (younger-than [number 21]))))
      (>> referent younger-than of ?child))
<-- 21


--> (clet* ((?child (a person (younger-than [number 21])))
            (?childs-father (a person
                              (child ?child))))
      (list ?child ?childs-father))
<-- (<person #76> <person #77>)
```

```
--> (funcall #'(clambda (?childl ?child2)
                 (a number
                   (definition [form (/ (+ (>> referent age of childl)
                                           (>> referent age of child2))
                                        2)])))
              (a person (age [number 12]))
              (a person (age [number 8])))
<-- <number #78>
--> (get-referent *)
<-- 10
```

Concept-variables are particularly useful to successively access each element out of a list of concepts, for example when describing an operation to be executed on each of them.

EXAMPLE

```
(defconcept FAMILY
  (father (a person))
  (mother (a person))
  (children (a list-of-persons))
  (members
    (a list-of-persons
      (definition
        [form (list* (>> father) (>> mother) (>> referent children))])))
  (adult-members
    (a list-of-person
      (definition
        [form (mapcan #'(clambda (?one-member)
                          (when (>> true-p adult-p of ?one-member)
                            (list ?one-member)))
                      (>> referent members))]))))
```

REMARK

Concept-variables are lexically scoped. Hence, when a subject is inherited, its concept-variable environment is inherited accordingly. Concept-variables can thus not be used as dynamic variables as in the erroneous example (76).

```
(defconcept OLD-PERSON                                                    76
  (a person
    (age ?big-number)))

(clet ((?big-number [number 98]))
  (defconcept MY-GRANDFATHER
    (an old-person)))

--> (>> age of my-grandfather)
<-- Error: Attempt to use an undefined concept-variable ?BIG-NUMBER.
```

3.6.2. Tilde-Expressions.

Typically KRS expressions do not evaluate their arguments. This makes it difficult for a KRS expression to access the LISP-environment when executed within a LISP-function. Tilde-expressions solve this handicap.

For a LISP-form starting with "#~", all subforms starting with a "~" are evaluated within the LISP-environment of the global form. It can be compared with the existing LISP construct backquote-comma, except that a tilde-expression is **not quoted**.

EXAMPLES

```
(defun MAKE-PERSON (name age)                                    77
  #~(defconcept ~name
      (a person
        (age ~age))))

-->  (make-person mary [number 20])
<--  <mary name>

-->  (>> age of mary)
<--  <number 20>
```

Tilde-expressions can be used as an evaluatable LISP-form, as a concept-name, as a subject-name and as a concept-description.

3.7. Note on Virtual Concepts.

The concept-graph is defined in an infinite way. To be able to implement the concept-graph, it is necessary to shortcut this infinity at various places. This shortcut is done by making virtual concepts.

Virtual concepts are concepts which are not explicitly created. Instead, some condensed structure is built out of which the full concept can be deduced when necessary. In practice this means that a virtual concept is made explicit the moment a user wants to access it.

For efficiency reasons, virtual concepts are more frequently used than is absolutely necessary to stop the circularity. In the current implementation, all concept-name-concepts and all implicit subject-concepts are virtual.

4. THE INTERFACE.

4.1. Overview.

The KRS interface is a programmer's interface. It is designed to give meaningful information to a programmer building applications in KRS. Unlike for example the KEE-interface, it is not designed to be an end-user interface.

The KRS interface gives multiple views on the concept-graph. It knows of different types of windows, each having a special way of showing the concept-graph. There are for example tree-windows which show the type-tree, or inspect-windows which show the logical internal structure of one concept, and editor-windows which show the textual description of concepts. Each of the window-types has an associated menu through which commands can be sent.

The user can make as many instances of any of the window-types and configure them on the screen to have an optimal overview. The screen can be dynamically reconfigured any time. To easily move between different screen configurations, multiple pages can be constructed each having their own configuration.

Concepts shown in one window can be picked up with the mouse and dragged to other windows. For example, a node of the type-tree can be picked up in a tree-window and put down in an inspector-window. At that time, the inspector view on the concept is shown.

To each of the windows, a KRS concept can be attached. In that case, all of the menu-commands can also be performed via the subjects of the associated concept. This way, KRS programs may steer the interface. For example, if the concept Person-Inspector is associated to an inspector-window, then the request in (78) puts the concept John in this window.

```
(>> (putdown john) of person-inspector)                    78
```

4.2. Global Functionality.

4.2.1. General Commands on Windows.

All windows in the KRS interface obey the following commands.

OPEN/CLOSE MENU

When the mouse is moved close to the left edge of a window, a left-click opens the menu if it is closed, or closes it if it is open. The mouse cursor indicates when the cursor enters the sensitive region around the left edge (figure 20).

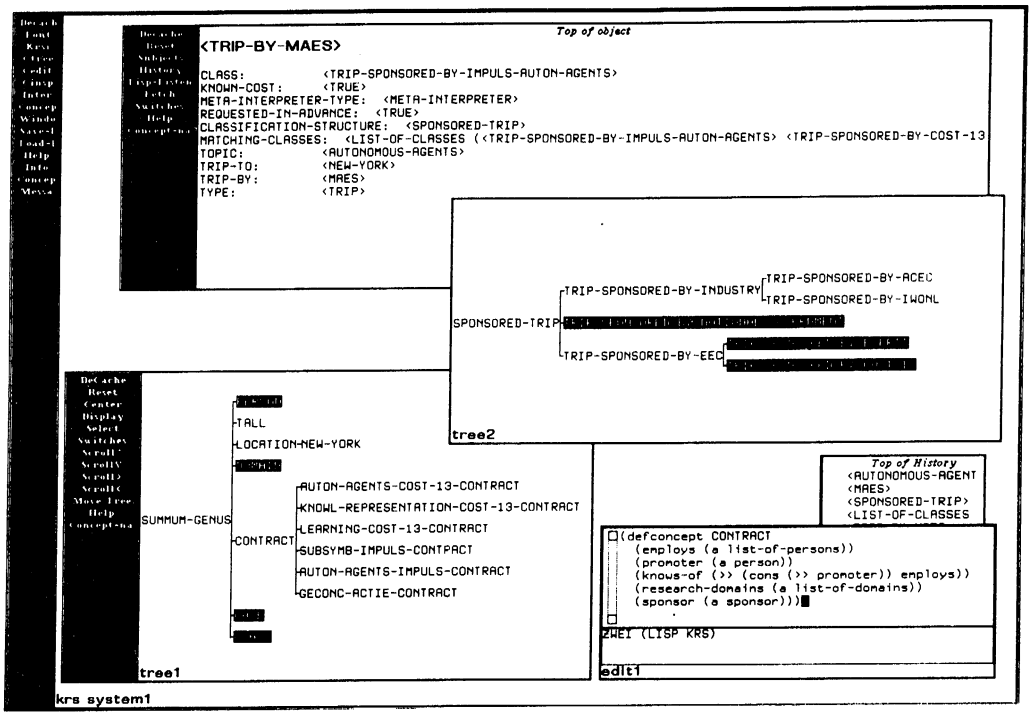


Fig 19: The KRS-interface.

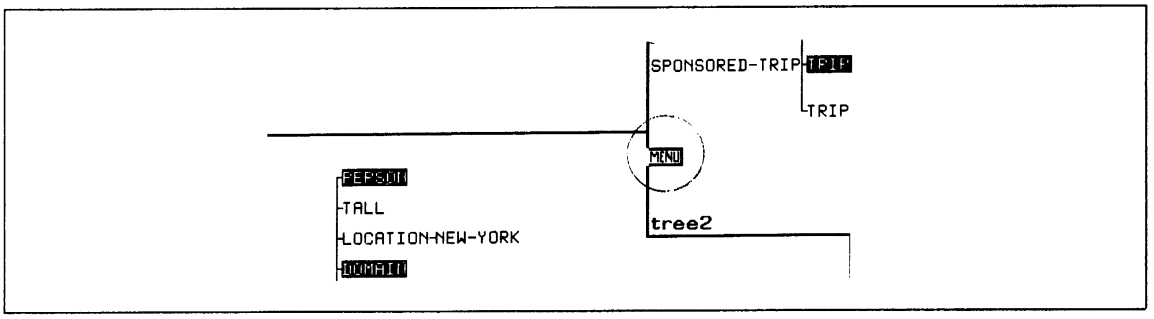


Fig 20: The mouse cursor changes when it is close to a left edge.

DRAG WINDOW

When the mouse is moved close to the upper left or the lower right corner of a window, one left click grabs the window to be dragged to another position. A second left click releases the window on its new position. The mouse cursor indicates when the mouse enters the sensitive region around the corner (figure 21).

RESHAPE WINDOW

When the mouse is moved close to the upper left or the lower right corner of a window, a double left click[5] allows to reshape the window. First the selected corner can be repositioned, followed by the opposite corner.

[5] Some symbolics-machines are configured such that double clicks are replaced by single clicks while the shift-key is pushed.

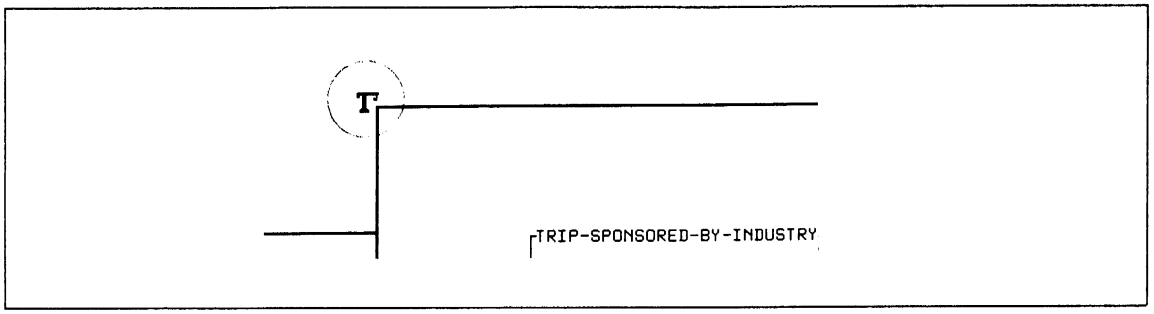


Fig 21: The mouse cursor changes when it is close to a corner.

SELECT
One single click on a window selects the window.

4.2.2. Dragging Concepts Around.

The pickup facility is the mechanism to move concepts around. Concepts can be picked up from a particular window by a double left click (figure 22a), dragged to another window by moving the mouse (figure 22b), and put down in another window by one left click in the other window (figure 22c).

4.3. Window Types.

4.3.1. The Interface Window.

This section explains all commands which appear in the menu of a global interface window.

refresh *[choice in interface-menu]*

Refreshes the screen. When a window is dragged or reshaped, the screen is not automatically refreshed. It can be refreshed with the menu-command *refresh.* To make the refresh instantaneous, the variable "ks::*automatic-refresh*" must be set to T.

fonts *[choice in interface-menu]*

Gives a list of fonts (figure 23). The selected font will be the default font for all KRS windows created afterwards.

krsi *[choice in interface-menu]*

Creates a new KRS Interface-Window.

ctree *[choice in interface-menu]*

Creates a Tree-Window (see section 4.3.2.).

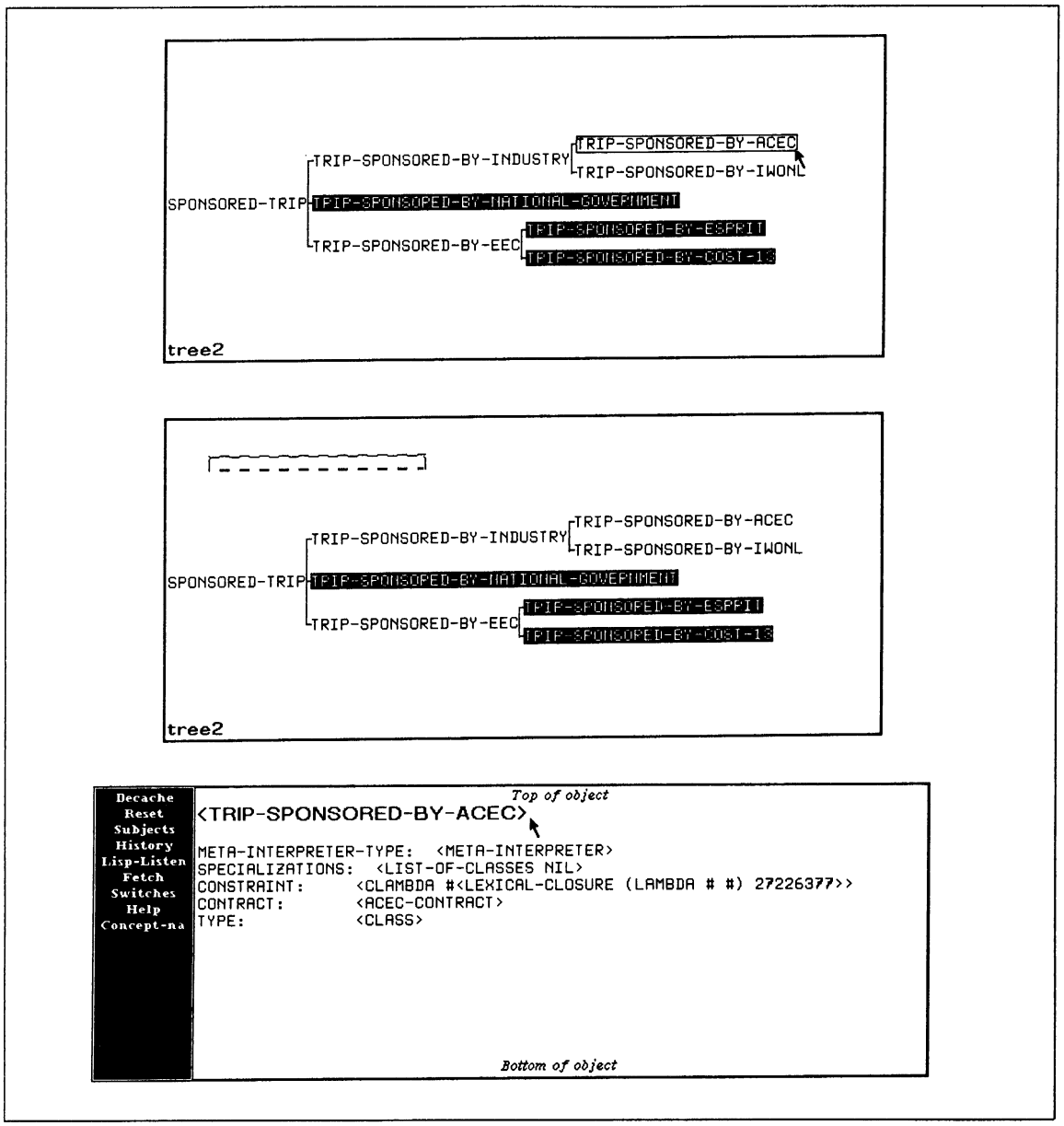


Fig 22: Pickup, Drag and Putdown.

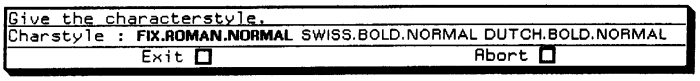


Fig 23: The font-list.

cedit *[choice in interface-menu]*

Creates an Editor-Window (see section 4.3.3.).

cinsp *[choice in interface-menu]*

Creates an Inspect-Window (see section section 4.3.4.).

inter *[choice in interface-menu]*

Creates an Interactor-Window (see section 4.3.7.).

concepts *[choice in interface-menu]*

Creates a Concept-Window (see section 4.3.5.).

windows *[choice in interface-menu]*

Pops up a multiple-choice window. Can be used to select or delete inferior windows or other KRS-interface-windows (figure 24).

concept-name *[choice in interface-menu]*

Asks for a concept-name. A concept with the given name is created with referent the particular window (see section 4.4.).

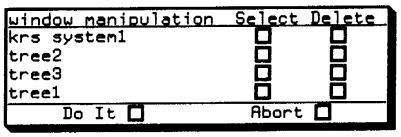


Fig 24: The windows-multiple-choice-menu.

4.3.2. The Tree-Window.

A tree-window (figure 25) is by default used to show the KRS type-tree. Only named concepts are shown. Nodes of the tree are by default closed, meaning that the inferior concepts are not shown. Black nodes are closed nodes with inferiors. White leaf nodes have no inferiors. Concepts which are not yet created (because of lazy evaluation) are not shown in the tree.

The menu-commands of a tree-window are listed below.

display *[choice in tree-menu]*

Redisplays the entire tree.

center *[choice in tree-menu]*

Moves the root of the tree to the middle of the left edge of the window.

select *[choice in tree-menu]*

Queries for a new root concept.

switches *[choice in tree-menu]*

Pops up a menu that allows to redefine some tree layout constants.

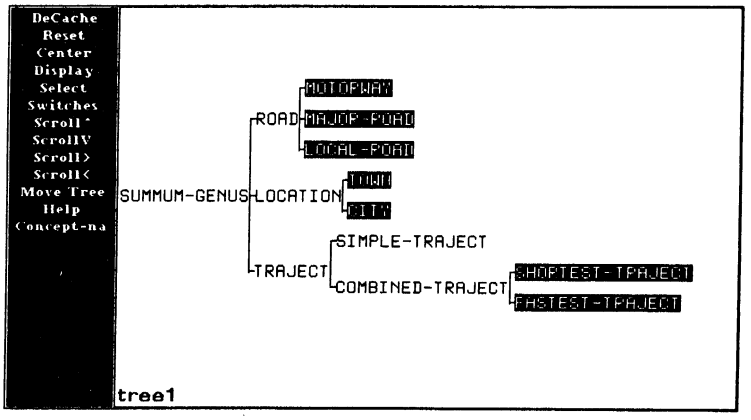


Fig 25: A Tree-Window.

refresh *[choice in tree-menu]*

Recomputes the tree layout.

scroll↑, scroll↓, scroll→, scroll← *[choice in tree-menu]*

Scrolls upwards, downwards, right or left.

move tree *[choice in tree-menu]*

waits for a second mouse click and moves the root of the tree to the place of that click. The new-position can be outside the tree-window, in which case only the part within the tree-window is shown.

The commands below are associated to mouse-clicks on the nodes of the tree.

MIDDLE-CLICK

Makes this node the new root of the tree.

DOUBLE-LEFT-CLICK

Pickup this concept.

RIGHT-CLICK

Pops up a menu with commands to close, open or hide the node. *Closing* a node means making its inferiors invisible. *Opening* a node means showing its inferiors. *Hiding* a node means removing this node and its inferiors from the graphical-tree (not from the physical tree behind it). To recover a hidden node, it must be explicitly selected with the menu-command select.

PUTDOWN

When a concept is put down in a tree-window, it becomes the new root of the tree.

The type-tree is actually a subgraph of the concept-graph, formed by the same nodes and by those links which are named "type". For many applications, other tree-structures can be extracted from the concept-graph in a similar way, like part-subpart

or goal-subgoal trees. The switches-menu allows one to define a different tree to be displayed in the tree-window (figure 26). By changing *root-concept*, a different root can be specified. By changing *subject-name*, a different way of selecting the branches can be given. When subject-name is "type", the parent-son relation is computed such that the parent is the type of the son. However, when the subject-name is something else, the parent-son relation is computed in the opposite way. If for example the subject-name is "foo", then the sons of a node are all elements in the list found by asking for the referent of the foo of this node (figure 27).

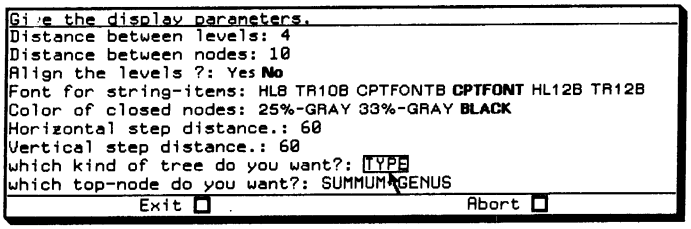


Fig 26: The switches-menu of a tree-window.

EXAMPLE

```
(defconcept PART
  (subparts [list-of-parts ()]))

(defconcept LIST-OF-PARTS
  (a concept-list))

(defconcept BRIDGE
  (a part
    (subparts
      [list-of-parts ((>> left-column)(>> right-column)(>> cross-beam))])
    (left-column (a column))
    (right-column (a column))
    (cross-beam (a cross-beam))))

(defconcept COLUMN
  (a part
    (subparts [list-of-parts ((>> base) (>> stack))])
    (base (a base))
    (stack (a stack))))

(defconcept BASE
  (a part))

(defconcept STACK
  (a part))

(defconcept CROSS-BEAM
  (a part))
```

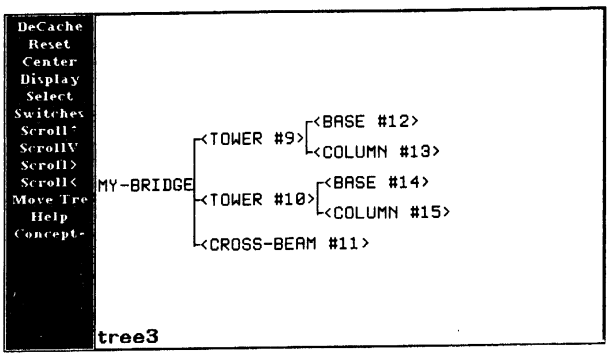


Fig 27: The subpart hierarchy.

4.3.3. The Editor-Window.

The editor-window is a ZWEI-editor-window [Symbolics87]. It can be used to type in or to change defconcept-descriptions. When a named concept is put down in an editor-window, the corresponding defconcept-description is retrieved and shown in the editor (figure 28).

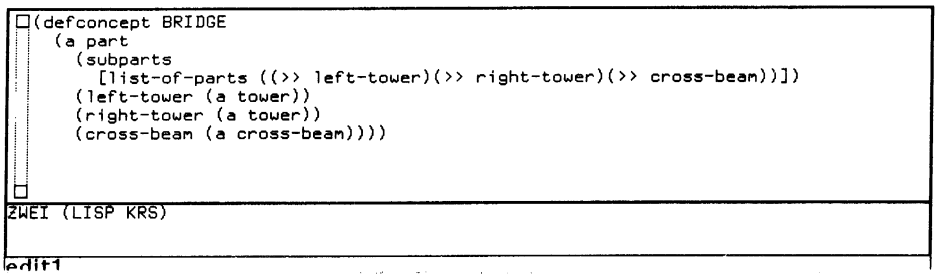


Fig 28: The Editor-Window.

The menu-commands for an editor-window are listed below.

eval *[choice in editor-menu]*

evaluates the defconcept-description. (this can also be done with the editor command control-shift-E).

save *[choice in editor-menu]*

asks for a filename to save the defconcept-description. The file is loaded into an emacs-buffer and the defconcept-description is added at the end of the buffer.

PUTDOWN

When a concept is putdown in an editor-window, its description is shown.

4.3.4. The Inspect-Window.

An inspect-window shows the state of a concept, i.e. its subjects and, if computed, also their fillers. Inherited subjects are not shown unless they were already requested. The keyword :failed indicates that a subject-filler is not yet computed.

One left-click on a subject-filler inspects this concept. One left-click on a subject-name inspects this subject-concept. One middle-click on a subject-name tries to compute this subject-filler. If it remains uncomputed, an error has happened. To find out which error, the request must be typed in an interactor-window. A double-left-click on a subject-filler picks up the concept.

The menu-commands for an inspector-window are listed below.

fetch *[choice in inspector-menu]*

asks for a concept-description. The resulting concept is inspected.

history *[choice in inspector-menu]*

creates and attaches a history-window to the inspect-window. All concepts inspected in the window are remembered in this history window. If there is already a history-window connected to this inspect-window, it is selected.

lisp listener *[choice in inspector-menu]*

creates and attaches a LISP-listener to the inspect-window. All results of LISP-forms evaluated in the LISP-listener are inspected. If there is already a LISP-listener associated with the inspect-window, it is selected.

refresh *[choice in inspector-menu]*

refreshes the window, to make sure that it reflects the current state of the concept it inspects.

The following commands are associated with mouse-clicks on concepts in a history window.

LEFT-CLICK
inspects the item in the corresponding inspect-window.

DOUBLE-LEFT-CLICK
picks up the concept to be dragged and put down somewhere else.

MIDDLE-CLICK
removes this concept form the history.

PUTDOWN

When a concept is putdown in an inspect-window, its description is inspected.

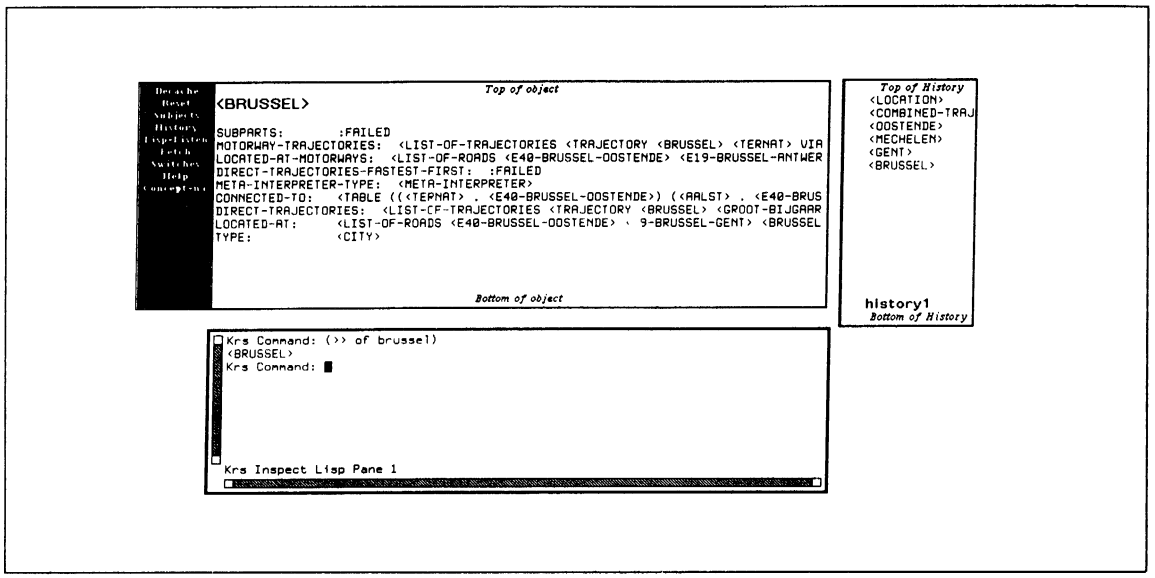


Fig 29: An inspect window with history and LISP-listener.

Figure 29 shows an inspect-window with associated LISP-listener and history-window.

4.3.5. The Concepts-Window.

The concepts-window displays all existing concept-names, in alphabetical order. A double-left-click on a name picks up this concept, if needed also creating it. A middle-click on a name creates that concept if it did not yet exist.

4.3.6. The Message-Window.

The message-window is a window to which output-messages can be sent. There is only one menu-command.

refresh *[choice in message-menu]*

clears the message-window.

4.3.7. The Interactor-Window.

The interactor-window is a LISP-listener, i.e. a window in which LISP-expressions can be evaluated. There is only one menu-command.

refresh *[choice in interactor-menu]*

clears the interactor-window.

Putting down a concept in an interactor-window binds the variable "*" to this

concept.

4.4. Associating Windows with Concepts.

The concept-name-command in a window's menu allows to define the name of a concept which will have the window as referent. This way, commands can be sent to windows using KRS requests.

(79) shows the kind of defconcept-description generated for example when the concept-name "part-whole-hierarchy" is given to a tree-window. (80) shows how the concept My-Bridge is put down in this tree-window.

```
(defconcept PART-WHOLE-HIERARCHY                                    79
  (a krs-tree-window
    (definition [form '<#:tree-window 12342>])))


(>> (putdown my-bridge) of part-whole-hierarchy)                     80
```

Interface-Concept *[concept]*

the root of all interface concepts.

refresh *[subject of Interface-Concept]*

refreshes its referent window.

putdown *?concept* *[subject of Interface-Concept]*

puts ?concept down in its referent window.

KRS-Interface-Window *[concept]*

concept representing a top-level interface-window. Its type is Interface-Concept.

new *?type* *[subject of KRS-Interface-Window]*

Creates a new concept whose type is *?type* and whose referent is a new inferior window of the kind described by *?type*.

KRS-Tree-Window *[concept]*

concept representing a tree-window. Its type is Interface-Concept.

center *[subject of KRS-Tree-Window]*

centers the tree in its referent window.

display *[subject of KRS-Tree-Window]*

redisplays the tree in its referent.

set-subject-name *?symbol* *[subject of KRS-Tree-Window]*

Changes the kind of tree to be displayed by its referent. ?Symbol is a concept with as referent a LISP-symbol. It denotes the name of the subjects which constitute the tree to be shown in the tree-window.

KRS-Editor-Window *[concept]*

concept representing an editor-window. Its type is Interface-Concept.

KRS-Inspector-Window *[concept]*

concept representing an inspector-window. Its type is Interface-Concept.

history *[subject of KRS-Inspector-Window]*

creates and/or returns a concept representing the inspector's history-window.

lisp-listener *[subject of KRS-Inspector-Window]*

creates and/or returns a concept representing the inspector's LISP-listener.

fetch *[subject of KRS-Inspector-Window]*

prompts for a concept-description and inspects the corresponding concept.

KRS-History-Window *[concept]*

concept representing a history-window of an inspector-window. Its type is Interface-Concept.

clear *[subject of KRS-History-Window]*

makes the history empty.

KRS-Inspect-Lisp-Window *[concept]*

concept representing a LISP-listener of an inspector-window.

KRS-Interactor-Window *[concept]*

concept representing an interactor-window. Its type is Interface-Concept.

KRS-Message-Window *[concept]*

concept representing a message-window. Its type is Interface-Concept.

refresh *[subject of KRS-Message-Window]*

clears the message-window.

print *?string* *[subject of KRS-Message-Window]*

prints the referent of ?string on the message-window.

5. ERRORS AND DEBUGGING.

5.1. Overview.

This section discusses errors that occur and how they can be solved. Besides the tracer, no dedicated KRS debugger is available at the moment. So when an error occurs, the standard LISP-machine debugger is entered.

5.2. Standard Errors.

5.2.1. Errors due to undefined objects.

UNDEFINED CONCEPT

MESSAGE

Error: Attempt to use an undefined concept XXX.

The LISP-symbol XXX in a concept-name-description is unknown as a concept-name.

REMEDY
Either define the concept or change the concept-name-description.

UNDEFINED SUBJECT

MESSAGE

Error: The subject YYY does not exist for the concept XXX.

Subject YYY is not found for concept XXX.

REMEDY
The subject-name might be misspelled in the request or in the concept-description. Sometimes this error is caused by scope-problems. In that case it is likely to be a problem of *incomplete instantiation* (see section 5.2.2.).

UNDEFINED CONCEPT-VARIABLE

MESSAGE

Error: The concept-variable ?XXX is not bound.

This error occurs whenever a concept-variable ?XXX is used as a concept-description at a place where it is not defined.

REMEDY

The name of the variable might be misspelled either where it is used or where it is defined. This error also occurs when a user tries to use concept-variables as dynamic variables: when a subject-description is inherited, its concept-variable environment is inherited accordingly. The concept-variable environment of the local concept cannot be used by the inherited description.

5.2.2. Errors due to Uninterpretable Concept-Structures.

CIRCULAR DEFINITIONS

MESSAGE

> *Error: Impossible to compute the referent of XXX, because it is defined in a circular way.*

The referent of a concept is requested during the computation of that same referent.

EXAMPLE

In (81) the form "(1+ (>> referent of impossible-number))" needs to be evaluated to compute the referent of Impossible-Number.

```
(defconcept IMPOSSIBLE-NUMBER                                    81
  (a number
    (definition [form (1+ (>> referent))])))
```

REMEDY

The circularity is often not so straightforward as in the preceding example. Usually it is the result of bad modeling, such that the only solution is to reconsider the model.

This problem can also occur with mutating subjects, in particular if one wants to change the filler of a subject such that the new filler is computed on the basis of the previous one. In that case, the problem can often be solved by first catching the previous filler within a concept-variable and then computing the new-filler making use of this variable.

EXAMPLE

```
(defconcept STACK
  (a organization
    ((elements
       (a mutating-subject
         (initial-value [concept-list ()])))
    (top (>> first elements))
    ((pop
       (an action-subject
          (definition [form (clet ((?previous (>> elements)))
                                  (>> (mutate-to (>> rest of ?previous))
                                      handler elements))])))))))
```

INCOMPLETE INSTANTIATION

MESSAGE

> *Error: Impossible to compute the context of XXX, due to incomplete instantiation.*

Incomplete Instantiation is a situation that sometimes leads to a very subtle error. It clearly exposes the edges of the inheritance mechanism and the lexical-scope principle.

A concept is incompletely instantiated if the context of some of its inherited subjects cannot be computed. It generates an error when a relative-definite-description attempts to access the lost information.

EXAMPLE

```
(defconcept CAR
  (owner (a person
           (insured-by (>> company insurance-contract))))
  (driver (>> owner))
  (insurance-contract
    (a contract
      (company (an insurance-company))
      (insured (>> owner)))))


(defconcept INVALID-CONTRACT-23                                    82
  (a (>> insurance-contract of car)
    (invalidity-reason reached-end-date)
    (company Drive-Carefully-Inc)))


-->  (>> insured of invalid-contract-23)
<--  Error:  Impossible to compute the context of "(>> owner)",
             due to incomplete instantiation.
```

The insured of the insurance-contract of a car is the owner of that car. The concept Invalid-Contract-23 however is an instance of the insurance-contract of Car. So when asking for the insured of Invalid-Contract-23, we need to know the owner of the car Invalid-Contract-23 is the insurance-contract of, and this car is not known. The problem is caused by the fact that a specialization is made of only a part of the description of Car. Hence references to the global description cannot be properly

interpreted.

REMEDY

This problem cannot occur if all descriptions of type-subjects are concept-name-descriptions or more general when all types are named concepts. If the type of a concept is not a named concept but a concept which is described as the filler of a subject of some other concept, one cannot use inherited relative-requests. This remark does not apply for parallel types.

Sometimes the problem can be solved by making first a specialization of the complete structure and afterwards using a defconcept-description to take that part of this specialization one needs. For the above example, this would be the following:

```
(clet ((?dummy-car (a car
                     (insurance-contract
                       (invalidity-reason reached-end-date)
                       (company Drive-Carefully-Inc)))))
  (defconcept INVALID-CONTRACT-23
    (>> insurance-contract of ?dummy-car)))
```

ERROR-CONCEPT-PNAME

When concepts have special pnames, it is possible that an error occurs during the computation of a pname. Such an error is very hard to debug, because it happens over and over again while the debugger is giving information. To avoid this, the system catches this error and prints the concept as is shown in (83).

```
<error-concept-pname #25>                                    83
```

REMEDY

To actually see what went wrong, one must ask for the referent of the pname of the meta-interpreter explicitly.

CIRCULAR INHERITANCE

MESSAGE

Error: Circular Inheritance. The concept XXX has no subject YYY.

For inheritance to work properly, the type-tree must be a strict hierarchy. However, since types are user-definable subjects, it is possible that they are defined in a circular way. This gives in principle no problems if all requested subjects are defined for some concept in the circle. If not, the above error occurs.

REMEDY

If this error occurs, one can either break up the circle or define the missing subject. Beware that the meta-interpreter-type subject is often implicitly requested by the KRS interpreter.

5.2.3. Some General Remarks.

- When an error occurs during the computation of a referent, it may help to ask for the referent of the definition of the same concept. If the error still occurs, it must occur during the computation of the definition or during the computation of this definition's referent. If not, the resulting LISP-form can be used to find-out what goes wrong.

```
--> (>> referent of foo)
<-- Error: ...

--> (>> referent definition of foo)
<-- (some lisp form)
```

- When an error occurs during the computation of a subject's filler, this is a special case of the previous problem. Ask for the referent of the definition of the subject-concept.

```
--> (>> bar of foo)
<-- Error: ...

--> (>> referent definition handler bar of foo)
<-- (some lisp form)
```

- When a relative-request occurs in a definition, its context can be searched for by asking for the context of the definition's meta-interpreter.

```
--> (>> context meta-interpreter definition handler bar of foo)
<-- <foo>
```

- When one does not know which subject-name to use in a request, the possible subject-names can be asked to the concept's meta-interpreter.

```
--> (>> referent subject-names meta-interpreter of foo)
<-- (meta-interpreter meta-interpreter-type bar bazz referent definition)
```

5.3. The Tracer.

The tracer allows one to follow the interpretation process. There are three primitive tracer-modes, which can be combined.

- Mode 1 shows all requests and their results.
- Mode 2 shows which subject-fillers are requested.
- Mode 3 gives information about requested referents.

FUNCTIONS

tracer-0 *[function]*

turns all tracer-modes off.

tracer *modes &optional step* *[function]*

Modes is a list of numbers between 1 and 3. All specified modes are turned on. *Step* is a flag. When set to T, the stepper is activated too.

>>-trace *modes @subject-name-sequence &optional OF concept-description* *[function]*

responds to the request while all tracer-modes specified in *modes* are on. *modes* can be a list or one single mode.

EXAMPLES

```
(defconcept CAR
  (owner (a person
           (insured-by (>> company insurance-contract))))
  (driver (>> owner))
  (insurance-contract
    (a contract
      (company (an insurance-company))
      (insured (>> owner)))))


(defconcept JOHNS-CAR
  (a car
    (owner john)))


(>>-trace 1 insured insurance-contract of johns-car)

1 Ev: (>> insured insurance-contract of johns-car)
2  Ev: (>> owner) in context <johns-car>
2  Rt: (>> owner) yields <john>
1 Rt: (>> insured insurance-contract of johns-car) yields <john>

(>>-trace 1 insured insurance-contract of johns-car)

1 Ev: (>> insured insurance-contract of johns-car)
1 Rt: (>> insured insurance-contract of johns-car) yields <john>
```

The prefix "Ev" means 'evaluating', "Rt" means 'returns'. The output indents one column whenever a new description is found before the previous one finishes.

Notice that the second time we ask the same question, less information is given. This is because referents have been cached.

```
(>>-trace 2 insured insurance-contract of johns-car)

1 Su: definition of <johns-car name>, local
1 Su: insurance-contract of <johns-car>, inherited
2  Su: definition of <subject #41>, inherited
1 Su: insured of <insurance-contract #42>, local
2  Su: owner of <johns-car>, local

(>>-trace 2 insured insurance-contract of johns-car)

1 Su: insurance-contract of <johns-car>, inherited
1 Su: insured of <insurance-contract #42>, local
```

The prefix "Su" means 'find subject'. The output indents one column if a new subject is retrieved while computing the referent of the previous subject.

Notice that in the second request, there are no referents computed any more, and hence no second level subject requests are needed.

```
(>>-trace 3 insured insurance-contract of johns-car)

1 Rf: <johns-car name>, computing
2  Rf: <form ...>, cached: (make- ...)
1 Ca: <johns-car>, referent of <johns-car name>
1 Rf: <insurance-contract subject>, computing
2  Rf: <form ...>, cached: (make- ...)
1 Ca: <contract #43>, referent of <insurance-contract subject>
1 Rf: <insured subject>, computing
2  Rf: <form ...>, cached: (>> owner)
2  Rf: <owner subject>, computing
3   Rf: <form ...>, cached: (>> of john)
2  Ca: <john>, referent of <owner subject>
1 Ca: <john>, referent of <insured subject>

(>>-trace 3 insured insurance-contract of johns-car)

1 Rf: <johns-car name>, cached: <johns-car>
1 Rf: <insurance-contract subject>, cached: <contract #43>
1 Rf: <insured subject>, cached: <john>
```

The prefix "Rf" means 'get referent', "Ca" means 'cache referent'. The output indents one column if a new referent is asked while computing the previous referent.

```
(>>-trace (1 2 3) insured insurance-contract of johns-car)

1 Ev: (>> insured insurance-contract of johns-car)
2  Rf: <johns-car name>, computing
3   Su: definition of <johns-car name>, local
3   Rf: <form ...>, cached: (make- ...)
2  Ca: <johns-car>, referent of <johns-car name>
2  Su: insurance-contract of <johns-car>, inherited
3   Rf: <insurance-contract subject>, computing
4    Su: definition of <insurance-contract subject>, inherited
4    Rf: <form ...>, cached: (make- ...)
3   Ca: <contract #43>, referent of <insurance-contract subject>
2  Su: insured of <insurance-contract #42>, local
3   Rf: <insured subject>, computing
4    Rf: <form ...>, cached: (>> owner)
4    Ev: (>> owner) in context <johns-car>
5     Su: owner of <johns-car>, local
6      Rf: <owner subject>, computing
7       Rf: <form ...>, cached: (>> of john)
6      Ca: <john>, referent of <owner subject>
4    Rt: (>> owner) yields <john>
3   Ca: <john>, referent of <insured subject>
1 Rt: (>> insured insurance-contract of johns-car) yields <john>


(>>-trace (1 2 3) insured insurance-contract of johns-car)

1 Ev: (>> insured insurance-contract of johns-car)
2  Rf: <johns-car name>, cached : <johns-car>
2  Su: insurance-contract of <johns-car>, inherited
3   Rf: <insurance-contract subject>, cached: <contract #43>
2  Su: insured of <insurance-contract #42>, local
3   Rf: <insured subject>, cached: <john>
1 Rt: (>> insured insurance-contract of johns-car) yields <john>
```

```
(tracer '(1 2 3) t)
(>> insured insurance-contract of johns-car

1 Ev: (>> insured insurance-contract of johns-car)
  tr (y/n)? y

2  Rf: <johns-car name>, computing
   tr (y/n)? n

2  Ca: <johns-car>, referent of <johns-car name>
2  Su: insurance-contract of <johns-car>, inherited
3   Rf: <insurance-contract subject>, computing
    tr (y/n)? n
 3   Ca: <contract #43>, referent of <insurance-contract subject>
2  Su: insured of <insurance-contract #42>, local
3   Rf: <insured subject>, computing
    tr (y/n)? y
 4    Rf: <form ...>, cached: (>> owner)
4    Ev: (>> owner) in context <johns-car>
     tr (y/n)? y
5     Su: owner of <johns-car>, local
6      Rf: <owner subject>, computing
       tr (y/n)? y

7       Rf: <form ...>, cached: (>> of john)
6      Ca: <john>, referent of <owner subject>
4    Rt: (>> owner) yields <john>
3   Ca: <john>, referent of <insured subject>
1 Rt: (>> insured insurance-contract of johns-car) yields <john>
```

6. LIBRARIES.

6.1. Introduction.

The concept-libraries are an important aspect of the KRS programming environment. By providing a workable set of libraries, we encourage the KRS programmer to focus his efforts on the problem domain. With the lazy evaluation mechanism, vast libraries can be provided, without consuming too much memory.

6.2. The Data-Concept Library.

6.2.1. Booleans.

Booleans can be used in two ways. First, the user can create instances of the concept Boolean whenever a boolean is needed. Second, the user can make use of the predefined concepts True and False.

Where possible, we choose for the second solution. It creates less concepts, makes better use of caching, and has a more readable outcome.

(84) shows how instances of the concept Boolean can be used. (85) illustrates the use of the concepts True and False.

```
(defsubject ADULT-P of PERSON                                          84
  (a boolean
    (definition
      [form (>= (>> referent age) 21)])))


(define-subject ADULT-P of PERSON                                      85
  (a subject
    (definition
      [form (if (>= (>> referent age) 21)
                (>> of true)
                (>> of false))])))
```

The concept Unknown is a Boolean with unknown referent. It cannot be used in combination with arbitrary booleans. When used in combination with the booleans True and False, the subjects "not", "and" and "or" take into account the possibility that one of them is unknown, as illustrated in (86).

```
-->  (>> not of unknown)                                              86
<--  <unknown>

-->  (>> (and unknown) of false)
<--  <false>

-->  (>> (and unknown) of true)
<--  <unknown>
```

To facilitate conditionals in LISP when working with the concepts True, False and Unknown, the subjects "true-p", "false-p" and "unknown-p" are added. These

subjects return t or nil.[6] An example of their use is given in (87).

```
(defconcept LIST-OF-POSSIBILITIES                                    87
  (a concept-list
    ((all-satisfied
       (a subject
         (definition
           [form (or (cdolist (?element (>> referent))
                       (when (or (>> false-p satisfied of ?element)
                                 (>> unknown-p satisfied of ?element))
                         (return (>> of false))))
                     (>> of false))])))))))
```

The definitions of the not- and- and or-subjects of Boolean use the referent of the boolean. The boolean Unknown however has no referent. To make it possible to work with a concept Unknown, these subjects are redefined for the concepts True, False and Unknown, such that they do not ask for referents any more.

Boolean *[concept]*

A concept with type Data-Concept.

not *[subject of Boolean]*

If the referent of the boolean is nil, this subject returns <true>, otherwise <false>. Subject-definition:

```
(if (>> referent)
    (>> of false)
    (>> of true))
```

and *?boolean* *[subject of Boolean]*

If either one of the referents of this boolean and ?boolean is NIL, this subject returns <false>, otherwise <true>. Subject-definition:

```
(if (and (>> referent) (>> referent of ?boolean))
    (>> of true)
    (>> of false))
```

or *?boolean* *[subject of Boolean]*

If both the referents of this boolean and ?boolean are nil, this subject returns <false>, otherwise <true>. Subject-definition:

```
(if (or (>> referent) (>> referent of ?boolean))
    (>> of true)
    (>> of false))
```

[6] Though it is against the KRS philosophy to let subjects return lisp-objects, we think it is justified in this case. These subjects represent tests which are very often made from within a piece of LISP code. In fact, the subject "true-p" replaces the test '(equal *the-boolean* (>> of true)'.

True *[concept]*

A boolean with referent t.

False *[concept]*

A boolean with referent nil.

Unknown *[concept]*

A boolean.

true-p *[subject of True, False, Unknown]*

returns t for True, nil for False and Unknown.

false-p *[subject of True, False, Unknown]*

returns t for False, nil for True and Unknown.

unknown-p *[subject of True, False, Unknown]*

returns t for Unknown, nil for True and False.

not *[subject of True]*

returns <false>.

and *?boolean* *[subject of True]*

returns ?boolean.

or *?boolean* *[subject of True]*

returns <true>.

not *[subject of True]*

returns <true>.

and *?boolean* *[subject of False]*

returns <false>

or *?boolean* *[subject of False]*

returns ?boolean.

not *[subject of Unknown]*

returns Unknown.

and *?boolean* *[subject of Unknown]*

If ?boolean is <false>, this subject returns <false>, otherwise <unknown>. Subject-definition:

```
(if (>> false-p of ?boolean)
    (>> of false)
    (>> of unknown))
```

or *?boolean* *[subject of Unknown]*

If ?boolean is <true>, this subject returns <true>, otherwise <unknown>. Subject-definition:

```
(if (>> true-p of ?boolean)
    (>> of true)
    (>> of unknown))
```

6.2.2. Numbers.

Numbers contain subjects to do simple arithmetic. (88) shows a small example of the use of the number-library.

```
(defconcept CURRENT-YEAR                                                    88
  [number 1987])

(defconcept PERSON
  (birthyear (a number))
  (age (>> (subtract (>> birthyear)) of current-year)
  (adult-p (>> not (smaller-than [number 21]) age)))
```

Number *[concept]*

A concept with type Data-Concept.

zero-p *[subject of Number]*

Returns <true> if the referent is zero, otherwise <false>.

positive-p *[subject of Number]*

Returns <true> if the referent is positive, otherwise <false>.

negative-p *[subject of Number]*

Returns <true> if the referent is negative, otherwise <false>.

equal *?to* *[subject of Number]*

Returns <true> if the referent and the referent of ?to are equal, otherwise <false>.

smaller-than *?smaller-than-what* *[subject of Number]*

Returns <true> if the referent is smaller than the referent of ?smaller-than-what, <false> otherwise.

larger-than *?larger-than-what* *[subject of Number]*

Returns <true> if the referent is larger than the referent of ?larger-than-what, <false> otherwise.

next *[subject of Number]*

Returns a number with referent one more than referent.

previous *[subject of Number]*

Returns a number with referent one less than referent.

add *?add-what* *[subject of Number]*

Returns a number with referent the sum of referent and the referent of ?add-what.

subtract *?subtract-what* *[subject of Number]*

Returns a number with referent the difference between referent and the referent of ?subtract-what.

times *?multiply-by* *[subject of Number]*

Returns a number with referent the result of the multiplication of referent with the referent of ?multiply-by.

divide *?divide-by* *[subject of Number]*

Returns a number with referent the result of the division of referent by the referent of ?divide-by.

odd-p *[subject of Number]*

Returns <true> if referent is odd, <false> otherwise.

even-p *[subject of Number]*

Returns <true> if referent is even, <false> otherwise.

6.2.3. Lists, Concept-Lists and Sequences.

A List is a concept representing a lisp-list. A Concept-List is a concept representing a list of concepts. A Sequence is a Concept-List. These three types are distinguished because the LISP-reader interprets them differently when defined with rectangular-brackets. The special-instance-creator for List is the normal Data-Concept special-instance-creator. The special-instance-creator for Concept-List interprets every element of the list as a concept-description to be parsed into a concept. The special-instance-creator for Sequence does not need the lisp-brackets around the list.

EXAMPLES

```
--> [list (a b c)]
<-- <list (a b c)>

--> (>> referent of [list (a b c)])
<-- (a b c)

--> [concept-list (john (a person) (>> age of john))]
<-- <concept-list (<john> <person #23> <number 26>)>

--> (>> referent of [concept-list (john (a person) (>> age of john))])
<-- (<john> <person #23> <number 26>)

--> [sequence john (a person) (>> age of john)]
<-- <sequence <john> <person #23> <number 26>>

--> (>> referent of [sequence john (a person) (>> age of john)])
<-- (<john> <person #23> <number 26>)
```

When defining generic operations over lists, we have to consider what the type of the outcome must be. If we would use the type List (89), we can run into problems, like the one illustrated in (90) with the operation reverse.

```
(defconcept LIST                                                    89
  (a data-concept
    ...
    (reverse
      (a list
        (definition
          [form (reverse (>> referent))]))))
    ...))


-->  (>> reverse of [concept-list (john mary frank)])               90
<--  <list (<frank> <mary> <john>)>

-->  (>> last reverse of [concept-list (john mary frank)])
     ERROR: The concept List has no subject LAST.
```

To solve this problem, generic operations return in general a subtype of the type of the original concept (91).

```
(defconcept LIST                                                    91
  (a data-concept
    ...
    (reverse
      (a (>> type)
        (definition
          [form (reverse (>> referent))]))))
    ...))
```

That this solution often works is illustrated in (92). However, (93) shows that it can give wrong results too.

```
-->  (>> reverse of [concept-list (john mary frank)])               92
<--  <concept-list (<frank> <mary> <john>)>

-->  (>> last reverse of [concept-list (john mary frank)])
<--  <frank>


-->  (>> reverse of [ordered-number-list ([number 2] [number 6] [number 10])]) 93
<--  <ordered-number-list (<number 10> <number 6> <number 2>)>
```

List *[concept]*

A concept with type Data-Concept.

empty-p *[subject of List]*

Returns <true> if the list is nil, <false> otherwise.

equal *?to* *[subject of List]*

Returns <true> if the list and the referent of ?to are equal.

rest *[subject of List]*

Returns a concept with the same type and with referent the cdr of the list.

reverse *[subject of List]*

Returns a concept with the same type and with referent the reverse of the list.

length *[subject of List]*

Returns a number with referent the length of the list.

cons *?element* *[subject of List]*

Returns a concept with the same type and with referent the result of consing ?element onto the list.

append *?list* *[subject of List]*

Returns a concept with the same type and with referent the result of appending the list with the referent of ?list.

Concept-List *[concept]*

A concept with type List. A Concept-List represents a list of concepts.

first *[subject of Concept-List]*

Returns the first concept out of the concept-list.

second *[subject of Concept-List]*

Returns the second concept out of the concept-list.

third *[subject of Concept-List]*

Returns the third concept out of the concept-list.

fourth *[subject of Concept-List]*

Returns the fourth concept out of the concept-list.

fifth *[subject of Concept-List]*

Returns the fifth concept out of the concept-list.

last *[subject of Concept-List]*

Returns the last concept out of the concept-list.

nth *?number* *[subject of Concept-List]*

Returns the nth element out of the the concept-list.

member *?element* *[subject of Concept-List]*

Returns <true> if ?element is member of the concept-list, otherwise <false>.

Sequence *[concept]*

A concept with type Concept-List.

6.2.4. Functions and Clambda's.

Functions are concepts with LISP-functions as referent. A function concept can be created as in (94). A Clambda is a subtype of Function. It has a special-instance-creator to define a function as illustrated in (95).

```
[function plus]                                                    94

-->  (>> referent of [function plus])
<--  <compiled-function 'Plus>


[clambda (?number) (>> next of ?number)]                           95

-->  (>> referent of ~*)
<--  (clambda (?number) (>> next of ?number))
```

Function *[concept]*

A concept with type Data-Concept.

Clambda *[concept]*

A concept with type Function. An instance of Clambda is created by:

```
[clambda (krs-variable-list) body]
```

apply *?element* *[subject of Clambda]*

The subject "apply" can be used to apply a clambda with one argument. Subject-definition:

```
(funcall (>> referent) ?element)
```

An example of the use of Clambda's.

```
(defconcept CONSTRAINT                                             96
  (a clambda))

(defconcept CLASS
  (membership-constraint (a constraint))
  ((belongs-to
     (a subject
       (arguments [args ?element])
       (definition
         [form (>> (apply ?element) membership-constraint)])))))
```

6.2.5. Various Concepts.

Form *[concept]*

A concept with type Data-Concept. Represents lisp-forms.

Symbol *[concept]*

A concept with type Data-Concept. Represents lisp-symbols.

String *[concept]*

A concept with type Data-Concept. Represents lisp-strings.

Eager-Data-Concept-Meta-Interpreter *[concept]*

Redefines the pname of Data-Concept-Meta-Interpreter such that the referent is computed before printing. An example of its use it given in (97).

```
(defconcept CURRENCY                                                    97
  (a number
    (meta-interpreter-type eager-data-concept-meta-interpreter)
    (rate (a number))
    (in-belgian-franks
      (>> (times (>> rate))))))

(defconcept DOLLAR
  (a currency
    (rate [number 34])))

(defconcept BELGIAN-FRANKS
  (a currency
    (rate [number 1])))

-->  (>> in-belgian-franks of [dollar 5])
<--  <belgian-franks 170>
```

6.3. The Cliche Library

Cliches are inspired by the work of Viviane Jonckers [Jonckers87] and by recent tendencies in programming environments [Rich78, Waters81, Abbott87].

A cliche is a subtype of Form. It describes a particular computation at a conceptual-level. The definition of a cliche constructs a LISP-form making use of the information in the cliche's subjects.

Transform *[concept]*

(transform-what (a list)) (transformer (a form)) (transform-with [variable ?element])

A Transform describes a lisp-form to transform a given list (the transform-what) into a new list. The elements of the new list are computed by evaluating the transformer, once for the concept-pointer in "transform-with" bound to every element in the input list. The default concept-pointer is "?element".

EXAMPLE

```
(defconcept FAMILY
  (children (a person-list))
  (ages-of-children
    (a number-list
      (definition
        (a transform
          (transform-what (>> children))
          (transform-with [variable ?one-child])
          (transformer [form (>> age of ?one-child)])))))))
```

Transform-Subject *[concept]*

(transform-what (a list)) (transformer (a form)) (transform-with [variable ?element]) (transform-in (>> type transform-what))

The Transform-Subject is a subject. It creates an instance of its transform-in, with definition a transform. The default transform-in is the type of transform-what.

EXAMPLE

```
(defconcept FAMILY
  (children (a person-list))
  ((ages-of-children
     (a transform-subject
       (transform-in number-list)
       (transform-what (>> children))
       (transform-with [variable ?one-child])
       (transformer [form (>> age of ?one-child)])))))))
```

Transform-And-Collect *[concept]*

(transform-what (a list)) (transformer (a form)) (transform-with [variable ?element])

The Transform-And-Collect behaves like the transform, except that the transformer does not return individual elements, but a list of elements. All the elements of the sublists are collected in one list.

EXAMPLE

```
(defconcept PERSON
  (children (a list-of-persons))
  (grandchildren
    (a list-of-persons
      (definition
        (a transform-and-collect
          (transform-what (>> children))
          (transform-with [variable ?one-child])
          (transformer [form (>> referent children of ?one-child)])))))))
```

Transform-And-Collect-Subject *[concept]*

(transform-what (a list)) (transformer (a form)) (transform-with [variable ?element]) (transform-in (>> type transform-what))

The Transform-And-Collect-Subject is a subject. It creates an instance of its transform-in, with definition a transform-and-collect. The default transform-in is the type of transform-what.

EXAMPLE

```
(defconcept PERSON
  (children (a list-of-persons))
  ((grandchildren
     (a transform-and-collect-subject
       (transform-what (>> children))
       (transform-with [variable ?one-child])
       (transformer [form (>> referent children of ?one-child)])))))
```

Select *(select-from (a concept-list)) (selector (a form)) (select-with [variable ?element])* *[concept]*

A select describes a form which returns the first element of a concept-list for which the selector returns True. The selector is a lisp-form which is evaluated while the concept-pointer in the select-with is bound to an element. The default select-with is the concept-pointer ?element.

EXAMPLE

```
(defconcept FAMILY
  (children (a list-of-persons))
  ((an-adult-child
     (a subject
       (definition
         (a select
           (select-from (>> children))
           (select-with [variable ?one-child])
           (selector [form (>> adult-p of ?one-child)])))))))
```

Select-Subject *[concept]*

(select-from (a concept-list)) (selector (a form)) (select-with [variable ?element])

A select-subject is a subject with definition a select.

EXAMPLE

```
(defconcept FAMILY
  (children (a list-of-persons))
  ((an-adult-child
     (a select-subject
       (select-from (>> children))
       (select-with [variable ?one-child])
       (selector [form (>> adult-p of ?one-child)]))))
```

Select-Best *(select-from (a concept-list)) (selector (a form)) (select-with [variable ?element]) [concept]*

A select-best describes a form which returns that element out of a concept-list for which the selector returns the highest number. The selector is a LISP-form which returns a number if evaluated when the concept-variable in the select-with is bound to an element of the list. The default select-with is the concept-pointer ?element.

EXAMPLE

```
(defconcept FAMILY
  (children (a list-of-persons))
  ((oldest-child
     (a subject
       (definition
         (a select-best
           (select-from (>> children))
           (select-with [variable ?one-child])
           (selector [form (>> age of ?one-child)])))))))
```

Select-Best-Subject *[concept]*

(select-from (a concept-list)) (selector (a form)) (select-with [variable ?element])

A select-best-subject is a subject with definition a select-best.

EXAMPLE

```
(defconcept FAMILY
  (children (a list-of-persons))
  ((oldest-child
     (a select-best-subject
       (select-from (>> children))
       (select-with [variable ?one-child])
       (selector [form (>> age of ?one-child)])))))
```

Filter *(filter-over (a list)) (filter-predicate (a form)) (filter-with [variable ?element])* *[concept]*

A Filter describes a lisp-form to filter a given list (the filter-over) into a new list. The elements of the new list are computed by evaluating the filter-predicate, once for the concept-pointer in "filter-with" bound to every element in the input list. All elements of the initial list for which this predicate returns the concept True are preserved. The default concept-pointer is "?element".

EXAMPLE

```
(defconcept FAMILY
  (children (a person-list))
  (adult-children
    (a person-list
      (definition
        (a filter
          (filter-over (>> children))
          (filter-with [variable ?one-child])
          (filter-predicate [form (>> adult-p of ?one-child)])))))))
```

Filter-Subject *[concept]*

(filter-over (a list)) (filter-predicate (a form)) (filter-with [variable ?element]) (filter-in (>> type filter-over))

The Filter-Subject is a subject. It creates an instance of its filter-in, with definition a filter. The default filter-in is the type of filter-over.

EXAMPLE

```
(defconcept FAMILY
  (children (a person-list))
  ((adult-children
     (a filter-subject
       (filter-over (>> children))
       (filter-with [variable ?one-child])
       (filter-predicate [form (>> adult-p of ?one-child)])))))
```

6.4. The Subject Library.

The subject library contains abstract subject-concepts, which are subtypes of Subject. They have a predefined definition which can be used by explicit subject-concepts which are created as a subtype of an abstract subject-concept.

Delegating-Subject *delegate-to* *[concept]*

A delegating-subject delegates a question to another concept. This other concept is the filler of the delegate-to-subject.

EXAMPLE

```
(defconcept PERSON
  (profession (a profession))
  ((salary
     (a delegating-subject
       (delegate-to (>> profession))))))

(defconcept JOHN
  (a person
    (profession teacher)))

(defconcept TEACHER
  (a profession
   (salary [bf 35000])))

-->   (>> salary of john)
<--   <bf 35000>
```

Non-Caching-Subject *definition* *[concept]*

A non-caching-subject is a subject that never caches its referent. It is particularly useful when the evaluation of the definition has side-effects. A synonym is Action-Subject.

Action-Subject *definition* *[concept]*

Action-subject is a synonym for Non-Caching-Subject.

Mutating-Subject *initial-value* *[concept]*

A mutating-subject is a subject-concept whose filler can be dynamically changed. Its initial-value-subject holds the initial filler. Changing the subject's referent is done with the mutate-to-subject of Mutating-Subject.

mutate-to *?new-value* *[subject of Mutating-Subject]*

Changes the filler of the mutating-subject to ?new-value. All cached referents which depend on the previous filler are withdrawn.

EXAMPLE

```
(defconcept NODE
  ((unexplored-p
     (a mutating-subject
       (initial-value true))))
  ((cost
     (a subject
       (definition
         [form (if (>> true-p unexplored-p)
                   (>> of low)
                   (>> of high))])))))


-->  (>> cost of node)
<--  <low>

-->  (>> (mutate-to false) handler unexplored-p of node)
<--  <done>

-->  (>> cost of node)
<--  <high>
```

REMARK

Notice that the mutate-to-subject is a subject of the subject-concept (and not of its filler).

Fixed-Subject *definition* *[concept]*

A fixed-subject is a subject whose referent will never be denied, once it is computed.

Eager-Subject *definition* *[concept]*

An eager-subject is a subject that recomputes its referent automatically whenever it is withdrawn. The referent is **not** automatically computed when the subject is created!

Always-Caching-Subject *definition* *[concept]*

Normal subjects only cache if the computation of their referent did not change something on which the computed value depends. Always-Caching-Subjects also cache in this situation.

Memorizing-Subject *[concept]*

arguments (body (a form)) selector

Subjects with arguments do not cache their fillers. A memorizing-subject is a subject with arguments that stores the values it computes in a hash-table, together with the arguments used to compute them.

In its simpliest form, a memorizing-subject does not recompute a filler if the question was asked before with the same arguments.

It is also possible to customize the way the memorizing-subject determines whether a previous result can be returned. Therefore, it must have a selector-subject. This subject must have the same arguments as the memorizing-subject itself. The selector-subject returns the key which is stored together with the result and which is used to reselect the result on a later request.

EXAMPLE

In (98) we assume that the computation of the protection-factor of a sunburn lotion is expensive. Since there are only a few locations toerists typically go to, it can pay of to remember previous computations. The first time the request in (99) is evaluated, it is computed. The second time the result can be returned without computation.

```
(defconcept VACATION-ORGANISER                                          98
  (location (a location))
  (sunburn-lotion (>> (protection-factor (>> location))
                      of sunburn-lotion-specialist)))

(defconcept SUNBURN-LOTION-SPECIALIST
  ((protection-factor
     (a memorizing-subject
       (arguments [args ?location])
       (body
         [form (some-hairy-function-to-compute-protection-factor
                 ?location)])))))


-->   (>> (protection-factor porto-rico) of sunburn-lotion-specialist)      99
<--   <type 5>
-->   (>> (protection-factor porto-rico) of sunburn-lotion-specialist)
-->   <type 5>
```

We can also assume that the protection-factor of a sunburn lotion only depends on the climate to the location. Hence, if the filler is requested with two different locations which have the same climate, it can be arranged that the result will not be recomputed the second time. This is the role of the selector-subject (100).

```
(defconcept SUNBURN-LOTION-SPECIALIST                                   100
  ((protection-factor
     (a memorizing-subject
       (arguments [args ?location])
       ((selector (?location) (>> climate of ?location)))
       (body
         [form (some-hairy-function-to-compute-protection-factor
                 ?location)])))))


-->   (>> (protection-factor porto-rico) of sunburn-lotion-specialist)      101
<--   <type 5>
-->   (>> (protection-factor jamaica) of sunburn-lotion-specialist)
<--   <type 5>
```

In this example, the protection-factor is not recomputed the second time, if and only if the climates of Porto-Rico and Jamaica are the same.

REMARK
Obviously, when one expects to have a lot of questions mostly with different arguments, it is not a good idea to use a memorizing-subject. The list of stored values would become very large and search in that list would become slow. Moreover, since one uses mostly different arguments, the chance for the search to be successful would be small.

REMARK
Memorizing-Subjects are sometimes used to avoid creating new concepts. If a subject with arguments can return the same concepts on different requests, there will be more benefit from the caching mechanism.

6.5. The Concept-Library.

6.5.1. Demons.

A 'Demon' is a concept that fires when it detects a mutation and its fire-when condition is true.

Demon *(fire-action (a form) (fire-when true) (deactivate (a boolean)) monitoring* *[concept]*

A demon can be activated by asking for the filler of its activate-subject. Once activated, it tries to fire whenever the filler of the monitoring-subject changes. It tries to fire by checking its fire-when condition. If this is true, it fires by evaluating its fire-action, and checks whether it has to remain active by checking its deactivate-condition. Once the monitor is deactivated, it can never fire again.

EXAMPLE
After its activation, the demon in (102) prints the string "Good Morning" in a message-window whenever the day of Current-Date changes, except on a monday. The demon will stop doing this when the year of Current-Year becomes 1989.

```
(defconcept GOOD-MORNING-DEMON                                          102
  (a demon
    (monitoring (>> day of current-date))
    (fire-action [form (>> (print [string "Good Morning"]) of message-window)])
    (fire-when (>> not (equal monday) weekday of current-date))
    (deactivate (>> (equal [year 1989]) year of current-date))))

(>> activate of good-morning-demon)
```

6.5.2. Counters.

Counter *(initial-value [number 0])* *[concept]*

A counter is a number with an initial-value. Its referent can be changed with the subjects set and increment

set *?new* *[subject of Counter]*

changes counter such that its new referent it equal to the referent of ?new.

increment *[subject of Counter]*

changes counter such that its new referent is its old referent plus one.

6.6. The FPPD-Library.

Concepts in the FPPD library have FPPD propositions as referent (see chapter 2).

Proposition *[concept]*

a data-concept.

Primitive-Proposition *initial-value* *[concept]*

a concept with referent an FPPD primitive-proposition. *Initial-value is the initial-value of this primitive-proposition.*

Relative-Proposition *(function (a function)) (context (a list))* *[concept]*

a concept with referent an FPPD relative-proposition.

value *[subject of Proposition]*

returns the value of the proposition's referent. *Function* and *context* are concepts whose referents are respectively the function and context of this relative-proposition.

redefine-value *?new-value* *[subject of Primitive-Proposition]*

redefines the value of its referent.

THE KRS MANUAL: INDEX

Chapter Two

THE KRS IMPLEMENTATION.

INTRODUCTION.

Chapter two describes the implementation of KRS release 3.0, specified in chapter one. Two important constraints have influenced the design decisions for this implementation.

EFFICIENCY

Early KRS implementations were all computationally expensive and slow. This did not come as a surprise in view of the size of the KRS concept-graph and the complexity of the computation. For target machines different from the Symbolics, those implementations ran on the edge of acceptable run-time efficiency. Several implementation efforts have continuously improved KRS performance leading to today's implementation which runs on Symbolics, Sun, Mac-II, etc.

PORTABILITY

A system running on a Symbolics has inherently a small group of potential users. For this reason, we always kept in mind the necessity of portability. In this context, KRS was initially implemented in ZETALISP on the Symbolics, using only those ZETALISP features which where likely to occur in COMMONLISP (which was being developed at that time). Consequently, unlike the implementation of its ancestor language KRS-83, this implementation is not based on FLAVORS. This did not make it easier, but it eventually turned out to be a profitable decision. The ZETALISP implementation has been straightforwardly translated to COMMONLISP, and this translation ran almost without modifications in GENERA COMMONLISP [Symbolics87] on the Symbolics, in KYOTO COMMONLISP [Yuasa84] under UNIX, and in ALLEGRO COMMONLISP [Byers87] on the Mac-II.

Besides staying within the efficiency and portability constraints described above, the technical challenge of this implementation was to construct a manageable program. Many programs consist of layers of which the lower ones are practically independent of the higher ones. In general the lower layers provide tools to facilitate the implementation of the higher layers. A similar architecture is virtually impossible for the KRS implementation. All KRS features such as referent computation, inheritance, lazy

evaluation, caching and consistency maintenance, are carefully chosen in function of all the others ones, such that none of them can be omitted without having the whole collapse. This reflects in an implementation where each component heavily depends on many others.

Various techniques, originally developed for KRS, eventually turn out to be relevant in a broader context. They can be extracted from the KRS implementation and applied for the implementation of different products. They include the data-dependency mechanism FPPD, the task-processor, and the inheritance implementation.

Of those, FPPD is the most innovative and most powerful component. It is a module which can be directly used independently from KRS. FPPD shows how very large data-dependency networks can be employed in computationally heavy systems to (i) augment the efficiency, (ii) increment the expressiveness, and (iii) support data-directed programming. FPPD was inspired by Truth Maintenance Systems (TMS's) by which data-dependencies were introduced in Artificial Intelligence programming. Although the functionality of FPPD is much smaller than the functionality of a TMS, the use of FPPD can lead to a more interesting behavior. There are basically two reasons for this. First FPPD operates efficiently on a very large scale, where all existing TMS implementations are far too computationally intensive to operate on large networks. Second, FPPD constructs, destructs and changes its dependency-network itself, while a TMS is told about existing justifications. This makes it possible to work with far more complex graphs.

OVERVIEW

The four large components of the KRS implementation described in this chapter are FPPD, the data structures, the task-processor and inheritance. Finally also the KRS environment is specified. It is a part of the LISP environment which has a specific function for the creation of concepts.

All of those descriptions are organized as follows. They start with a description of the problem, i.e. a description of the role of this component within the whole and of the constraints under which it must operate. Then, a sketch of the solution is given. This is an overall sketch of the kind of solution we choose and the context in which it can be situated. Third we give a functional description. This is sort of an interface to the component viewed as a black box. This is followed by a description of the implementation at the LISP-code level. Finally an example of its usage is given.

Section one describes the system FPPD. Section two introduces the data-structures representing the KRS concept-graph. In section three we discuss the primitive inference cycle of KRS and the Task-Processor which implements this cycle. In section four the inheritance implementation is described. Section five briefly describes KRS environments.

1. A DATA-DEPENDENCY MECHANISM FPPD.

FPPD is a data-dependency mechanism which plays a central role for the KRS implementation. It is developed to keep stored structures conform with the procedures they were computed from. Though the need for this follows directly from the KRS strategy to cache computed referents, the impact of FPPD goes much further.

1.1. Problem Description.

The focus of this section is on consistency maintenance. The theoretical functionality of consistency maintenance is to withdraw cached referents which are no longer conform with the definition they were computed from. Technically however, its involvement goes far beyond that. It also plays an important role when virtually infinite structures are shortcut, when inherited subjects are copied down, etc.

KRS's consistency maintenance system operates mostly behind the back of the user. Its functionality can be demonstrated by inspecting the KRS structures to see which values are cached at what time. This is illustrated in the following example. (1) shows a description of a part of the concept-graph. After the request given in (2), a lot of values are cached, as shows figure 1.

```
(defconcept PERSON                                                    1
  (birthyear (a number))
  (age (a number
         (definition
           (a subtraction
             (subtract-from current-year)
             (subtract-what (>> birthyear)))))))


(defconcept SUBTRACTION
  (a form
    (subtract-from (a number))
    (subtract-what (a number))
    (definition [form '(- (>> referent of ,(>> subtract-from))
                          (>> referent of ,(>> subtract-what)))])))


(defconcept JOHN                          (defconcept CURRENT-YEAR
  (a person                                 (a counter
    (birthyear [number 1961])))               (initial-value [number 1987])))


-->  (>> referent age of john)                                        2
<--  26
```

A possible mutation is shown in (3). By incrementing the concept Current-Year, the referent of John's age becomes invalid. All other cached referents remain (figure 2).

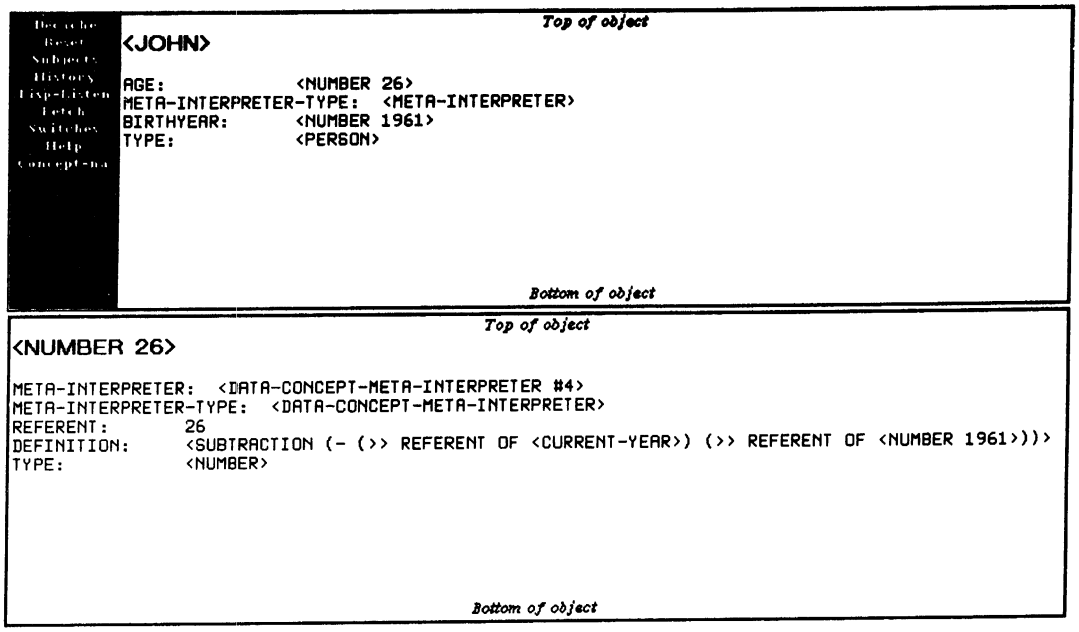


Fig 1: The concept John and its age when everything is cached.

```
--> (>> increment of current-year)                                    3
<-- <done>
```

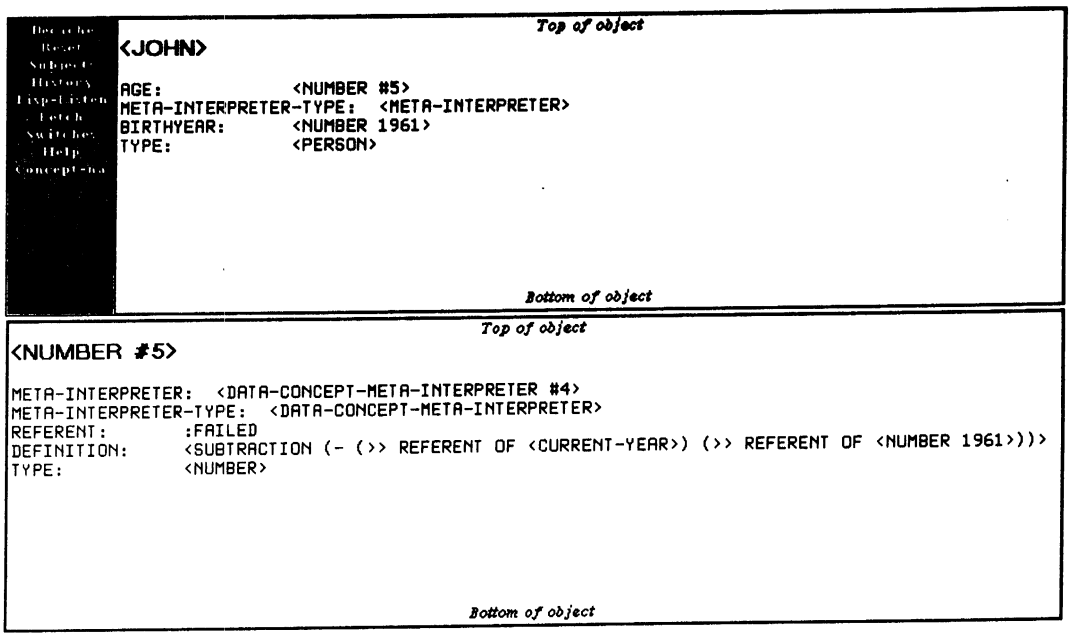


Fig 2: The concept John and its age when Current-Year is incremented.

If however instead of incrementing Current-Year this concept is redefined (4), not only the referent of John's age is withdrawn but also the referent of the definition of John's age (figure 3). Indeed, figure 1 showed that the previous version of Current-

Year was pulled into this LISP-form.

```
(defconcept CURRENT-YEAR                                    4
  [number 1988])
```

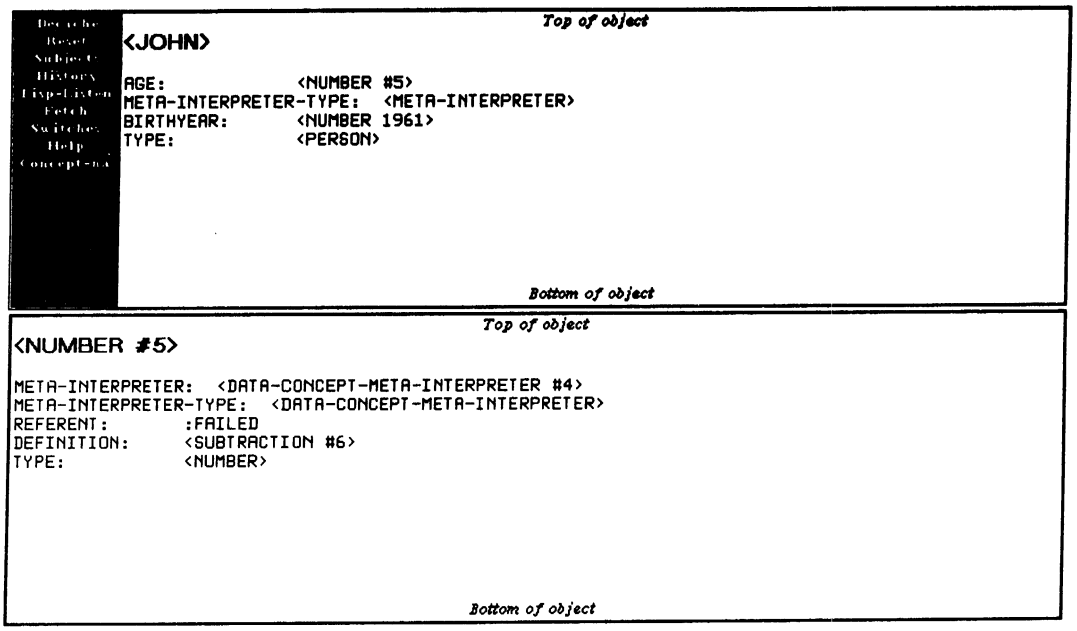


Fig 3: The concept John and its age when Current-Year is redefined.

Obviously, there are many possible mutations which may cause particular referents to be withdrawn. Figure 4 shows the effect of redefining the concept Person on the above example (5).

```
(defconcept PERSON                                          5
  (birthyear (a number))
  (age [number 30]))
```

As sketched in this example, the KRS consistency maintenance mechanism notices mutations and selectively withdraws cached referents, i.e. those that are affected by that change. This requires two functions to be performed: (i) during the computation, a data-dependency network must be constructed and (ii) when something changes, this network must be traced, to withdraw the affected cachings.

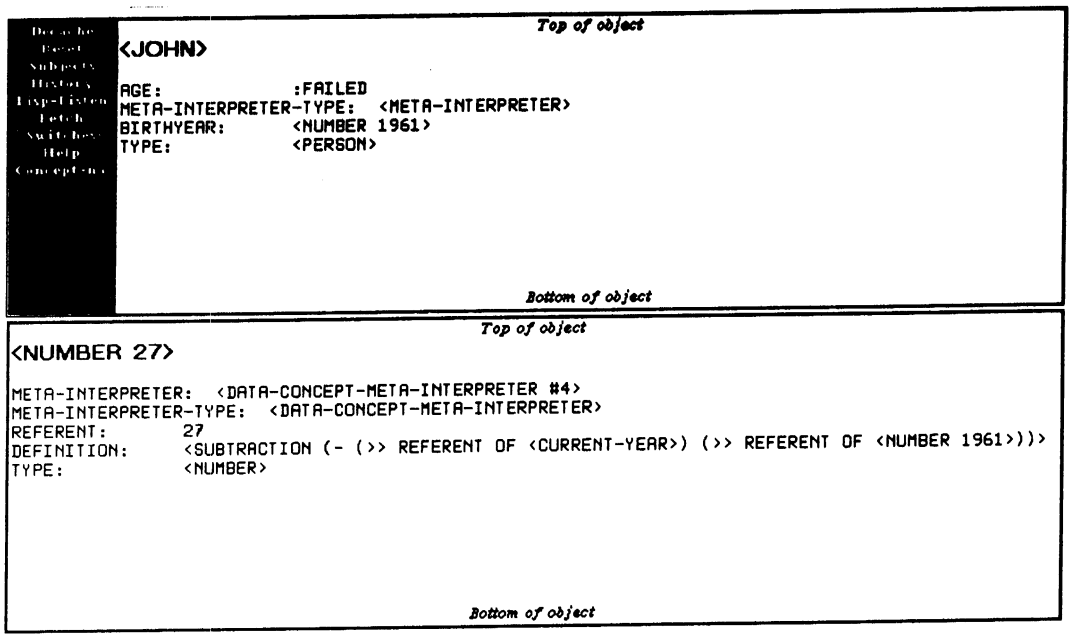


Fig 4: The concept John and its age when Person is redefined.

1.2. Sketch of the Solution.

1.2.1. Data-Dependencies.

[Stallman77] introduced the manipulation of data-dependencies in the problem-solving language ARS. They provide an alternative but more flexible solution for context dependent reasoning, known under the name *truth maintenance*. The mechanism is more flexible because it allows nonmonotonic inferences, i.e. falsifying previously assumed true facts or validating previously assumed false facts.

> *A data-dependency is an explicitly represented dependency between facts independently stored in a special purpose memory, caused by an implicit relationship.*

In ARS for example, there is an implicit relationship between the facts believed at a certain moment and the facts the problem solver is able to derive under those circumstances. The problem-solver uses this relationship to detect justifications, i.e. reasons for its beliefs. In KRS there is an implicit relationship between the state of the concept-graph just before and just after the computation of a referent. Out of this, support/dependent relations are constructed which are used to determine the validity of a computed value.

> *Data-dependency mechanisms are algorithms operating over a data-dependency graph.*

The data-dependency mechanism in ARS is called truth maintenance. It is explained in the next sub-section. The data-dependency mechanism operating in KRS is the consistency maintenance mechanism we have demonstrated in the previous sub-section.

1.2.2. Truth Maintenance Systems

Truth maintenance systems (TMS's) were introduced by Stallman and Sussman as a data-base management technique alternative to contexts [Stallman77].[7] These mechanism are extracted from ARS and generalized by Doyle in [Doyle79]. Doyle's TMS was the first autonomous data-management tool, usable by any separate problem solver.

The TMS serves three roles. First, it memorizes all deductions ever made, to avoid duplicating deductions or falling several times into the same contradiction. Second, by using the TMS, the problem solver can make nonmonotonic inferences. Therefore,

[7] The term Truth Maintenance is misleading. The mechanism does not determine which facts are true under the given set of assumptions, but which facts can reasonably be believed. Reason maintenance system is a more accurate term given in [McDermott83]. Indeed, an important role of these mechanisms is to keep track of reasons for its beliefs.

the TMS runs a sort of constraint satisfaction procedure (called Truth Maintenance) to find out which facts can be believed. Third, the TMS is responsible for the discovery and removal of contradictions in the fact data-base. Stallman and Sussman baptized this mechanism *dependency directed backtracking*. Dependency directed backtracking improves the behavior of traditional chronological backtracking by withdrawing an assumption for which the justifications indicate that it is actually co-responsible for the conflict at hand.

The data-dependencies in Doyle's TMS are called justifications.[8] Facts are stored in a data-base together with a set of justifications. A justification provides a reason for believing the fact. The justification together with the fact is called a node. The justification is said to support the fact. Every node has a status, which is either IN or OUT. IN and OUT does not correspond to True and False. An IN-node represents a fact which is currently believed by the problem-solver. An OUT-node represents a fact which is currently not believed. A justification contains two sets of nodes: its *in-justifiers* and its *out-justifiers*.

> *A node is IN if it has a justification for which all its in-justifiers are IN, and all its out-justifiers are OUT. Otherwise, it is OUT.*

A fact supported by a justification with no in-justifiers and no out-justifiers is believed under all circumstances. Such a fact is called a *premise*. A premise-node is always IN. A node without justifications is always OUT.

Out-justifiers make the system nonmonotonic. They allow believing a fact based on the lack of belief in another fact.

One of the roles of the TMS is to find a consistent and well founded status labeling. A status labeling is an assignment of labels IN or OUT to all nodes. Such a labeling is consistent if all the labels correspond with the rules described above. It is well-founded if for any IN-node, a noncircular chain of IN-nodes can be found back to out-nodes and premises. Well-foundedness is required to avoid situations where fact A is believed because fact B is believed and fact B is believed because fact A is.

EXAMPLE

Figure 5 shows a dribble of the TMS in use. The example starts by creating five TMS-nodes, representing the facts:

- John is Human.
- John is not Human.
- Every Human is Mortal.

[8] In fact there are several kinds of justifications in Doyle's program. The ones we describe here are called SL-justifications. They are the simpliest and the most intuitive ones.

```
(SETQ *ALL-NODES* NIL)                                  Clear the data-base.
nil

(NODE 'N1 '(IS-HUMAN JOHN))                             Add the facts:
n1                                                        "John is Human",
(NODE 'N2 '(IS-NOT-HUMAN JOHN))
n2                                                        "John is not Human",
(NODE 'N3 '(IS-HUMAN -> IS-MORTAL))
n3                                                        "Every Human is Mortal",
(NODE 'N4 '(IS-MORTAL JOHN))
n4                                                        "John is Mortal",
(NODE 'N5 '(IS-NOT-MORTAL JOHN))
n5                                                        "John is not Mortal".

(DESCRIBE-NODES)                                        So far, none of the facts is
((n5 out) (n4 out) (n3 out) (n2 out) (n1 out))          believed, because no
                                                        justifications are given yet.
(ADD-JUSTIFICATION '(IS-HUMAN JOHN) '(()(N2)))          Assume that John is Human.
n1

(DESCRIBE-NODES)                                        "John is Human" is believed now.
((n5 out) (n4 out) (n3 out) (n2 out) (n1 in))

(ADD-JUSTIFICATION '(IS-HUMAN -> IS-MORTAL) '(()()))    The rule "Every Human is Mortal"
n3                                                      is a premise.

(ADD-JUSTIFICATION '(IS-MORTAL JOHN) '((N1 N3)()))      "John is Mortal" can be believed
n4                                                      if believed that John is Human
                                                        and that every Human is Mortal.
(DESCRIBE-NODES)                                        Now, there is also believed that
((n5 out) (n4 in) (n3 in) (n2 out) (n1 in))             Every Human is Mortal, and that
                                                        John is Mortal.
(NODE 'N6 '(CAN-DIE JOHN))                              Add the fact: "John can die".
n6

(ADD-JUSTIFICATION '(CAN-DIE JOHN) '((N4) ()))          "John can die" can be believed
n6                                                      if believed that John is Mortal,
(ADD-JUSTIFICATION '(IS-MORTAL JOHN) '((N6)()))         and vice versa.
n4

(DESCRIBE-NODES)                                        Now also John can die is
((n6 in) (n5 out) (n4 in) (n3 in) (n2 out) (n1 in))     believed.

(ADD-JUSTIFICATION '(IS-NOT-HUMAN JOHN) '(() ()))       State that John is not Human by
n2                                                      making it a premise.

(DESCRIBE-NODES)                                        Now we belief no longer that
((n6 out) (n5 out) (n4 out) (n3 in) (n2 in) (n1 out))    1: John is Human,
                                                         2: John is Mortal,
                                                         3: John can die.
                                                        Notice that the circular
                                                        justifications for John is Mortal
                                                        and John can die are not used to
                                                        justify one another.
```

Fig 5: A Trace of the TMS in use.

- John is Mortal.
- John is not Mortal.

At that time, none of these facts is believed (all nodes are OUT), because no justifications for any node are given yet.

Then a justification for the fact "John is Human" is given. It says that this fact can be believed if we do not belief that John is not Human. In this way we make an assumption. The next justification makes the rule "Every Human is Mortal" a premise, e.g. always believed. The third justification says that "John is Mortal" may be believed if also "John is Human" and the previous rule are. By now, the facts "John is Human", "Every Human is Mortal" and "John is Mortal" are all believed.

Let's now skip part of the example and go to the last justification it gives. This justification makes the fact "John is not Human" a premise. This way the assumption that John is Human is withdrawn. Indeed, because the node n2 is the outjustifier which makes n1 IN, making n2 IN makes n1 OUT. Also n4 ("John is Mortal"), which was believed based on the belief in n1, becomes OUT.

The part of the example we have skipped illustrates the necessity of looking for well-founded support. We create a node n6 representing the fact "John can die", and two justifications saying that any of the facts "John can die" and "John is Mortal" can be believed if the other is. Obviously, also "John can die" is believed at this moment.

Interesting is the effect of the last justification on the nodes n4 and n6. This justification makes n2 IN and hence n1 OUT. Since n1 is now OUT, the justification which previously supported n4 is now violated. However, n4 has a second justification, saying that the node is IN if n6 is IN, which is the case. If this justification would make n4 IN, we would have the situation that n4 is IN because n6 is and vice versa. To avoid this sort of situations, the TMS searches for well-founded support for all the nodes it makes IN.

IMPLEMENTATION

The implementation of the basic truth maintenance mechanism differs in Doyle's approach from the approach of Stallman and Sussman. The latter algorithm is a kind of garbage collection algorithm, the former is constraint-relaxation.

When an assumption is taken back, Stallman and Sussman's implementation determines which facts are still to be believed by scanning forward from all assumptions which are currently believed and marking all nodes it can reach in this way. All marked nodes become in and all unmarked nodes out.

Doyle's truth maintenance procedure starts when a justification is added. If the node is out and the justification valid, truth maintenance must recalculate well-founded

support for the node and all its repercussions. This happens in three steps. In the first step the node and all its repercussions are marked for recalculation. All unmarked nodes still have well-founded support. In the second step the procedure tries to unmark as many marked nodes as possible by finding well-founded support for them solely based on other unmarked nodes. If there remain marked nodes after step two, they must have circular dependencies. Step three is a constraint relaxation procedure to search for well-founded support for those nodes. In outline this procedure works as in step 2, but it assumes that marked nodes are OUT. When, during this process, a node is found to be IN, all its consequences, which are nodes whose current status was found by assuming that the status of this node is OUT, will be reconsidered. A thorough description of this implementation is given in [Charniak80].

PROBLEMS

Doyle's TMS shows two fundamental deficiencies: the *single-state problem* and *unouting*.

The *single-state problem* refers to the fact that the problem solver can only explore one solution at a time. The problem solver can explore multiple solutions by making and withdrawing assumptions. Yet, the machineries to actually do so are so cumbersome and expensive, that it is unrealistic to say that the problem solver can actually reason with multiple hypothetical situations at the same time.

Unouting is the process of changing the status of a node to IN, when this status has been IN and has been made OUT before. When a node is unouted, all deductions previously made based on this node have to be unouted accordingly. However, the problem solver nor the TMS have any knowledge of the state which existed when the node was made OUT. To rebuild the state such that the node and all its consequences can be made IN consistently, a lot of search is required during which the states of many nodes may flip from IN to OUT and back many times. Hence, it is difficult for the problem solver to restart its computation at a point where it had stopped before.

A currently rather popular TMS is de Kleer's Assumption based TMS or ATMS [de Kleer86a, de Kleer86b]. It overcomes most traditional TMS problems. The ATMS explicitly distinguishes assumptions from assumed data. An *assumption* is a decision to assume something. The ATMS traces all justifications down to sets of assumptions. A set of assumptions is called a context. Each node has a label which is a set of contexts, i.e. contexts in which the datum represented by the node holds.

1.2.3. FPPD.

FPPD [Van Marcke86a] is a computational mechanism based on the manipulation of data-dependencies. It manipulates a large network, the nodes of which are called propositions, the connections are called direct-support-relations.

From an external point of view, a proposition returns a value when asked for. Internally however, the proposition may chose either to return an internally stored value or to compute the value using an internally stored procedure.

> *FPPD tries to minimize computation by storing as many computed results as possible, meanwhile regarding the correctness of the values it returns, i.e. the conformance between the values and the procedures they were computed from.*

Though inspired by the TMS mechanism, the environment in which FPPD runs poses fundamentally different requirements for its implementation. The techniques described in the previous sub-section can hence not be fully exploited for our purposes. Instead, FPPD is designed to be most effective under the following circumstances:

- *Massiveness.* Target applications for FPPD are large programs building up huge dependency networks. A typical KRS application for example can easily instantiate tens of thousands of FPPD-propositions (see chapter three).
- *Smoothness.* FPPD is developed to operate behind the scenes. There is always another program for which FPPD fulfills a role. FPPD should never cause any noticeable delay to that other program.
- *Complexity.* An FPPD network can be very complex. Data-dependencies are the result of arbitrary LISP computation and can thus form an arbitrary complex network. Therefore, FPPD dependencies must be detected automatically at run-time, unlike TMS justifications which are explicitly given.

Given the environment characteristics as described above, an FPPD proposition performs the following functions. (i) It stores a result when it is computed such that the next time this result is needed it can be used directly. (ii) If a computed result is no longer valid when it is requested, the proposition recomputes the value instead of returning the former result. A computed result is no longer valid if some other value which was used during the computation of this result has changed.

FORWARD PROPAGATION

There are two possible strategies to ensure the correctness of a stored value. Either one checks whether a value is still correct upon a request, or one withdraws the value when one of the values the stored value depends on changes. Both strategies require the presence of data-dependencies, i.e. dependencies between propositions for

which the value of the one was used for the computation of the value of the other. The first strategy implies using the dependencies in a backward way, the second strategy implies using them in a forward way.

FPPD opts for the second strategy, i.e. when a proposition's value is changed, all stored values of all propositions which have used the former value of the mutated proposition are withdrawn. This is more advantageous in a run-time environment in which there are far more requests then there are changes, which is in particular the case for KRS. When a proposition's value is withdrawn, we say that the proposition is *denied.*

LAZY CONSTRAINT SATISFACTION

Lazy constraint satisfaction indicates that values are not recomputed when they are denied, but only the next time they are requested. Lazy constraint satisfaction favors a gradual recomputation of invalidated values, in order to avoid massive recomputations after a small change. This strategy performs best when there are far more value requests than invalidations.

AUTOMATIC DEPENDENCY DISCOVERY

Data-dependencies, being the result of arbitrary LISP-computation, can form arbitrary complex networks. It can therefore become very difficult for a user program to trace dependencies down. To meet this difficulty, FPPD constructs the dependency network itself by looking over the shoulder as computation continues. Moreover, when some value has changed and a propagation of proposition denials is started, the corresponding part of the dependency network is destructed. The next time the values of the denied propositions are computed, a new, possibly different network will be built.

1.3. Functional Description.

FPPD manipulates a large network of propositions. The user[9] communicates with the FPPD network by creating propositions and by asking for values of propositions.

1.3.1. Types of Propositions.

In the minimal FPPD-model, there are two types of propositions. *Primitive-propositions* are propositions which have an initial-value. This value can be redefined by the user. *Relative-propositions* are propositions which have function and context. The value of a relative-proposition is computed by applying its function to its context. This happens the first time this value is requested. After its first computation, the same value is returned for every new request for as long as it remains valid. It becomes invalid if the value of one of the primitive-propositions on which this value ultimately depends changes.

The core of the FPPD dependency network consists of primitive and relative-propositions. However, a few other types of propositions were added later, because the user sometimes needs more control on the behavior of the dependency-network. For being able to describe the functionality of the different types, we have to define the terms *depends* and *changes* first.

> *We say that a value X depends on a value Y if X and Y are values of propositions and Y was requested during the computation of X.*
>
> *We say that a value X changes, if (i) X is the value of a primitive-proposition and X is redefined by the user, or (ii) if X depends on a value Y and Y changes.*

PRIMITIVE-PROPOSITION

A primitive-proposition has a stored value which can be set or reset by the user. Its only function is to return this value when requested.

RELATIVE-PROPOSITION

A relative-proposition has a function and a context. The value of a relative-proposition is the result of applying its function to its context. When this value is requested, it may be internally stored. In that case the value is not recomputed. If it is not stored, the value is computed and returned. Before returning however, this computed value is stored unless it is no longer in compliance with its function and context. This can be the case if the function itself changes a primitive-proposition on

[9] The word "user" is used in the sense of "the program using FPPD".

which the value it has computed depends.

SIMPLE-PROPOSITION

A simple-proposition has a function and a context. Every time its value is requested, it is computed by applying this function to this context. The value is thus never stored.

CONSTANT-PROPOSITION

A constant-proposition has a function and a context. The value of a constant-proposition is equal to the result of applying this function to this context the first time it is requested. After its first computation, this value is stored and never invalidated again.

STORING-PROPOSITION

A storing-proposition behaves exactly as a relative-proposition, except that it always stores its value after computation.

EAGER-PROPOSITION

An eager-proposition behaves exactly as a storing-proposition, except that once its value is computed, the proposition will automatically recompute its value each time any other proposition's value on which this proposition depends changes.

1.3.2. Dependency Nomenclature.

FPPD maintains a continuously changing graph of propositions based on the direct-dependent and direct-support relations.

> *We say that a proposition P1 is a direct-support of a proposition P2 if P2 is a relative-proposition and the value of P1 was asked during the computation of the current value of P2.*[10]

> *The inverse relation is direct-dependent. P1 is a direct-dependent of P2 if P2 is a direct-support of P1. Note that primitive-propositions can have direct-dependents but not direct-supports.*

The transitive-closure of the direct-support relation is the *Support* relation. A proposition P1 is a support of a proposition P2 if P1 is a direct-support of P2 or if P1 is a direct-support of a proposition P3 which is a support of P2.

[10] The term "was asked" is to be interpreted as follows: If the function of a proposition P1 (or another function called by this function) asks the value of a relative-proposition P2 whose function asks the value of a proposition P3, then, P3 is a direct-support of P2 and P2 a direct-support of P1, but P3 is not a direct-support of P1.

The inverse relation is *dependent*. It is also the transitive closure of the direct-dependent relation.

The *primitive-supports* of a proposition are all supports which are primitive-propositions.

Figure 6 shows those distinguished relations for the example given in (6).

```
(setq A (make-primitive t))                                                    6
(setq B (make-primitive 0))
(setq C (make-primitive 1))

(setq U (make-relative #'(lambda (x) (* 2 (get-proposition-value x)))
                       (list B)))

(setq V (make-relative #'(lambda (x) (* 2 (get-proposition-value x)))
                       (list C)))

(setq W (make-relative #'(lambda (x y z)
                           (if (get-proposition-value x)
                               (get-proposition-value y)
                               (get-proposition-value z)))
                       (list A U V)))
```

	type	*value*	*direct-dependents*	*direct-supports*	*dependents*	*supports*
A	primitive	T	W		W	
B	primitive	0	U		U, W	
C	primitive	1				
U	relative	0	W	B	W	B
V	relative	?				
W	relative	0		A, U		A, B, U

Fig 6: Data-Dependencies in FPPD; after computing the value of W.

1.3.3. The FPPD language.

The LISP functions described in this subsection constitute the interface to the FPPD mechanisms.

The following are functions to create propositions:

make-primitive *&optional (value nil)* *[function]*

Make-Primitive creates an returns a new primitive-proposition with a given initial value.

make-relative *function &optional (context nil)* *[function]*

Make-Relative creates and returns a new relative-proposition with a given function and context.

make-simple *function &optional (context nil)* *[function]*

Make-Simple creates and returns a new simple-proposition with a given function and context.

make-constant *function &optional (context nil)* *[function]*

Make-Constant creates and returns a new constant-proposition with a given function and context.

make-storing *function &optional (context nil)* *[function]*

Make-Storing creates and returns a new storing-proposition with a given function and context.

make-eager *function &optional (context nil)* *[function]*

Make-Eager creates and returns a new eager-proposition with a given function and context.

The following are functions to ask for the value of a proposition:

get-proposition-value *proposition* *[function]*

Get-Proposition-Value returns the value of *proposition*. If this function is called during the computation of another proposition's value, say the proposition X, then a direct-dependent link is instantiated from *proposition* to X.

probe-proposition-value *proposition* *[function]*

Probe-Proposition-Value returns the value of *proposition*. No new direct-dependent link is instantiated.

The functions Primitive-Redefine-Value and Deny-Proposition invoke the forward propagation of proposition denials:

primitive-redefine-value *primitive-proposition new-value* *[function]*

Primitive-Redefine-Value changes the value of *primitive-proposition* into *new-value*. It causes all dependents to be denied.

deny-proposition *proposition* *[function]*

Deny-Proposition denies *proposition*. All dependents of *proposition* are denied accordingly.

Proposition-Denied-P and Proposition-P are two auxiliary functions which are useful in making large programs based on FPPD:

proposition-denied-p *proposition* *[function]*

Proposition-Denied-P Returns T if the proposition is denied, NIL otherwise. If this function is called during the computation of another proposition's value, a direct-dependent link is instantiated. This way other propositions which depend on this proposition being denied will themselves be denied when this one's value gets computed.

proposition-p *thing* *[function]*

Proposition-P checks whether a given object is a proposition.

1.4. Implementation.

1.4.1. The Data-Structures.

The data-structure for a proposition is a structure (7). It has fields type, value, to-compute, context, direct-dependents and version.

```
(defstruct (PROPOSITION                                                    7
             ...)
  (type nil)
  (value :denied)
  (to-compute nil)
  (context nil)
  (direct-dependents nil)
  (version nil))
```

The functions to create propositions of different types are straightforward. (8) shows those to create primitive-propositions and relative-propositions.

```
(defun MAKE-PRIMITIVE (value)                                              8
  (make-proposition :type :primitive :value value))

(defun MAKE-RELATIVE (to-compute &optional (context nil))
  (make-proposition
    :type :relative
    :to-compute to-compute
    :context context))
```

1.4.2. Propagation of Proposition Denials.

The forward propagation of proposition denials must perform several functions. First, all dependents of the proposition the propagation starts with must be denied. Second, the corresponding part of the dependency-graph must be destructed, to avoid that the old dependency-graph wrongly interferes with the new graph which will be constructed when values get recomputed. Third, all touched eager-dependents must recompute their values. This must happen when the propagation is entirely finished, since during the propagation the network is in an inconsistent state.

The propagation of proposition denials is triggered either by the function Primitive-Redefine-Value (9) or by the function Deny-Proposition (10)). The propagation is executed with the function Deny-All-Dependents. This function returns all eager-dependents it denied such that they can be recomputed.

```
(defun PRIMITIVE-REDEFINE-VALUE (proposition new-value)                    9
  (if (primitive-p proposition)
      (progn
        (setf (proposition-value proposition) new-value)
        (mapc #'get-proposition-value (deny-all-dependents proposition))
        proposition)
      (fppd-error "Attempt to change the value of a non-primitive-proposition")))
```

```
(defun DENY-PROPOSITION (proposition)                                    10
  (unless (int-proposition-denied-p proposition)
    (setf (proposition-value proposition) :denied)
    (mapc #'get-proposition-value (deny-all-dependents proposition))))
```

The function Deny-All-Dependents (11) is written iteratively for efficiency reasons. Although it has become far more complex than its recursive version this way, we find this justified by the crucial role it plays within the FPPD implementation and the very large networks it is going to operate on.

The function consists of one large iteration, using the following variables:

- All-The-Dependents is an accumulator of lists of direct-dependents to be visited.
- The-Current-Dependents is one such list, the one whose elements are currently being visisted.
- Eager-Propositions is an accumulator of eager-propositions encountered. This list has to be returned at the end.

```
(defun DENY-ALL-DEPENDENTS (proposition)                                 11
  (let ((direct-dependents (proposition-direct-dependents proposition)))
    (setf (proposition-direct-dependents proposition) nil)
    (do ((all-the-dependents nil)
         (the-current-dependents direct-dependents))
         (eager-propositions
           (when (eager-p proposition)
             (list proposition))))
        ((null (or all-the-dependents the-current-dependents))
         eager-propositions)

      (let* ((current-proposition
               (proposition-in-version (pop the-current-dependents)))
             (current-proposition-value
               (and current-proposition
                    (proposition-value current-proposition))))

        (unless (eq current-proposition-value :denied)
          (push (proposition-direct-dependents current-proposition)
                all-the-dependents)
          (setf (proposition-value current-proposition) :denied)
          (setf (proposition-direct-dependents current-proposition) nil)
          (when (and (eager-p current-proposition)
                     (not (eq current-proposition-value :computing)))
            (push current-proposition eager-propositions)))

        (when (null the-current-dependents)
          (setq the-current-dependents (pop all-the-dependents)))))))
```

1.4.3. Versions.

The implementation of the forward propagation described in the previous sub-section explicitly stores direct-dependent links. Since the propagation requires accessing all dependents of a proposition, the direct-dependents in fact simulate the dependent links. Obviously it is unfeasible to make those dependent links explicit. This would cause an explosion of the number of stored pointers.

For a long time, FPPD has also been storing direct-support relations, the reason for which is the following. The propagation described in the previous sub-section only destructs the part of the dependency network it traverses. There usualy are however also propositions not accessed by the traversal who have direct-dependents to one of the denied propositions. These are not removed and can possibly cause unnecessary denials in the future (figure 7). Therefore, the direct-support links have been used at each visited proposition to go one step back to each of its direct-supports, and remove the obsolete direct-dependent link for this proposition.

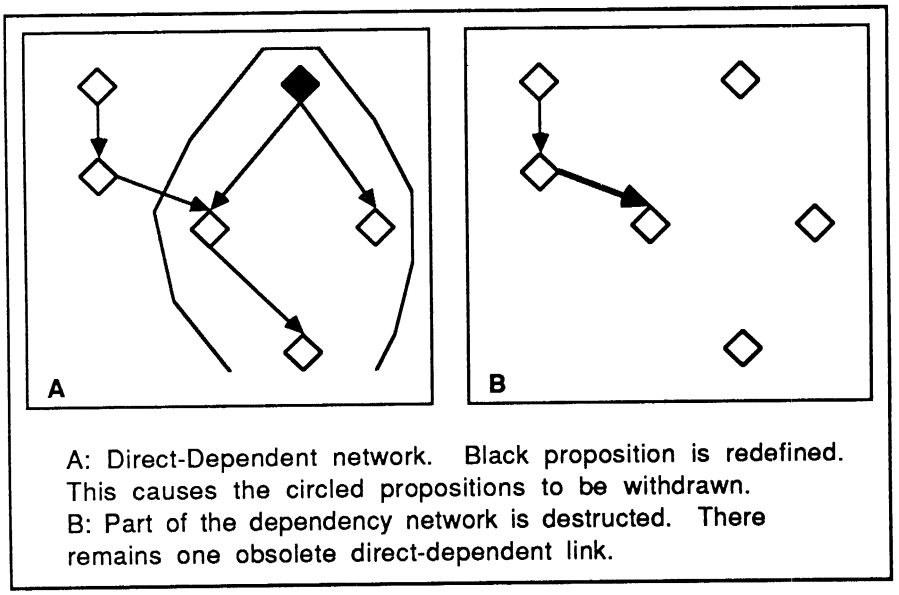


Fig 7: Obsolete direct-dependent links.

The current FPPD implementation has abandoned this step, since it was obviously a very expensive operation. To prevent the erroneous effect of all obsolete dependency links, we introduced a version system. Each proposition receives a version, which is a pointer back to the proposition. In addition, all direct-dependent links point to the version instead of directly to the proposition. Each time a proposition's value is computed, it receives a new version, and the pointer in the old version is removed. Hence, all obsolete direct-dependent links point to the old version through which they can no longer reach the proposition (figure 8).

New-Version (12) removes the reference to the proposition from the old version, and installs a new version in the proposition. Proposition-In-Version (13) returns, given a version, the proposition the version refers to.

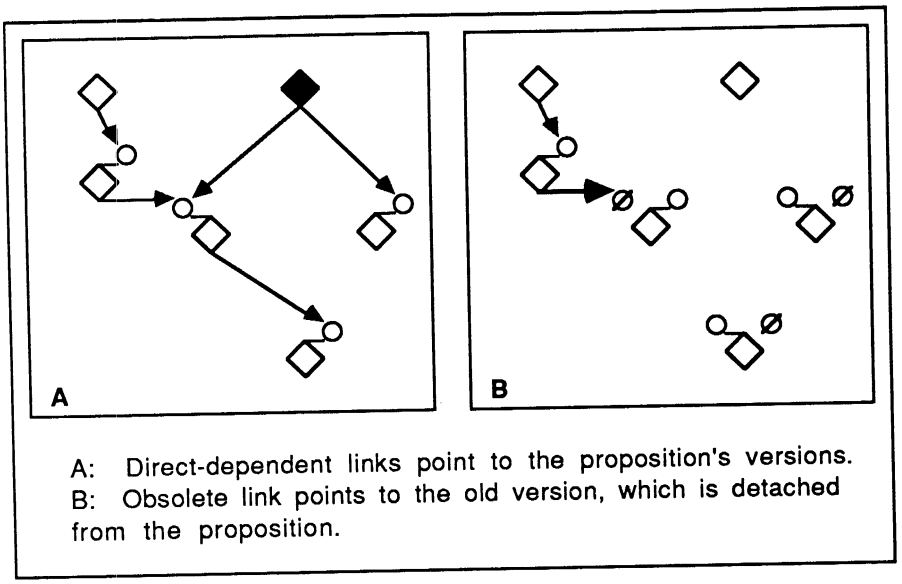


Fig 8: Direct-dependent links through versions.

```
(defun NEW-VERSION (proposition)                                          12
  (rplacd (proposition-version proposition) nil)
  (setf (proposition-version proposition) (cons proposition proposition)))


(defmacro PROPOSITION-IN-VERSION (version)                                13
  '(cdr ,version))
```

1.4.4. Getting and Probing a Proposition's Value.

The function Get-Proposition-Value (14) distinguishes the following possibilities. If the the current value of the proposition is ':computing', then the computation is circular. Otherwise, if the proposition is not denied, its value can be returned, after calling the function I-Am-Used. This function installs a data-dependency. If the proposition is denied, the value must be computed and stored before it can be returned.

```
(defun GET-PROPOSITION-VALUE (proposition)                                    14
  (let ((p-value (proposition-value proposition)))
    (cond
      ((eq p-value :computing)
       (fppd-error "You are trapped in a circular computation !"
                   :circular-computation
                   proposition))
      ((not (eq p-value :denied))
       (i-am-used proposition)
       p-value)
      (t
       (let ((eager-dependents (deny-all-dependents proposition)))
         (i-am-used proposition)
         (new-version proposition)
         (let ((value (compute-and-store-value proposition)))
           (mapc #'get-proposition-value eager-dependents)
           value))))))
```

The function Probe-Proposition-Value is identical, except that I-Am-Used is not called (15).

```
(defun PROBE-PROPOSITION-VALUE (proposition)                                  15
  (let ((p-value (proposition-value proposition)))
    (cond
      ((eq p-value :computing)
       (fppd-error "You are trapped in a circular computation !"
                   :circular-computation
                   proposition))
      ((not (eq p-value :denied))
       p-value)
      (t
       (let ((eager-dependents (deny-all-dependents proposition)))
         (new-version proposition)
         (let ((value (compute-and-store-value proposition)))
           (mapc #'get-proposition-value eager-dependents)
           value))))))
```

The function I-Am-Used installs a dependency between its argument and the current binding of the variable *active-proposition-version*.

```
(defun I-AM-USED (proposition)
  (when *active-proposition-version*
    (add-dependent proposition *active-proposition-version*)))


(defun ADD-DEPENDENT (proposition direct-dependent)
  (push direct-dependent (proposition-direct-dependents proposition)))
```

1.4.5. Computing a Value.

To compute a proposition's value, its function is applied to its context. The unwind-protect assures that the necessary cleaning-up is done when the computation is aborted. The variable *active-proposition-version* is dynamically bound such that the installation of new direct-dependent links will point to the version of this proposition.

```
(defun COMPUTE-AND-STORE-VALUE (proposition &aux *active-proposition-version*)
  (declare (special *active-proposition-version*))
  (setq *active-proposition-version* (proposition-version proposition))
  (setf (proposition-value proposition) :computing)
  (unwind-protect
      (store-value-if-needed
        proposition
        (apply (proposition-to-compute proposition)
               (proposition-context proposition)))
    (when (eq (proposition-value proposition) :computing)
      (setf (proposition-value proposition) :denied))))


(defun STORE-VALUE-IF-NEEDED (proposition value)
  (case (proposition-type proposition)
    ((:relative)
     (when (eq (proposition-value proposition) :computing)
       (setf (proposition-value proposition) value)))
    ((:storing :eager)
     (setf (proposition-value proposition) value))
    ((:simple))
    ((:constant)
     (new-version proposition)
     (setf (proposition-value proposition) value)))
  value)
```

1.5. Example.

In the following sub-sections, there will be many examples of the use of FPPD in the KRS implementation. In this sub-section we describe how FPPD is applied to compute a concept's referent.

To compute the referent of a concept, a relative-proposition is attached to this concept. The function of this proposition is the function Restart-Processor-To-Eval-Definition and the context only contains this concept (16).

```
(make-relative #'restart-processor-to-eval-definition                    16
               (list concept))
```

The function Restart-Processor-To-Eval-Definition (17) takes a concept as argument and retrieves and evaluates the referent of this concept's definition. Restart-Processor-To-Eval-Definition uses the task-processor, which is discussed in section three.

```
(defun RESTART-PROCESSOR-TO-EVAL-DEFINITION (concept)                    17
  (start-processor
    (create-task '(definition referent eval) concept nil)))
```

Another large example of FPPD programming is given in [Van Marcke86b]. It shows for example how a careful use of both the functions Get-Proposition-Value and Probe-Proposition-Value allows the user to tune the construction of the dependency-graph and hence to acquire a better performance.

2. DATA STRUCTURES.

This section defines the data structures which constitute the concept-graph. They have to be designed to economically represent a virtually infinite graph.

2.1. Problem Description.

MEMORY EFFICIENCY

During the execution of the program Extrak [Strickx87] we measured the following figures. After loading the KRS system (rel. 2.0), 357 concepts were created. Loading the EXTRAK program, installed another 194 concepts. After a first diagnose session, in total 2703 concepts were created. After the second diagnose session, this were 3365 concepts or another 662 new concepts. In chapter three, detailed metering data is given.

INFINITE REGRESS

The concept-graph is designed in a reflective way. Each component in the graph, i.e. each node and each link, is explicitly represented by a **separate** and unique concept. In particular, each concept has its meta-interpreter and each subject its subject-concept. We call meta-interpreters and subject-concepts *reflective concepts* because they represent the concept-graph of which they are themselves part.

Reflective concepts obviously lead to an infinite growth of the concept-graph. The key reason for this is the word *separate* we used above. A concept's meta-interpreter is the meta-interpreter of **only** this concept. Since it is itself a concept this recurs infinitely. The same holds for the subject-concepts. (18) shows some browsing through these structures. The depth of such requests is unlimited.

```
--> (>> meta-interpeter meta-interpreter ... meta-interpreter of john)      18
<-- <meta-interpreter #102>

--> (>> handler type handler subject-of handler age of John)
<-- <type-subject of <subject-of-subject of <age-subject of <john>>>>
```

Other approaches for reflectivity in object-oriented programming languages are fundamentally different. In the ObjVlisp model for example [Cointe86], each object is an instance from a class and each class is again an object. This model does not prescribe however that each object has a separate class. In fact, each class may well be the class of many objects. This way the recurrence of classes can be smoothly ended. Indeed, the class of the object representing the class called "Class" is the class Class itself. The KRS approach is extensively studied in [Maes87].

REFLECTIVE DATA

Although the data about a concept theoretically belongs to its meta-interpreter, it can not be internally organized this way because of the infinite regress described above. Hence we must sort of mix information of two levels within the concept itself, i.e. information about the concept's referent and information about the concept itself.

Since we are going to describe meta-concepts, which have as referent another concept, this information will certainly overlap if we do not carefully keep different levels apart. This overlap can easily be demonstrated. (19) shows a description of the concept Meta-Interpreter. It has a subject "concept-name". As subjects normally describe characteristics of the concept's referent, this concept-name-subject describes not the concept-name of the concept Mete-Interpreter but of its referent. (Remember that referents of meta-interpreters are themselves concepts.) Hence, the concept-name of the concept Meta-Interpreter is known by the meta-interpreter of the concept Meta-Interpreter (20).

```
(defconcept META-INTERPRETER                                          19
  (a meta-concept
    (concept-name ...)
    ...
    )))


-->  (>> concept-name meta-interpreter of Meta-Interpreter)           20
<--  <symbol meta-interpreter>
```

The assumption that the subjects of a concept describe the characteristics of its referent is followed by KRS *as well as possible*. It has however appeared impossible to remain consequent till the end. Particular subjects can not fit in this model. The subjects in question are the subjects "referent" and "meta-interpreter". Both subjects clearly describe characteristics of the concept itself and should therefore be subjects of the concept's meta-interpreter. Requests for a concept's referent or meta-interpreter can however not be formulated as in (21). This does obviously not work since we use exactly those subjects to shift from one level to the other.

```
-->  (>> referent meta-interpreter of [number 3])                    21
<--  3

-->  (>> meta-interpreter meta-interpreter of [number 3])
<--  <meta-interpreter of <number 3>>[11]
```

Also the location of the subjects "type" and "definition" can be questioned. It seems that those can be justified at any of both levels. The type-subject of John for example can be seen both as a characteristic of the person and as a characteristic of the

[11] Here we slightly deviate from the normal printed representation of meta-interpreters to make clear which meta-interpreter we mean.

concept. In the first case we say that it is a characteristic of the person "john" that he is a Person. In the second case, we say that it is a technical characteristic of the concept John that it inherits from the concept Person, i.e. it is the concept John that is augmented with the subjects of the concept Person.

2.2. Sketch of the Solution.

2.2.1. Virtual Subjects.

A *virtual subject* is a subject for which there is no subject-concept automatically created. Instead of computing the subject's filler by computing the referent of the subject-concept, an alternative data structure is used by which the subject's filler can be computed directly. This is more efficient than explicitly creating the subject-concept, both in terms of speed and in terms of memory allocation.

The data-structure to hold a virtual subject is an FPPD-proposition. The context of this proposition is the KRS environment (see section 5) in which the subject's filler is to be computed. The function in this proposition evaluates what would normally be the referent of the definition of the subject-concept within the correct KRS environment. By using an FPPD-proposition, we get the same behavior as a ordinary subject-concept.

Virtual subjects are invisible for the user. When the particular subject-concept is explicitly requested, it is created and stored at that time. This newly created subject-concept shows exactly the same behavior as any other explicit subject-concept.

2.2.2. Concept-Memory and Concept-Meta-Info.

Concepts have two information dictionaries, one called *memory* and one called *meta-info*. The subjects of the concept are stored in memory. All information about the concept is stored in meta-info such that it can be interpreted by the corresponding subject of the concept's meta-interpreter.

The definition of the concept-name of a meta-interpreter for example gets the actual concept-name of its referent out of this one's meta-info (22).

```
(defconcept META-INTERPRETER                                              22
  (a meta-concept
    (concept-name
      (a symbol
        (definition
          [form (ks::meta-info-element (>> referent) 'concept-name)])))
    ...
    )))
```

The KRS interpreter checks whether the meta-interpreter-type of a concept overrides the default behavior of the concept Meta-Interpreter. If not, it uses a built-in procedure instead of actually using the meta-interpreter. For example if the pname-subject of a concept's meta-interpreter-type is inherited from Meta-Interpreter, the concept is printed with the function Concept-Print-Self. This is a function which simulates the pname subject of Meta-Interpreter without using it.

The information whether a meta-interpreter-type overrides a default-subject is computed and stored on the meta-info of the meta-interpreter-type and thus interpreted by this one's meta-interpreter (23). Also this computation is done by FPPD-propositions to ensures that it remains correct.

```
--> (>> special-pname-p meta-interpreter meta-interpreter-type of john)      23
<-- <false>

--> (>> special-pname-p meta-interpreter meta-interpreter-type age of john)
<-- <true>
```

2.3. Functional Description.

2.3.1. Concept-Memory.

A concept's memory is a list of attributes. Each attribute represents one of the concept's subjects. Besides information about the subject itself, the attribute also contains information about the way the subject is internally represented.

An attribute is a record (implemented as a dotted list and a set of functions operating on this list) containing the following fields:

ACCESS
a LISP-symbol equal to the name of the subject the attribute represents.

THING
the actual subject data. Can be a subject-concept or an FPPD-proposition.

KEY
a keyword denoting the format in which the subject data is stored.

AUX
an optional auxiliary field in which additional information can be stored. For example when a virtual-subject is made explicit, it is stored here.

The following are the different possible keywords and the kind of structure they denote:

:SUBJECT
an explicit-subject.

:VIRTUAL-SUBJECT
a virtual subject.

:INHERITED-SUBJECT
an inherited subject.

:CACHED-REFERENT
a referent computed in an FPPD-proposition.

:STORED-REFERENT
a referent stored by the KRS parser.

The following operations constitute the attribute data structure:

make-attribute *access* *[macro]*

creates a new attribute given an access.

fill-attribute *att thing key &optional aux* *[macro]*

inserts new data in an existing attribute *att*.

attribute-access *att* *[macro]*

returns the contents of the field access of the attribute *att*.

attribute-thing *att* *[macro]*

returns the contents of the field thing of the attribute *att*.

attribute-key *att* *[macro]*

returns the contents of the field key of the attribute *att*.

attribute-aux *att* *[macro]*

returns the contents of the field aux of the attribute *att*.

find-attribute *concept access* *[macro]*

returns the attribute named *access* in the memory of *concept*, or NIL if this does not exist.

get-attribute *concept access* *[function]*

returns the attribute named *access* in the memory of *concept*, creating it if it does not exist.

2.3.2. Meta-Info.

A concept's meta-info is an association-list on which information can be stored in any format.

meta-info-element *concept access* *[macro]*

returns data stored in the meta-info-field of *concept* with the key *access*.

new-meta-info-element *concept access data* *[function]*

stores *data* in the item with key *access* in the meta-info of *concept*, hereby possibly replacing previous data.

The following functions retrieve data from a concept's meta-info:

concept-name *concept* *[function]*

returns the name of *concept* or NIL for an anonymous concept.

concept-distance *concept* *[function]*

returns the distance of *concept*. A concept's distance is part of the KRS environment.

concept-krs-env *concept* *[function]*

returns the krs-env of *concept*. A concept's krs-env is part of the KRS environment.

concept-lisp-env *concept* *[function]*

returns the lisp-env of *concept*. A concept's lisp-env is part of the KRS environment.

concept-id *concept* *[function]*

returns the concept-id of *concept*.

2.4. Implementation.

2.4.1. Concepts.

Concepts are created with a named defstruct, with two fields: memory and meta-info.

```
(defstruct (CONCEPT
             :named
             :predicate
             (:print-function print-concept)
             (:constructor make-cpt))
  (memory nil)
  (meta-info nil))
```

The function Make-Concept (24) creates a new concept. The KRS environment and the concept-name are stored on the meta-info.

```
(defun MAKE-CONCEPT (distance krs-env lisp-env &optional concept-name)          24
  (incf *concept-counter)
  (let* ((new-concept (make-cpt))
         (meta-info
           (append
             (list (cons 'distance (cons new-concept distance)))
             (when krs-env (list (cons 'krs-env krs-env)))
             (when lisp-env (list (cons 'lisp-env lisp-env)))
             (when concept-name (list (cons 'name 'concept-name))))))
    (setf (concept-meta-info new-concept) meta-info)
    new-concept))
```

Attributes are represented as dotted lists. All operations on them are straightforward LISP-functions or macros. Some of them are shown below.

```
(defmacro MAKE-CONCEPT-ATTRIBUTE (access)
  `(list ,access))

(defmacro FILL-ATTRIBUTE (att thing key v-sub)
  `(rplacd ,att (cons ,thing (cons ,key ,v-sub))))

(defmacro ATTRIBUTE-KEY (att)
  `(caddr ,att))


(defmacro FIND-ATTRIBUTE (concept access)
  `(assoc ,access (concept-memory ,concept)))

(defun GET-ATTRIBUTE (concept access)
  (let ((memory (concept-memory concept)))
    (or (assoc access memory)
        (let ((new-att (make-concept-attribute access)))
          (setf (concept-memory concept) (cons new-att memory))
          new-att))))
```

Also the functions to retrieve information about a concept itself are straightforward:

```
(defun CONCEPT-NAME (concept)
  (meta-info-element concept 'concept-name))
```

Meta-Info-Element is a macro to search data in the meta-info-field of a concept:

```
(defmacro META-INFO-ELEMENT (concept access)
  `(cdr (assoc access (concept-meta-info ,concept))))
```

2.4.2. Subjects

The creation of a subject depends on the kind of description. For an explicit description a subject-concept is created, for an implicit subject-description a virtual-subject is created. In both cases the data-structure representing the new structure is stored in an FPPD primitive-proposition on the corresponding attribute.

```
(defun MAKE-SUBJECT-FOR-CONCEPT (concept access filler-description)
  (let* ((att (get-attribute concept access))
         (thing (attribute-thing att))
         (key (if (atom access) :virtual-subject :subject))
         (subject
          (case key
           (:virtual-subject
            (make-virtual-subject-for-concept concept access filler-description))
           (:subject
            (make-explicit-subject-for-concept concept
                                               (first access) (cdr access))))))
    (if (primitive-p thing)
      (progn
        (primitive-redefine-value thing subject)
        (fill-attribute att thing key nil))
      (progn
        (when (proposition-p thing)
          (deny-proposition thing))
        (fill-attribute att (make-primitive subject) key nil)))))
```

An explicit subject-concept is created by interpreting the corresponding concept-description.

```
(defun MAKE-EXPLICIT-SUBJECT-FOR-CONCEPT  (concept access concept-description)
  (let ((subject
         (krs-eval
          (funcall
           (select-transformer concept-description)
           concept-description
           `(>>-pinned ',concept '(type ,access handler)))
          (list (concept-distance concept))
          (concept-krs-env concept)
          (concept-lisp-env concept))))
    (choose-proposition-type subject)
    (new-meta-info-element subject 'access access)
    subject))
```

A virtual-subject is represented by an FPPD-proposition.

```
(defun MAKE-VIRTUAL-SUBJECT-FOR-CONCEPT (concept access filler-description)    25
  (make-relative
   `(lambda (distance krs-env lisp-env)
      ,(funcall (select-transformer filler-description)
                filler-description `(>>-pinned ',concept '(type ,access))))
   (list
    (concept-distance concept)
    (concept-krs-env concept)
    (concept-lisp-env concept))))
```

2.4.3. Virtual-Subjects.

The proposition to create a virtual-subject is created by (26). The symbols concept and subject-filler-description are bound to the owner of the subject and the description of the subject's filler respectively.

```
(fppd:make-relative                                                    26
  `(lambda (distance krs-env lisp-env)
     ,subject-filler-description)
  (list (concept-distance concept)
        (concept-krs-env concept)
        (concept-lisp-env concept)))
```

The function to make a virtual-subject explicit is shown in (27). The referent of the definition and the KRS environment for this virtual subject are again filtered out of the FPPD-proposition which represents the virtual-subject. The virtual-subject is stored as referent-proposition of the new subject-concept.

```
(defun MAKE-VIRTUAL-SUBJECT-EXPLICIT (concept access)                  27
  (let*
    ((att (find-attribute concept access))
     (proposition (attribute-thing att))
     (definition (third (fppd::proposition-to-compute proposition)))
     (prop-context (fppd::proposition-context proposition))
     (krs-env (second prop-context))
     (lisp-env (third prop-context))
     (subject
      (make-instance-from-description
       '((type subject))
       (list (first prop-context))
       krs-env lisp-env)))
   (make-virtual-subject-for-concept
     subject
     'definition
     `(data-concept-special-instance-creator
        'form
        '(,definition)
        ',(concept-distance subject) krs-env lisp-env))
   (add-referent-proposition subject proposition)
   (new-meta-info-element subject 'access access)
   (fill-attribute att proposition :virtual-subject subject)
   subject))


(defmacro ADD-REFERENT-PROPOSITION (concept proposition)
  `(fill-attribute
     (get-attribute ,concept)
     ,proposition
     :cached-referent
     nil))
```

2.5. Example.

Figure 9 shows the data-structure representing the concept John (28).

```
(defconcept JOHN                                                    28
  (a person
    (birthyear [number 1961])
    ((adult-p (a subject
                (definition [form (>> of true)])))))))
```

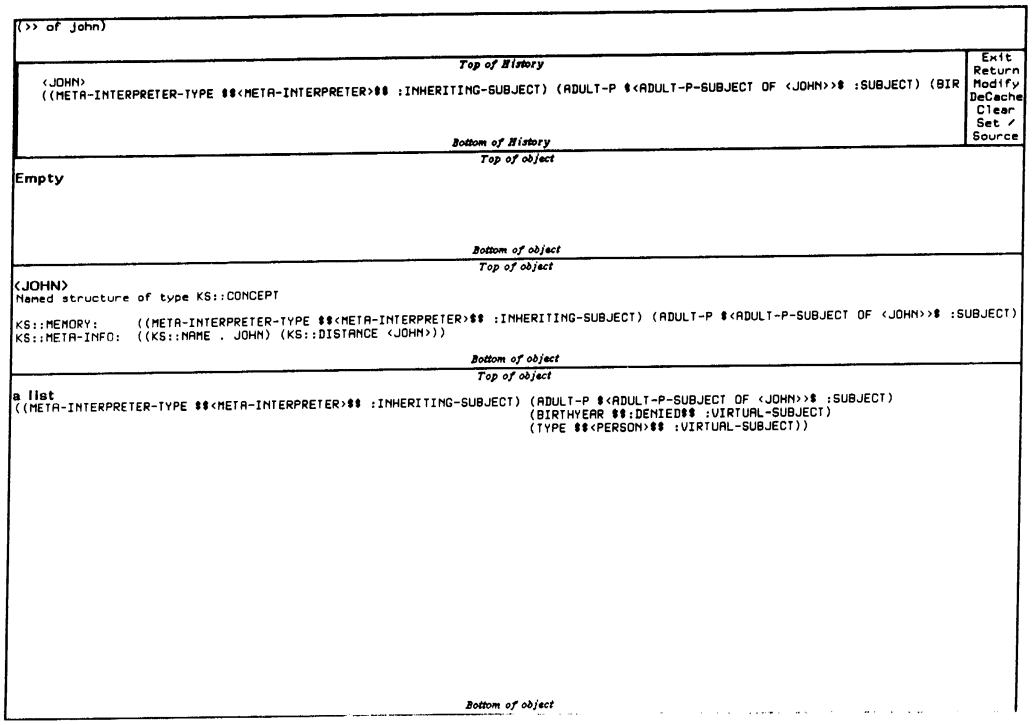


Fig 9: The concept John.

```
-->  (>> birthyear of john)                                         29
<--  <number 1961>

-->  (>> adult-p of john)
<--  <true>
```

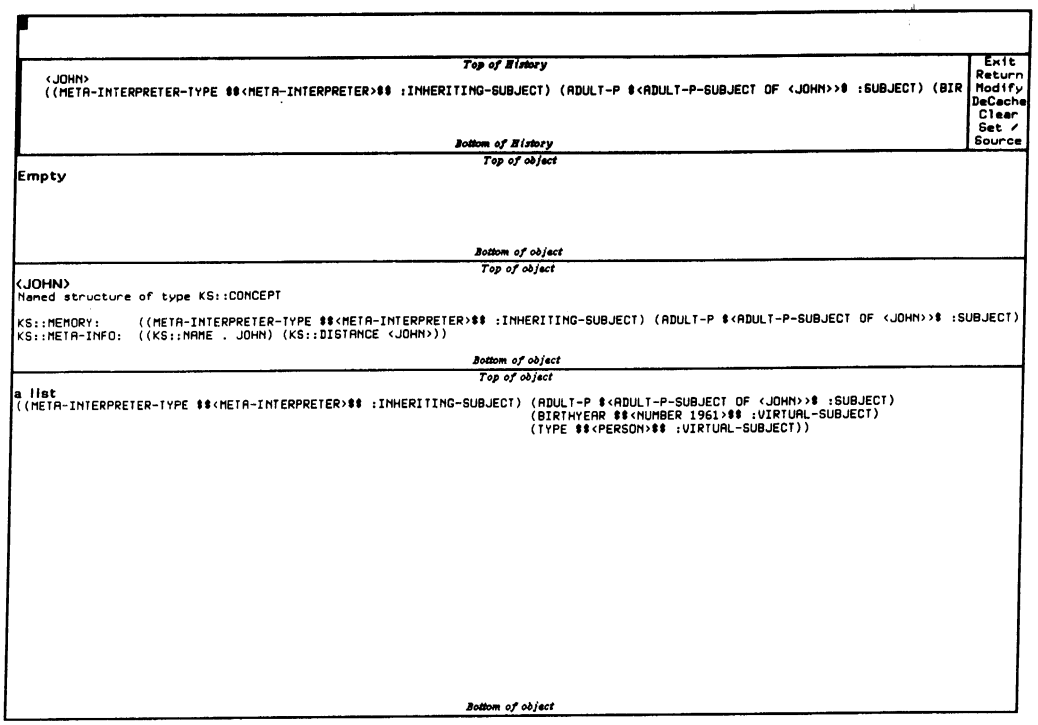


Fig 10: The concept John when subject-fillers are computed.

```
--> (>> handler birthyear-subject of john)                    30
<-- <birthyear-subject of John>
```

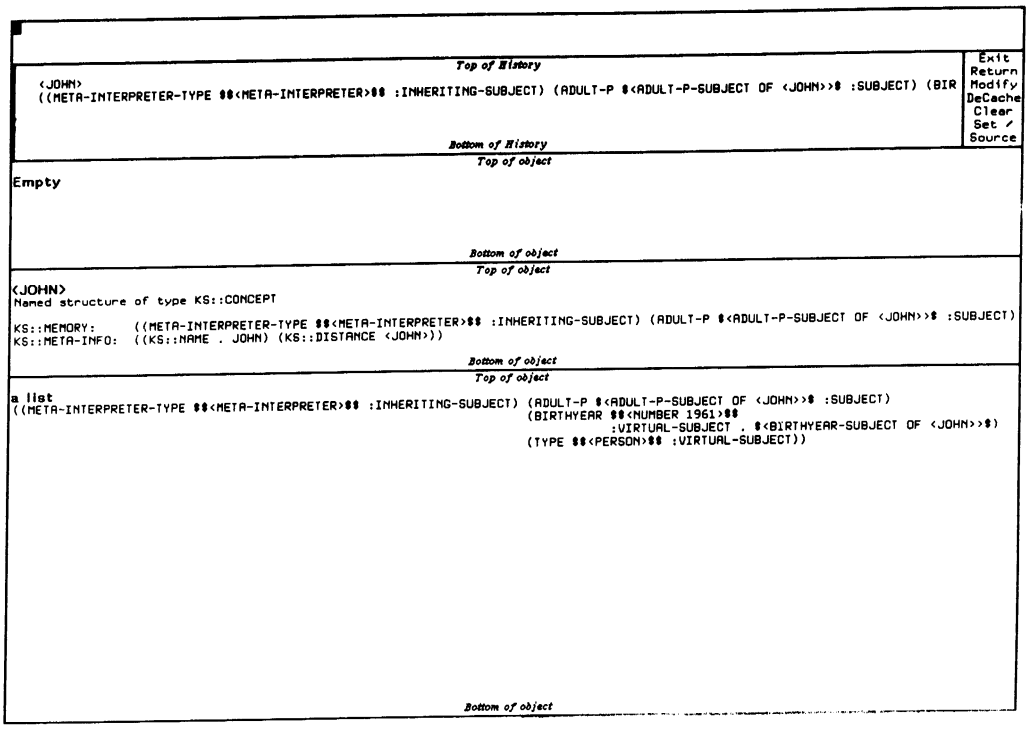


Fig 11: The concept John when birthyear-subject is made explicit.

3. THE INTERPRETATION CYCLE.

This section describes the primitive interpretation cycle of KRS. This cycle, typically started by a request, has in outline the following role:

> *The functions of the interpretation cycle are to decide which step to take next, to give the command for executing this task, and to decide how to continue.*

The interpretation is implemented using an agenda-based technique, which we call the task-processor.

3.1. Problem Description.

For the implementation of the interpretation cycle, there are two major considerations. One, it must run with optimal speed. Two, it must provide maximal flexibility. Both requirements are mainly due to the fact that this interpreter is at the heart of the KRS interpretation mechanism.

EFFICIENCY

Metering of the execution of a large KRS program on a Symbolics 3600 has shown that of the total time the processor is executing KRS functions, fifty percent is being taken by the interpretation cycle.[12] These observations immediately rule out the possibility of a recursive implementation, although this would have been the most natural solution given the nature of a KRS request (31).

```
(>> referent age friend of john)                                    31

==>  (>> referent age of (>> friend of john))
```

A recursive implementation is far too expensive for several reasons. First of all, recursion can go very deep. It is not limited by the length of the requests the user gives, since short requests can expand into very long sequences, as illustrates (32).

```
(>> referent age friend of john)                                    32

==> (>> referent referent handler age referent handler friend referent
handler of john)

==> (>> eval referent definition eval referent definition handler age eval
referent definition handler friend eval referent definition handler of john)
```

[12] These tests where done during the execution of the program EXTRAK [Strickx87].

FLEXIBILITY

Theoretically, the KRS interpreter takes only a few different steps: finding a concept, finding a subject, taking a referent and evaluating a definition.

In the actual implementation however, there are quite a few more. This is caused by the following reasons:

- Infinite structures, resulting from the reflectivity of the concept-graph, must be shortcut. At a certain moment, the interpreter must stop creating structure and simulate the behavior of the shortcut structure in a different way.
- Shortcuts must be invisible. Hence, when a shortcut structure is explicitly requested, it must be created and stored at that time.
- The definition of the concept-graph is massive. The performance of the interpreter can be largely improved by simulating parts of its behavior instead of following the theoretical strategy. Again, this must happen in a inconspicuous way.

Along the KRS development process, the number of different steps the task-processor has to take into account has been continuously shrinking or growing. Many updates of the KRS implementation have been concerned with the interpretation cycle. This is the main reason why flexibility is such an important constraint.

3.2. Sketch of the Solution.

The *task-processor* is an agenda-based mechanism to implement an inherently complex recursive algorithm in an iterative way, while preserving the flexibility, maintainability and understandability of the recursive implementation.

3.2.1. Agenda-Based Systems.

This sub-section discusses the main advantages of agenda-based programming. It is based on the item about agenda-based systems in [Shapiro87]. [Charniak87] elaborates on LISP-implementations of agendas.

Agendas constitute a popular programming technique in Artificial Intelligence. One of the first AI-systems to use an agenda was the DENDRAL program [Feigenbaum71, Lindsay80], a program used to elucidate molecular structures. In DENDRAL, the agenda was mainly used to implement a breadth-first search with preference for particular branches. In the discovery program AM [Lenat76], an agenda was used to evaluate tasks prior to selection and execution. The elements of an agenda in KRL [Bobrow77] are queues of processes. It makes processes with a higher priority run earlier.

Agendas can be used for many reason. One reason is to allow one to look ahead for tasks to come and to reason about tasks prior to their execution. For example in production-systems one can reason about the possible benefit firing a rule can bring.

Agendas are also useful to extend the control-flow of LISP, by deciding dynamically at run-time instead of at coding-time upon the order of execution. Breadth-first search is the best known example of a process which can be most elegantly implemented with an agenda-mechanism.

For our purposes the most important advantage is that an agenda makes all steps in the system's execution explicit and concrete. This results in a highly modular and cleanly structured system.

3.2.2. The KRS Task-Processor.

The *task-processor* demonstrates an implementation methodology for implementing an inherently complex recursive process in an iterative way, while preserving optimal flexibility. It consists of an agenda and an iteration which executes all tasks on the agenda one by one, until there isn't any left.

The task-processor's agenda is not used to reason globally about sets of tasks. In fact, the agenda is merely used in a metaphorical sense, i.e. to provide better insight in the control flow. This is achieved by making all run-time execution steps explicit. A *task* represents one concrete step in the system's execution.

A different view of the task-processor is as a simulation of continuation based programming [Hewitt76]. Tasks can be (and are often) generated to represent the continuation of the processing.

To tune the task-processor to a particular application, one has to specify the contents of the tasks, and to define a function Process-Tasks. This function interprets individual tasks. Its argument-list must match with the contents of the tasks.

3.3. Functional Description.

3.3.1. The Task-Processor.

The task-processor processes tasks one by one for as long as there remain tasks on its agenda. It is started with one initial task on the agenda. The processing of one task may generate one or more other tasks on the agenda. The result of the last execution is returned.

start-processor *task* *[function]*

starts a task-processor with one initial-task on its agenda.

add-task *task* *[function]*

adds a task to the agenda of the active task-processor. The active task-processor is the one who is accessible by a dynamic scoping mechanism.

create-task *&rest task-components* *[function]*

creates a task, which is a list of all its components.

3.3.2. Customizing the Task-Processor for the KRS Interpretation Cycle.

To customize the task-processor for our needs, we need to define the contents of the Tasks, and the internal body of the function Process-Task which interprets the individual tasks.

A task is a triple containing an access-path, a concept and a concept-handler.

ACCESS-PATH

An access-path embodies the sequence of steps to be taken to go from the current concept in focus to the required result. A subject-name in an access-path denotes taking the filler of that subject. A list of a subject-name together with a set of concept-descriptions denotes a request for the filler of a subject-with arguments.

CONCEPT

The concept is the concept in focus. It is typically the result of a previously processed task. Occasionally, it can be a LISP-object, for example after computing the referent of a data-concept.

An empty access-path denotes that the current concept is the required result.

CONCEPT-HANDLER

Most of the time, the concept-handler is NIL. When it is not, it is a concept of which the concept in focus is the referent.

A simple request like in (33) is transformed into the first task shown in (34). (34) also shows some of the subsequent tasks generated to solve this request.

```
(>> referent age of john)                                          33

((referent age referent) <john name> nil)                        34
((age referent) <john> <john name>)
((referent referent) <age-subject of <john>>)
((referent) <number #52>)
```

Notice that the sequence of tasks depends on what has been cached before and how the subjects within the concept are represented. The sequence of tasks just given is generated when both the referents of John-Name and the age-subject of John were already cached, and when the age-subject is defined explicitly.

The function Process-Task distinguishes the following concrete actions:

- *Empty access path.* When the access-path of the task is empty, the task's concept is the result to be returned.
- *Evaluate a definition.* Some tasks have a LISP-form as concept and a request to evaluate it in the access-path. In this case the form is evaluated after retrieving the appropriate KRS environment from the concept-handler.
- *Referent computation.* Sometimes the tasks requests to compute the referent of its concept. In most cases, this referent is found by asking for the value of the relative-proposition which computes it. If this proposition does not yet exist, it is first created and stored.
- *Handler.* When the subject handler occurs in an access-path, it can be skipped if the following element is referent.
- *Subject arguments.* A task which requests the filler of a subject with arguments is translated into another task which will first lookup the subject and afterwards evaluate its definition with the arguments properly bound.
- *Explicit subject requests.* If there is a request for a subject-concept this is returned if it exists. If not, it is first created.
- *Subject filler computation.* A task which requests the filler of a subject, can be processed in two ways. If there is an explicit subject, a new task is generated to get that subject's referent. If there is a virtual subject, the value of the corresponding FPPD proposition is asked.

3.4. Implementation.

3.4.1. The Task-Processor.

The task-processor is a simple iteration as shown in (36). Its agenda is represented by a dynamically scoped variable *agenda* (35). It must be dynamic so that it can be accessed by the function Process-Task to push new tasks on it. It can not be a global variable since it is possible to have several task-processors run simultaneously.

```
(defvar *agenda*)                                                    35
```

```
(defun START-PROCESSOR (task)                                        36
  (do ((*agenda* (list task))
       (result))
      ((null *agenda*) result)
    (setq result (apply #'process-task (pop *agenda*)))))
```

The macro Add-Task (37) adds a new task to the Task-Processor's agenda.

```
(defmacro ADD-TASK (task)                                            37
  `(push ,task *agenda*))
```

3.4.2. Customizing the Tasks-Processor to implement the KRS interpretation cycle.

To customize the task-processor, we have to define the internal structure of a task and the function Process-Task.

A task is a three element list, created with the function Create-Task (38).

```
(defmacro CREATE-TASK (access-path concept concept-handler)          38
  `(progn (new-task-number)
          (list ,access-path ,concept ,concept-handler)))

(defmacro NEW-TASK-NUMBER ()
  `(incf *last-task-number*))
```

Process-Task binds the symbol *first-path* to the first element of the task's access-path which it uses to select the next step to be taken (39).

```
(defun PROCESS-TASK (access-path concept concept-handler &aux first-path)   39
  (setq first-path (first access-path))
  (cond
    ...))
```

The main part of the body of Process-Task is a large conditional, selecting the action to be taken next. There are several distinguished cases.

EMPTY ACCESS-PATH

When the access-path is empty, the requested result is the current concept.

```
((NULL ACCESS-PATH)                                                    40
 concept)
```

EVALUATING A LISP-FORM

An access-path starting with the LISP-symbol "eval" indicates that the current context is a LISP-form to be evaluated. The KRS environment in which it has to be evaluated is extracted from the concept-handler.

```
((EQ FIRST-PATH 'EVAL)                                                 41
 (add-task
  (create-task
   (cdr access-path)
   (krs-eval
    concept
    (distance-for-referent
     (cdr (concept-distance concept-handler)))
    (concept-krs-env concept-handler)
    (concept-lisp-env concept-handler))
   nil)))
```

COMPUTING A REFERENT

A request to compute a referent is executed by asking for the value of a referent-computation proposition. In case there is an explicit referent subject-concept, handling the request is postponed. If the next element in the access-path is "handler", the referent is not computed, but the subpath "referent handler" is just skipped.

```
((AND (EQ FIRST-PATH 'REFERENT)                                        42
      (NOT (EQ ATTRIBUTE-KEY (GET-ATTRIBUTE CONCEPT 'REFERENT) :SUBJECT))))
 (if (eq (second access-path) 'handler)
   (add-task
    (create-task
     (cddr access-path)
     concept
     concept-handler))
   (add-task
    (create-task
     (cdr access-path)
     (let* ((att (get-attribute concept 'referent))
            (referent
             (get-proposition-value
              (or (attribute-thing att)
                  (attribute-thing
                   (fill-attribute
                    att
                    (make-relative
                     #'restart-processor-to-eval-definition
                     (list concept))
                    :cached-referent
                    nil))))))
       referent)
     concept))))
```

A HANDLER REQUEST

An access-path with first element the LISP-symbol "handler" can only be processed if the next element is "referent", in which case the two of them are skipped. Otherwise, an error occurs.

```
((EQ FIRST-PATH 'HANDLER)                                                    43
 (if (eq (second access-path) 'referent)
   (add-task
    (create-task
     (cddr access-path)
     concept
     concept-handler))
   (krs-error "Impossible to take the handler of ~S" concept)))
```

SUBJECT-ARGUMENTS

To execute a request with arguments, first the subject-concept is retrieved, then the referent of its definition and the referent of its arguments. Finally the referent of its definition is evaluated while the arguments in the subject-concept are bound to the concepts described by the additional concept-descriptions in the request.

```
((LISTP FIRST-PATH)                                                          44
 (let* ((subject-concept (>>-pinned concept (list (first first-path) 'handler)))
        (subject-arguments (>>-pinned subject-concept '(arguments referent)))
        (subject-definition (>>-pinned subject-concept '(definition referent))))
   (create-task
    (cdr access-path)
    (krs-eval subject-definition
              (distance-for-referent (concept-distance subject-concept))
              (extend-krs-env
                (concept-krs-env subject-concept)
                subject-arguments
                (parse-concept-descriptions (cdr first-path) distance krs-env lisp-env))
              (concept-lisp-env subject-concept))
    subject-concept)))
```

A SUBJECT-CONCEPT REQUEST

The treatment of a request for a subject-concept depends on the internal structure of the subject. If there is an explicit subject or an explicit inherited subject, it is just returned. If there is a virtual-subject or an inherited virtual-subject, it is made explicit.

```
((and (eq (second access-path) 'handler)                                    45
      (not (eq (third access-path) 'referent)))
 (let* ((att (find-attribute concept first-path))
        (thing (attribute-thing att))
        (key (attribute-key att))
        (v-sub (attribute-v-sub att)))
   (unless thing
     (progn
       (setq thing
             (create-inheriting-subject-proposition
              concept first-path))
       (setq key :inheriting-subject)))
   (case key
     (:subject
      (add-task
       (create-task
        (cddr access-path)
        (get-proposition-value thing)
        nil)))
     (:virtual-subject
      (add-task
       (create-task
        (cddr access-path)
        (or v-sub (make-virtual-subject-explicit concept first-path))
        nil)))
     (:inheriting-subject
      (add-task
       (create-task
        (cddr access-path)
        (or v-sub (make-inheriting-subject-explicit concept first-path))
        nil))))))
```

A SUBJECT-FILLER REQUEST

To compute a subject-filler there are again three cases. If there is an explicit subject-concept, its referent must be computed. If there is a virtual-subject, its value must be asked. If there is an inheriting-subject, either its referent must be computed or its value asked depending on whether the original subject is explicit or virtual.

```
(T                                                                    46
 (let*
   ((att (find-attribute concept first-path))
    (thing (attribute-thing att))
    (key (attribute-key att)))
   ;; First check if we must create an inheriting subject.
   (unless thing
     (progn
       (setq thing
             (create-inheriting-subject-proposition
              concept first-path))
       (setq key :inheriting-subject)))
   (case key
     (:subject
      (add-task
       (create-task
        (cons 'referent (cdr access-path))
        (get-proposition-value thing)
        nil)))
     (:virtual-subject
      (add-task
       (create-task
        (cdr access-path)
        (get-proposition-value (get-proposition-value thing))
        nil)))
     (:inheriting-subject
      (let ((inherited-structure (get-proposition-value thing)))
        (if (proposition-p inherited-structure)
          (add-task
           (create-task
            (cdr access-path)
            (get-proposition-value inherited-structure)
            nil))
          (add-task
           (create-task
            (cons 'referent (cdr access-path))
            inherited-structure
            nil))))))))
```

3.5. Example.

This example shows the tasks which are generated when requests about the concepts shown in (47) are executed.

```
(defconcept PERSON                                                    47
  (birthyear (a number))
  (age (a number
         (definition
           [form (- (>> referent of current-year)
                    (>> referent birthyear))])))))


(defconcept JOHN
  (a person
    (birthyear [number 1961])
    ((adult-p
       (a subject
         (definition [form (>> of true)])))))))
```

Figure 12 shows the series of tasks generated when (48) is executed. Figure 13 shows the tasks generated when the birthyear is requested a second time. Notice that for this request the second reply does not require less tasks. However, the resulting concept is created by asking for the value of the virtual subject the first time, while it is directly returned the second time. Notice that the task-numbers are not successive. The reason for this is that to determine the pname of the concepts printed, there are requests sent too. These are not shown by the task-processor-tracer since this would lead to an endless regression.

```
-->  (>> birthyear of john)                                          48
<--  <number 1961>
```

```
? (>>-trace-processor birthyear of john))
248:     ((BIRTHYEAR) <JOHN> NIL)
251:     (NIL <NUMBER 1961> NIL)
<NUMBER 1961>
```

Fig 12: Computing birthyear of John.

```
? (>>-trace-processor birthyear of john))
287:     ((BIRTHYEAR) <JOHN> NIL)
290:     (NIL <NUMBER 1961> NIL)
<NUMBER 1961>
```

Fig 13: Computing birthyear of John after caching.

Figure 14 shows the series of tasks generated when (49) is executed. Figure 15 shows the tasks generated when the adult-p is requested a second time. Since there is an explicit adult-p-subject, its filler is computed by computing its referent. The

indentation in the output of tasks denotes a recursive application of the task-processor, in this case by calling the function Restart-Processor-To-Eval-Definition. In the second request, this referent is not recomputed.

```
--> (>> adult-p of john)                                                    49
<-- <number 1961
```

```
? (>>-trace-processor adult-p of john))
319:     ((ADULT-P) <JOHN> NIL)
322:     ((REFERENT) <ADULT-P-SUBJECT OF <JOHN>> NIL)
377:      ((DEFINITION REFERENT EVAL) <ADULT-P-SUBJECT OF <JOHN>> NIL)
386:      ((REFERENT EVAL) <FORM (>> OF TRUE)> NIL)
416:      ((EVAL) (>> OF TRUE) <FORM (>> OF TRUE)>)
424:       (NIL <TRUE> NIL)
430:      (NIL <TRUE> NIL)
436:     (NIL <TRUE> <ADULT-P-SUBJECT OF <JOHN>>)
<TRUE>
```

Fig 14: Computing adult-p of John.

```
? (>>-trace-processor adult-p of john))
457:     ((ADULT-P) <JOHN> NIL)
460:     ((REFERENT) <ADULT-P-SUBJECT OF <JOHN>> NIL)
469:     (NIL <TRUE> <ADULT-P-SUBJECT OF <JOHN>>)
<TRUE>
```

Fig 15: Computing adult-p of John after caching.

Figure 16 shows the series of tasks generated when (50) is executed. Figure 17 shows the tasks generated when the age is requested a second time. It shows that in the first request the definition of the age-subject is searched within the concept Person. The concept printed "<number #1>" is the age of John. Once the computation of its referent is started, it is printed "<number computing>". In the second request, this concept is printed "<number 27>", since its referent is cached.

```
--> (>> age of john)                                                       50
<-- <number 1961
```

```
? (>>-trace-processor referent age of john))
490:     ((AGE REFERENT) <JOHN> NIL)
493:      ((TYPE) <JOHN> NIL)
496:      (NIL <PERSON> NIL)
499:     ((REFERENT) <NUMBER #1> NIL)
532:      ((DEFINITION REFERENT EVAL) <NUMBER COMPUTING> NIL)
551:      ((REFERENT EVAL) <FORM (- (>> REFERENT OF CURRENT-YEAR)
                                    (>> REFERENT BIRTHYEAR))> NIL)
581:      ((EVAL) (- (>> REFERENT OF CURRENT-YEAR)
                     (>> REFERENT BIRTHYEAR))
                  <FORM (- (>> REFERENT OF CURRENT-YEAR)
                           (>> REFERENT BIRTHYEAR)))>)
589:       ((REFERENT) <CURRENT-YEAR> NIL)
615:       (NIL 1988 <CURRENT-YEAR>)
622:       ((BIRTHYEAR REFERENT) <JOHN> NIL)
625:       ((REFERENT) <NUMBER 1961> NIL)
632:       (NIL 1961 <NUMBER 1961>)
638:      (NIL 27 NIL)
638:     (NIL 27 <NUMBER COMPUTING>)
27
```

Fig 16: Computing age of John.

```
? (>>-trace-processor referent age of john))
645:     ((AGE REFERENT) <JOHN> NIL)
648:     ((REFERENT) <NUMBER 27> NIL)
655:     (NIL 27 <NUMBER 27>)
27
```

Fig 17: Computing age of John after caching.

4. INHERITANCE.

Inheritance in KRS is defined as follows:

A concept inherits all subjects of its type except those it locally overrides.

A concept locally overrides a subject if it has a local subject with the same name as one of the subjects of the concept's type.

4.1. Problem Description.

4.1.1. Single versus Multiple Inheritance.

Multiple inheritance occurs when different retrieval tasks within one system require a different organization of the retrieval hierarchies [Touretzky86]. Multiple inheritance lately grew into a thorny issue in the object-oriented system community. The reason for this opposition is the complexity it has led to, both in semantic networks [Brachman85b] and in object-oriented programming languages [Moon86]. [America87] argues that many of these problems originate from the unification of two different things: *inheritance* and *subtyping*. The former, he argues, is a mechanism to share code, the latter is a conceptual subtype hierarchy. It is perfectly possible that this gives two hierarchies which can be mapped onto one another but this should not be necessarily so.

4.1.2. Class-Based and Prototype-Based Inheritance.

Inheritance systems can be classified in two distinct categories: those using *class-based* inheritance and those using *prototype-based* inheritance.

In a class-based inheritance strategy, inheritance (or information sharing) is between classes. All objects belong to exactly one class from which they extract all information (local or inherited). The standard example of a class-based object-oriented system is Smalltalk-80 [Goldberg83].

In a prototype-based inheritance strategy, any object can serve as prototype for other objects. Arguments in favor of prototype-based inheritance can be found in [Lieberman86a, Borning86].

So in the former strategy, there is a step from the object to its class which differs fundamentally from the inheritance step between a class and its super-class(es). This distinction is at the basis of the ObjVlisp model [Briot86]. This model allows representing classes as first-class objects, i.e. as instances of the class Class, without needing a complex bootstrap phase.

[Lieberman86b] distinguishes inheritance and delegation as two possible ways to implement information sharing. A *delegation strategy* forwards messages it can not reply to to other objects in the hierarchy. An *inheritance strategy* on the other hand searches itself in the information tables of the parent objects. Initially Lieberman has argued that inheritance goes along with class-based systems while delegation goes along with prototype systems. This however has turned out differently [Cook88]. The language SELF [Ungar87] for example demonstrates information sharing with inheritance in a prototype-based organization. [Stein87] argues that delegation is not inherently more powerful than inheritance.

4.1.3. Caching and Copying.

In general, the benefits of inheritance are assumed to be twofold: (i) it enhances modularity, i.e. code does not have to be duplicated, and (ii) it enhances efficiency, i.e. data is shared. In KRS however, relative definite descriptions and caching lay a constraint on the possibilities for data-sharing. Take the following example:

```
(defconcept PERSON
  (father (a person
            (wife (a person))))
  (mother (>> wife father)))

(defconcept JOHN                          (defconcept GEORGE
  (a person                                 (a person
    (father george)))                         (wife mary)))
```

The two requests for the mothers of Person and John (51) return different results. This is because the definition of the mother-subject is context-dependent, i.e. it depends on the concept to which the request was sent. Since those results have to be cached, they have to be cached separately for each concept the requests is asked for.

```
-->  (>> mother of person)                                          51
<--  <person #107>

-->  (>> mother of john)
<--  <mary>
```

Consequently, the mother-subject of Person and the mother-subject of John have different referents and can thus not be the same subject-concepts (52).

```
-->  (get-referent (>> handler mother of person))                   52
<--  <person #107>

-->  (get-referent (>> handler mother of john))
<--  <mary>
```

Moreover, those subject-concepts have themselves subjects with different fillers, for example their owner-subject (53).

```
--> (>> owner handler wife of person)                                    53
<-- <person>

--> (>> owner handler wife of john)
<-- <john>
```

All the arguments above indicate that inheritance in KRS does not allow concepts to share subjects. What is shared is the description of subjects. In particular, when the concept John inherits the wife-subject of Person, it takes the description of the wife-subject of Person and reinterprets this description in a different KRS environment. Hence, an inherited subject is a copy of the original subject.

4.1.4. Lexical Scope.

The KRS manual gives the following definition of lexical-scope:

> *The lexical-scope of a description d is the concept described by the global (outermost) description d is a part of.*
>
> *If there is no inheritance, the context of a relative-request is the lexical-scope of this request.*
>
> *Whenever a concept A inherits a subject-description from a concept B, and B is the lexical-scope of relative-requests in the original subject-description, then A becomes the lexical-scope of the inherited relative-requests.*

The computation of the lexical-scope of an inherited description turned out difficult to implement, especially when additive-inheritance is involved. It is easy when a subject is inherited from a concept which is its own lexical-scope. In that case, the lexical-scope of the inherited subject becomes the inheriting concept.

(54) illustrates this case. The concept John inherits its age-subject from the concept Person. The concept Person is its own context and hence John is the context of the inherited age-subject (figure 18).

```
(defconcept PERSON                                                       54
  (age (a number
         (definition
           [form (- (>> referent of current-year)
                    (>> referent birthyear([])))])

(defconcept JOHN
  (a person
    (birthyear [number 1961])))
```

Also in (55), when the father of John inherits its age-subject, it becomes itself the lexical-scope of the inherited subject (figure 19).

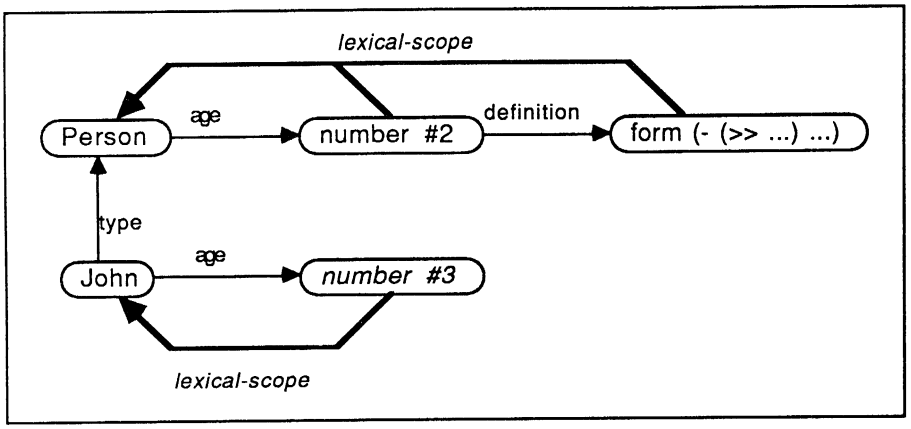


Fig 18: Inheriting-concept = lexical-scope (1).

```
(defconcept JOHN                                                    55
  (a person
    (birthyear [number 1961])
    (father (a person
              (birthyear [number 1930])))))
```

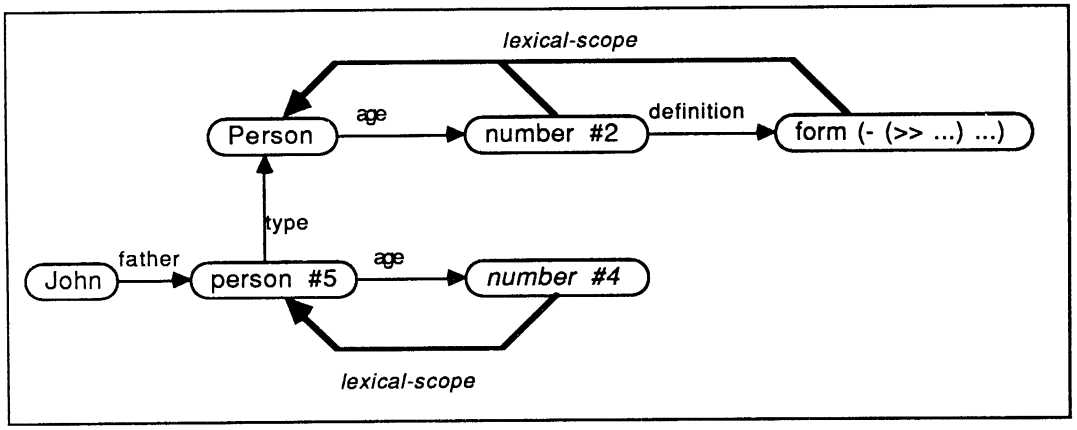


Fig 19: Inheriting-concept = lexical-scope (2).

When a subject is inherited from a concept which is not its own lexical-scope, it is more difficult to determine the new scope. (56-57) shows such a typical case, i.e. with additive inheritance. The first-born of Jones-Twin inherits its birthyear-subject from the first-born of Twin. The lexical-scope of the latter is the concept Twin. Hence, the lexical-scope of the first-born of Jones-Twin must be the concept Jones-Twin who inherits from Twin (figure 20). When however, instead of the birthyear-subject the age-subject is inherited, the lexical-scope of the inherited age-subject is the first-born of Twin itself. Although this is intuitively obvious, computationally it is not. Deeply nested descriptions can be constructed in which each concept, from the

outermost to the innermost, is the lexical-scope of a particular subject inherited by the innermost concept.

```
(defconcept TWIN                                                    56
  (birthday (a date))
  (first-born
    (a person
      (birthyear (>> year birthday))))
  (second-born
    (a person
      (birthyear (>> year birthday))))))
```

```
(defconcept JONES-TWIN                                              57
  (a twin
    (first-born
      (name [string "John"]))
    (second-born
      (name [string "Mary"]))))
```

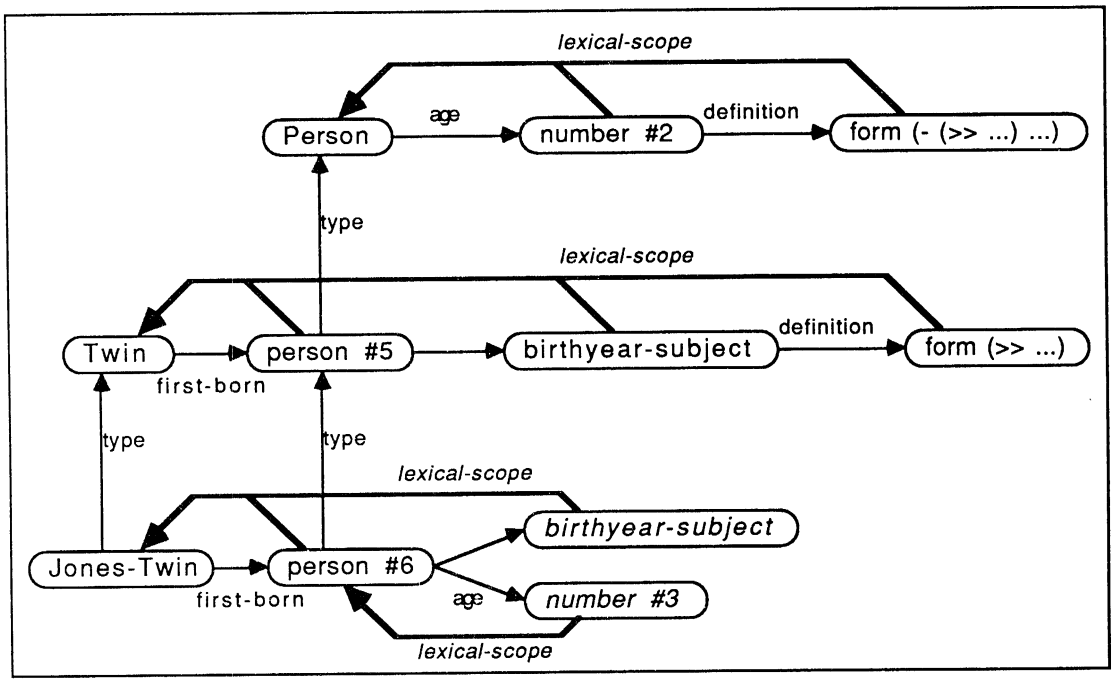


Fig 20: Lexical-scope-computation with additive inheritance.

The problem can not be solved by testing in the sequence of nested concepts which one inherits from the current lexical-scope. This is illustrated with example (58). The wife of the father of John is the mother of John. The wife of the father of the father of John is the mother of the father of John. The same holds for any ancestor of John. Nevertheless, the lexical scope of both relative definite-descriptions is the concept Person, who is at the same time the type of all the nested ancestors (figure 21).

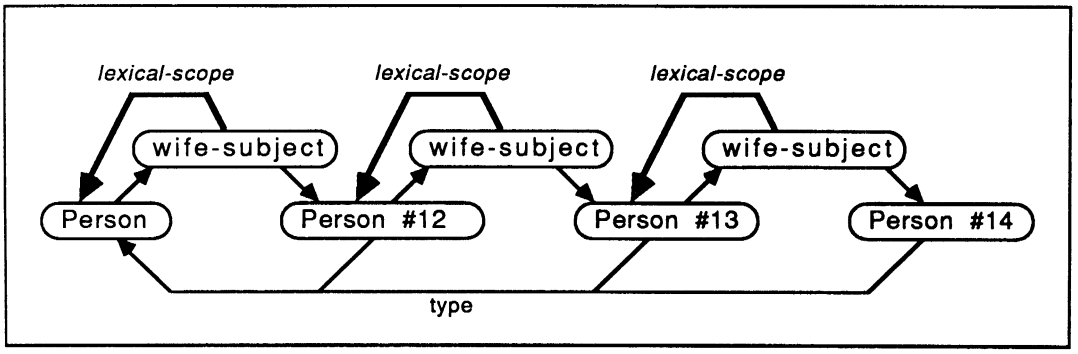


Fig 21: Lexical-scope-computation.

```
(defconcept PERSON                                                    58
  (father (a person
            (wife (>> mother))))
  (mother (a person
            (husband (>> father)))))
```

4.2. Sketch of the Solution.

4.2.1. Simple Single Inheritance.

KRS opts for a simple single inheritance mechanism for simplicity reasons, hereby deliberately giving up particular extras like inheritance from multiple orthogonal components. Multiple inheritance often leads to complicated and conceptual unclear architectures. We think that simple inheritance contributes to the conceptual cleanliness of representations.

Where multiple retrieval hierarchies are required, they can be modeled explicitly as well as the retrieval tasks operating on them. (59) illustrates this with an example. The concept Door-Of-Vehicle retrieves its functions-subject using normal inheritance, i.e. via its type. At the same time, it retrieves its color via the part-of-subject.

```
(defconcept DOOR                                                    59
  (functions [sequence entrance exit]))

(defconcept DOOR-OF-VEHICLE
  (a door
    (part-of (a vehicle))
    ((color (a delegating-subject
              (delegate-to (>> part-of)))))))

(defconcept CAR
  (a vehicle
     (color (a color))
     (door (a door-of-vehicle
             (part-of (>>))))))

(defconcept MY-CAR
  (a car
    (color gray)))


-->  (>> color door of my-car)                                      60
<--  <gray>
-->  (>> functions door of my-car)
<--  <sequence <entrance> <exit>>
```

[Van Marcke88b] argues in favor of making inheritance schemes explicit.

4.2.2. Prototype-Based Inheritance.

Following many of its ancestor languages such as ARLO, LOOPS or KRL, KRS supports prototype inheritance. Any concept can be the type of other concepts. This provides the flexibility which is required for Artificial Intelligence programming.

Three aspects substantiate this argument. (i) At all times a specialization of a given concept can be made for example to make a small deviation. (ii) For concepts for which there is only one instance, one needs not make a class first of which only one instance is created afterwards. (iii) All parts of the system speak the same language and can be inspected and/or questioned in the same manner.

4.2.3. Using FPPD to Copy Inherited Structure.

We argued before that an inherited subject must be a copy of the original subject. To make this copy, we must distinguish two cases; either there exists an explicit subject-concept or the subject is represented by a virtual-subject.

SUBJECT-CONCEPT

When the original subject is represented by an explicit subject-concept, its copy is made by creating a new subject-concept whose type is the original subject-concept. The new subject-concept is called an *inheriting-subject.*

The inheriting-subject thus recursively inherits the subjects of the original subject-concept (figure 22). This recursion bottoms out when there are virtual-subjects.

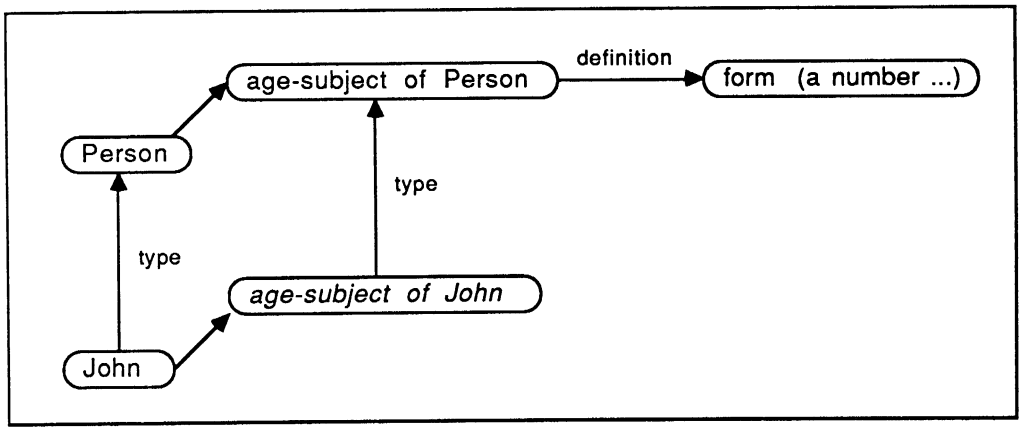


Fig 22: An inheriting subject-concept.

VIRTUAL-SUBJECT

A virtual-subject is an FPPD-proposition. Its copy is a new FPPD proposition containing the same function as the original proposition and almost the same context. In this context, only the distance is changed, such that inherited relative definite-descriptions take the correct context (figure 23).

Since inheriting a subject means copying this subject, we must also ensure that the copy remains consistent with the original. For this purpose, the inherited structure, i.e. either the inheriting-subject or the copy of the virtual-subject is computed by an FPPD-proposition.

This results in the following structure. The key of an attribute for an inherited-subject is :inherited-subject. The thing of this attribute is always an FPPD-proposition. The value of this proposition is either a virtual-subject, i.e. a new FPPD-proposition, or an inheriting-subject, i.e. a subject-concept whose type is the

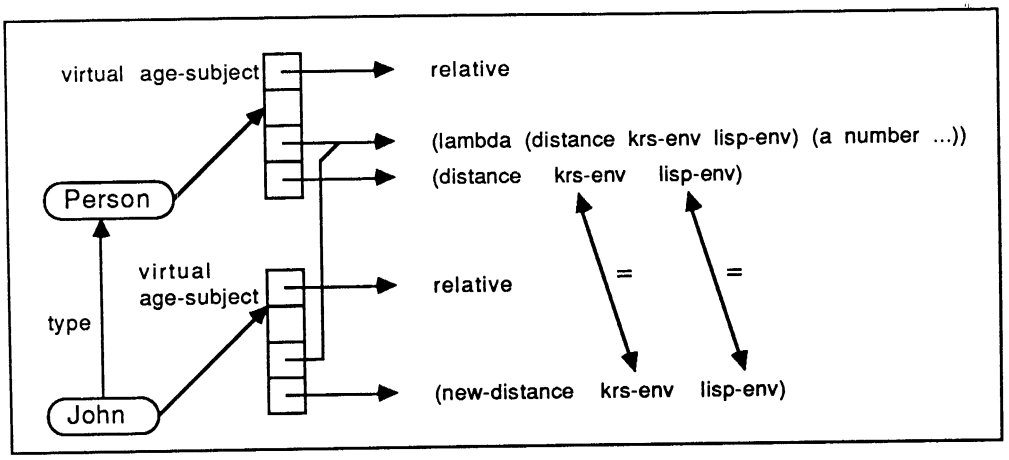


Fig 23: An inherited virtual subject.

original subject-concept.

4.2.4. Lexical Scope and Distances.

The lexical-scope of an inherited subject is computed by matching distances. The distance of a concept is a pattern of concepts out of which the concept's lexical-scope can be extracted. When a subject is inherited, the distance of the inherited subject is the result of matching the distance of the concept the subject is inherited from with the distance of the inheriting-concept.

The matching process basically rebuilds the same structure as the structure of the distance of the owner of the original structure (since this is the structure out of which the original lexical-scope can be computed). The elements of the new structure however are not the concepts in the original distance, but the corresponding concepts in the distance of the inheriting concept. Figure 24 illustrates the matching of the distance of the concept Twin with the distance of the concept Family-Jones described in (61).

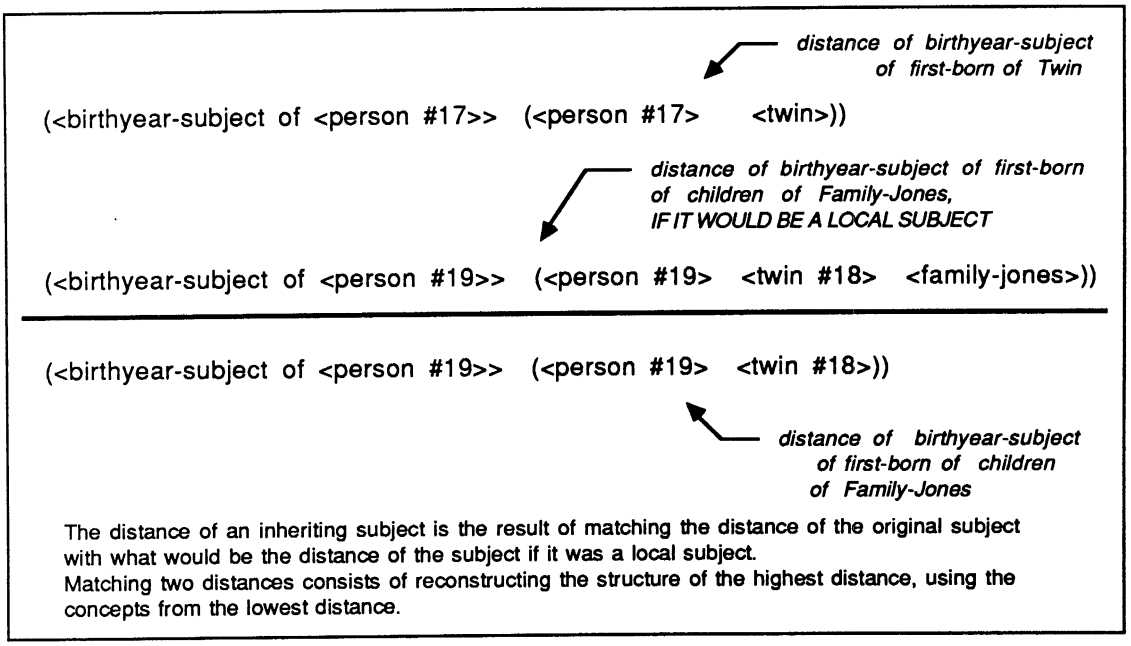


Fig 24: Matching of distances.

```
(defconcept FAMILY-JONES                                    61
  (a family
    (children
      (a twin
        (birthday [data march 10 1961])
        (first-born
          (name [string "John"]))
        (second-born
          (name [string "Mary"])))))))
```

4.3. Functional Description.

4.3.1. Functions.

create-inheriting-subject-proposition *concept access* *[function]*

creates an FPPD proposition in which a copy of the original inherited structure will be computed, and stores this proposition in the concept's memory.

compute-inheriting-subject *concept access* *[function]*

makes the copy of the original inherited subject.

find-subject-to-inherit-from *concept access* *[function]*

searches the original subject.

4.3.2. Distances.

The *distance* of a concept is a pattern of concepts. This pattern is a list specified by the following constraints.

1 Every element of a distance is a concept, except the last one.

2 The last element of a distance is either a concept or a new distance.

3 The first element of a concept's distance is the concept itself.

4 The context of a distance is the last element if this is a concept, otherwise it is this last element's context.

5 If a concept occurs in a distance and the next element in the distance is also a concept, then the former concept is the filler of a subject of the latter concept.

6 If a concept occurs in a distance and the next element in the distance is another distance, then this concept is a subject-concept and the next element is the distance of this subject-concept's owner.

(62) shows the distance of the first-born of the children of Family-Jones. (63) shows the distance of the definition of the name-subject of the first-born of the children of family-jones.

```
(<person #16> <twin #17> <family-jones>)                                    62
```

```
(<form #$18> <name-subject of <person #16>>                                 63
             (<person #16> <twin #17> <family-jones>))
```

4.4. Implementation.

4.4.1. Computing the Data-Structure.

The function Create-Inheriting-Subject-Proposition (64) creates a relative-proposition to compute the inherited structure. This proposition is stored onto the concept's memory.

```
(defun CREATE-INHERITING-SUBJECT-PROPOSITION (concept access)                64
  (let
    ((inheriting-subject-proposition
       (make-relative
         #'compute-inheriting-subject
         (list concept access))))
    (fill-attribute
      (get-attribute concept access)
      inheriting-subject-proposition
      :inheriting-subject
      nil)
    inheriting-subject-proposition))
```

The function Compute-Inheriting-Subject is given in (65). It first retrieves the original subject. Then a new inherited subject or a new virtual-subject is created and stored on the corresponding concept-attribute.

```
(defun COMPUTE-INHERITING-SUBJECT (concept access)                            65
  (let ((original-subject (find-subject-to-inherit-from concept access)))
    (cond
      ((null original-subject)
       nil)
      ((concept-p original-subject)
       (compute-explicit-inheriting-subject
         concept
         access
         original-subject)
      (t
       (compute-virtual-inheriting-subject
         concept
         access
         original-subject))))
```

If the original subject is represented explicitly, an inheriting subject is created (66). If it is a virtual-subject, it is copied in a new virtual-subject (67).

```
(defun COMPUTE-EXPLICIT-INHERITING-SUBJECT (concept access original-subject)   66
  (let*
    ((subject
       (make-instance-from-description
         '((type ,original-subject))
         (list
           (analyze-distances
             (concept-distance concept)
             (second (concept-distance original-subject))))
         (concept-krs-env original-subject)
         (concept-krs-env original-subject)))
     (referent-proposition
       (make-relative
          #'restart-processor-to-eval-definition
          (list subject))))
    (add-referent-proposition subject referent-proposition)
    (new-meta-info-element subject 'access access)
    subject))


(defun COMPUTE-VIRTUAL-INHERITING-SUBJECT (concept access original-subject)    67
  (declare (ignore access))
  (let* ((original-context (fppd::proposition-context original-subject))
         (original-function (fppd::proposition-to-compute original-subject))
         (original-subject-definition (caddr original-function)))
    (if (member (car original-subject-definition) '(find-concept >>-pinned-save))
      original-subject
      (make-proposition
       :type :relative
       :to-compute (fppd::proposition-to-compute original-subject)
       :context
       (list
        (analyze-distances
         (concept-distance concept)
         (first original-context))
        (second original-context)
        (third original-context))))))
```

The function Find-Subject-To-Inherit-From finds the original subject in the type-hierarchy. When searching upwards in the type-tree, a new inheriting-proposition must be created at each level. This way the data-dependency network is correctly constructed.

```
(defun FIND-SUBJECT-TO-INHERIT-FROM (concept access)                    68
  (unless (eq concept *summum-genus*)
   (let* ((type (concept-type concept))
          (att (find-attribute type access))
          (key (attribute-key att))
          (thing (attribute-thing att)))
    (case key
       ((nil)
        (let ((inherited-structure
                (get-proposition-value
                  (create-inheriting-subject-proposition type access))))
          (if (proposition-p inherited-structure)
              inherited-structure
              (concept-type inherited-structure))))
       (:inheriting-subject
         (let ((inherited-structure
                 (get-proposition-value thing)))
          (if (proposition-p inherited-structure)
              inherited-structure
              (concept-type inherited-structure))))
       ((:subject :virtual-subject)
         (get-proposition-value thing))))))
```

4.4.2. Distance Computation.

The functions Analyze-Distances (69) takes two distances as input and creates a new distance having the same structure as its second input but containing the corresponding elements of its first input.

```
(defun ANALYZE-DISTANCES (local-distance inherited-distance)             69
  (cond
    ((null local-distance) nil)
    ((and (concept-p (first inherited-distance))
          (null (cdr inherited-distance)))
     (list (first local-distance)))
    ((and (concept-p (first local-distance))
          (concept-p (first inherited-distance)))
     (cons (first local-distance)
           (analyze-distances (cdr local-distance)
                              (cdr inherited-distance))))
    ((and (listp (first local-distance))
          (listp (first inherited-distance)))
     (list (analyze-distances (car local-distance)
                              (car inherited-distance))))
    (t (krs-error "Incomplete instantiation."))))
```

The function Get-Pinpoint-Out-Distance (70) retrieves the context out of a given distance.

```
(defun GET-PINPOINT-OUT-DISTANCE (distance)                             70
    (cond
      ((null distance)
       nil)
      ((cdr distance)
       (get-pinpoint-out-distance (cdr distance)))
      ((listp (first distance))
       (get-pinpoint-out-distance (first distance)))
      (t (first distance))))
```

4.5. Example.

```
(defconcept PERSON                                                    71
  (birthyear (a number))
  (age (a number
         (definition [form (- (>> referent of current-year)
                              (>> referent birthyear))])))
  ((adult-p (a subject
              (definition [form (if (> 20 (>> referent age))
                                    (>> of true)
                                    (>> of false))])))))


(defconcept JOHN                                                      72
  (a person
    (birthyear [number 1961])))


-->  (>> adult-p of john)                                            73
<--  <true>
```

Figure 25 shows the adult-p-subject of John and its memory. Notice that it is an inherited subject-concept and that its type is the original subject-concept: the adult-p-subject of Person

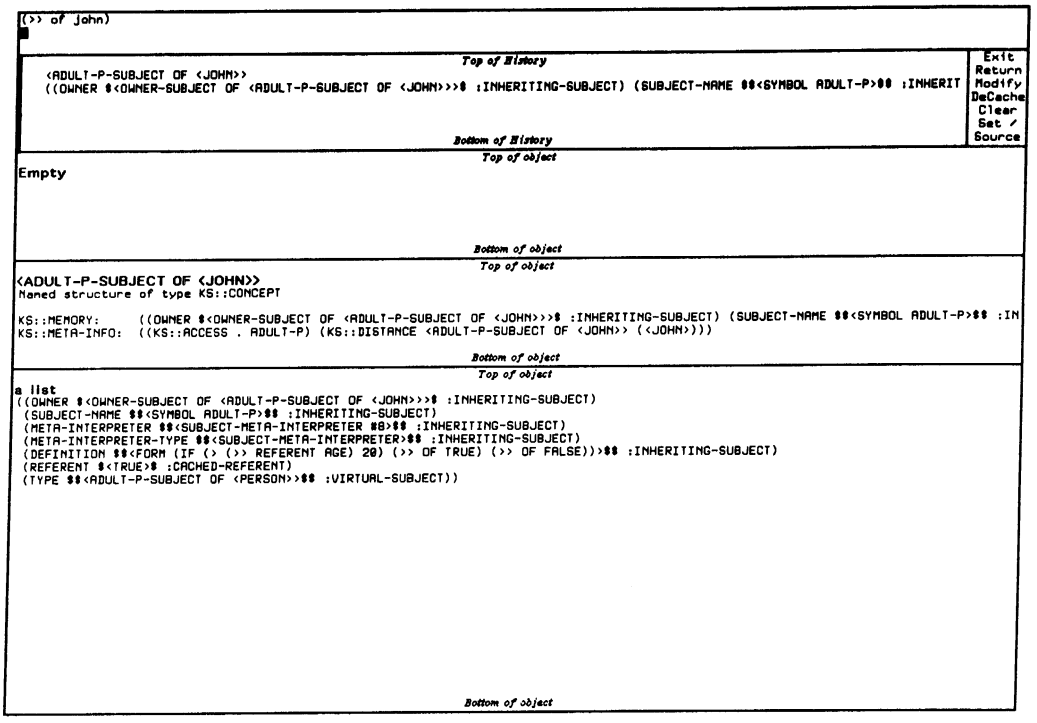


Fig 25: An inherited explicit subject-concept.

Figure 26 shows in the top-window the virtual-subject representing the original age-subject of Person. In the middle, it shows the FPPD-proposition to inherit the age subject for John. Its value is a new virtual-subject, i.e. a copy of the original one.

This copy is shown in the bottom-window

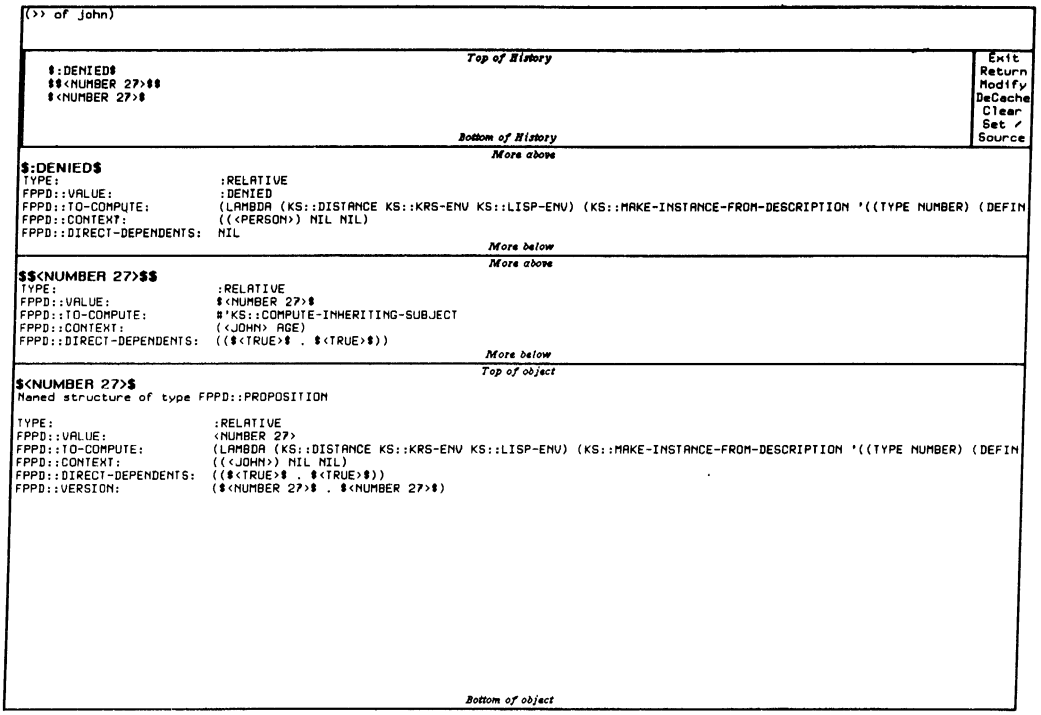


Fig 26: An inherited virtual subject.

5. THE KRS ENVIRONMENT.

The KRS environment contains data a concept needs to proper reply to particular requests. It consists of data to compute the concept's lexical scope and to interpret concept-variable-descriptions and tilde-descriptions. A concept picks up this data from the LISP environment when it is created.

A KRS environment consists of a *distance*, a *krs-env*, and a *lisp-env*.

5.1. Problem-Description.

5.1.1. Distance.

The first element of the KRS environment is the concept's distance. It is used by the concept to determine its lexical scope. The exact role of distances was explained in the previous section.

5.1.2. Krs-Env and Lisp-Env.

As a result of the lazy evaluation mechanism, certain descriptions are not interpreted the moment they are typed in. Therefore we must make sure that all data which is required for the correct interpretation is still accessible when the descriptions are finally interpreted.

The set of concept-variable bindings for example is data of this kind. In example (74), the age of the father of John is not yet computed by the evaluation of this description. It will be computed when (or if) the request in (75) is evaluated. To preserve the binding of the concept-variable '?age', it is stored in the second part of the KRS environment of the concept John: the krs-env. This krs-env will be passed from John via the father of John and this one's age-subject to the definition of the age-subject of the father of John. This way it can be used when the referent of this definition is evaluated.

```
(clet ((?age [number 27]))                                    74
  (defconcept JOHN
    (a person
      (father (a person
                (age ?age))))))


(>> age father of john)                                       75
```

When tilde-expressions are used, the according parts of the LISP environment need to be stored in the same way and for the same reason. In (76) for example, the value of the LISP-variable 'x' will be needed when it is already removed from the stack. This data is stored in the third element of the KRS environment: the lisp-env.

```
(let ((x 5))                                                    76
  #~(a person
      (age
        (a number
          (definition
            [form (* 2 ~x)])))))
```

5.2. Sketch of the Solution.

5.2.1. Picking up the KRS environment.

The KRS primitives which create new concepts (defconcept, a, an) are macros. The macro-expanded version of a call of one of these primitives has additional arguments via which the KRS environment is given along. From here on, the KRS environment is given trough all levels until the function Make-Concept is called. This function stores the KRS environment in the concept's meta-info-field (section 2.4.1).

(77) shows the macro-expansion of the defconcept-description for John given above. It has arguments via which the current bindings of the lisp-variables krs-env and lisp-env are picked up.

```
(defc 'john                                                    77
      '((a person
          (father
           (a person
             (age ?age)))))
      krs-env
      lisp-env)
```

(79) shows the macro-expansion of (78). It has arguments to pick up the current bindings of distance, krs-env and lisp-env.

```
(a person                                                      78
  (age ?age))


(make-instance-from-description                                79
   '((type person) (age (krs-variable-eval 'age krs-env)))
   distance
   krs-env
   lisp-env)
```

5.2.2. Constructing the Lisp-Env.

The way distances are constructed is explained in section 4.2.4. The construction of krs-env is straightforward. The construction of lisp-env however is not. To find which part of the LISP environment to store, the reader macro '#~' parses the entire description. All tilde-descriptions it encounters are moved to the beginning of the concept-expression, such that they can be interpreted in the correct LISP environment and stored in lisp-env. At the original location of the tilde-expression, a symbolic reference is left, which is used to recover the desired description out of lisp-env when needed.

(81) shows the result the reader-macro #~ in (80) produces. By moving the description "x" to the front, it can be evaluated while its binding is still known. The result is stored in the association-list lisp-env. At the place where the variable was

originally used, there is now a reference to retrieve the variable's binding out of this association-list.

```
        (age (a number                                                    80
               (definition
                 [form (* 2 ~x))


(let ((lisp-env (list (cons '#:g4435 x))))                                81
  (a person
    (age
     (a number
       (definition
        [form (* 2 (lisp-rpl '#:g4435 lisp-env))])))))
```

5.3. Functional Description.

The creation of a concept typically happens during the evaluation of the referent of a definition. During this evaluation, the LISP symbols *distance*, *krs-env* and *lisp-env* are bound to the current distance, the current krs-env and the current lisp-env. The concept computes its own distance by consing itself onto the current distance. The krs-env stores all concept-variable bindings this concept must have access to. The lisp-env stores parts of the lisp-environment.

The expanded version of the macro's Defconcept, A and An have additional arguments which are automatically bound to the bindings of *distance*, *krs-env* and *lisp-env*. Also the reader-macro '[' generates a form which accesses the bindings of the three variables.

The function Pre-Parse-For-Hooks-To-Lisp is called by the reader-macro '#~'. It takes as input a concept-description and returns two outputs. The first output is a transformed description in which all tilde-descriptions are replaced by symbolic references to lisp-env. The second result is a list of couples. The second elements of those are the removed tilde-descriptions. The first elements are the lisp-symbols used in the symbolic references to lisp-env. With this list of couples, lisp-env can be constructed such that it can be correctly consulted afterwards.

5.4. Implementation.

The macro Defconcept translates to a call of the function Defc.

```
(defmacro DEFCONCEPT (concept-name &body concept-description)                82
  `(defc ,(if (atom concept-name)
              `',concept-name
              concept-name)
         ',concept-description
         krs-env lisp-env))
```

The macro A translates to a call of the function Make-Instance-From-Description.

```
(defmacro A (type &body subject-descriptions)                                83
  `(make-instance-from-description
     ',(append `((type ,type)) subject-descriptions)
     distance
     krs-env
     lisp-env))
```

The macro with-tildes is called by the reader-macro '#~'. During macro-expansion, the function pre-parse-for-hooks-to-lisp is called. This function reformulates the description such that tilde-expressions are evaluated at the beginning of the descriptions.

```
(defmacro with-tildes (expression)                                           84
  (multiple-value-bind (form l-list)
     (pre-parse-for-hooks-to-lisp expression)
    `(let ((lisp-env ,(cons 'list l-list)))
       ,form)))


(defun PRE-PARSE-FOR-HOOKS-TO-LISP (description)                             85
  (cond
    ((atom description)
     (values description nil))
    ((eq (first description) 'lisp-eval)
     (let ((dummy-symbol (gensym)))
       (values
         (list 'lisp-rpl `',dummy-symbol 'lisp-env)
         (list (list 'cons `',dummy-symbol (second description))))))
    ((and (cdr description)(atom (cdr description)))
     (values description nil))
    (t
     (let ((form)(l-list))
       (dolist (d (reverse description))
         (multiple-value-bind (sub-form sub-l-list)
             (pre-parse-for-hooks-to-lisp d)
           (push sub-form form)
           (setq l-list (append sub-l-list l-list))))
       (values form l-list)))))
```

5.5. Example.

5.5.1. Krs-Env.

The add-subject of Number (86) has arguments. When the filler of such a subject is computed, the bindings of the arguments are stored on its krs-env (87). It will be retrieved from there the moment the referent of this filler is computed.

```
(defconcept NUMBER                                                    86
  (a data-concept
    ((add
       (a subject
         (arguments [args ?add-what])
         (definition
           [form (a number
                   (definition
                     [form (+ (>> referent)
                              (>> referent of ?add-what))])])))))))


-->  (>> (add [number 2]) of [number 3])                              87
<--  <number #70>

-->  (concept-krs-env *)
<--  ((add-what . <number 2>))
```

5.5.2. Lisp-Env.

The description of the age of the concept created by the function Make-Concept (88) uses a tilde-expression.

```
(defun MAKE-PERSON (age)                                              88
  #~(defconcept MARY
      (a person
        (age ~age))))

-->  (make-person [number 20])
<--  mary
```

The definition of the age-subject consults the association-list lisp-env, using the key 'g:4440'. In lisp-env, the result of the tilde-description '~age' is stored with the same key.

```
-->  (>> definition handler age of mary)                             89
<--  <form (lisp-rpl '#:g4440 lisp-env)>

-->  (concept-lisp-env *)
<--  ((#:g4440 . <number 20>))
```

6. CONCLUSION.

This chapter described the techniques which have been used for the KRS implementation. The following considerations have played a role in their development.

EFFICIENCE

The KRS concept-graph is by definition a large structure and the mechanisms operating on them are complex. To make a usable implementation, we had to consider very carefully all efficiency issues.

PORTABILITY

KRS was first implemented in ZETALISP, which runs only on Symbolics machines. To make sure that the implementation was easy portable to other LISP dialects (eventually to COMMON-LISP), we could only use the standard LISP features, i.e. those features that occur in some way in every LISP dialect.

FLEXIBILITY AND MAINTAINABILITY

KRS is built in many design and implementation cycles. In order to not having to reimplement everything from scratch at each cycle, the flexibility of the implementation was very important. This was very difficult to achieve because every component of KRS depends on many other components.

Some of the techniques used in the KRS implementation can be viewed in a broader context. It consists of implementation techniques with a function which can be useful in many other application.

The following large parts are described:

- FPPD is a data-dependency mechanism. It is designed to preserve consistency between cached results of computations. It is implemented to operate efficiently on very large and complex data-dependency networks.
- The data-structures must efficiency represent the concept-graph, which is a virtual infinite structure. It represents this graph in a finite way but such that the user can not see where it is shortcut.
- The task-processor is a metaphor for implementing inherently recursive mechanism in an iterative way (by simulating continuation-based programming).
- Inheritance in KRS would be easy to implement, if we did not have the lexical scope mechanism. This mechanism however makes that inheritance influences the scope of descriptions. The KRS solution to determine the scope of an inherited description is based on matching patterns of concepts.
- The KRS environment consists of a distance, a krs-env and a lisp-env. The distance contains data out of which the concept's lexical scope can be computed. The krs-env contains the concept-variable bindings which were instantiated when the concept was created. The lisp-env contains data for the correct interpretation of tilde-expressions.

Chapter Three

KRS USAGE.

INTRODUCTION.

Chapter three illustrates different KRS programming styles. Four extended examples each show a different facet of KRS programming, introduce different programming styles, and argue about the methodology which was followed.

THE ROAD-MAP

The first example shows the use of KRS for straightforward data modeling. There is a direct mapping between the objects in the representation domain and their representation in the concept-graph. In particular, all of the relevant topics in the domain are explicitly represented with a concept, and all relevant characteristics of such an entity are captured by the concept's subjects. It also exemplifies programming with *cliches*.

CLASSIFICATION

The second example demonstrates how data-retrieval mechanisms can be explicitly modeled in KRS. An explicit data-retrieval mechanism provides a manner for a concept to deduce more data about itself in a way different from the standard inheritance mechanism. The example implements a classification-tree and uses the tree in an administrative application to infer by which contract a particular trip can be sponsored.

DEMONS

The third example shows a KRS implementation of demons. Demons are a well-known A.I. technique to model active procedures, i.e. procedures which have to be executed when a certain event occurs. All traditional demon implementations typically trigger demons when a particular variable is set or referenced. We propose a more general triggering mechanisms based on FPPD's dependency network. We use for this the eager-propositions, which automatically recompute their value when they get denied.

A SMALL PRODUCTION SYSTEM

Example four implements a small production system to show how traditional knowledge representation formalisms can be constructed. It is a simple forward chaining rule-system, with a structured rule-base. It also demonstrates how the

consistency maintenance of KRS can be deliberately employed to enhance the efficiency of applications.

1. THE ROAD-MAP.

1.1. Introduction.

The road-map example shows the most common KRS programming style: straightforward data modeling. There is a direct mapping between a static body of knowledge in the representation domain and its representation in the concept-graph. On top of this, reasoning mechanisms may be implemented in LISP.

The example incorporates a knowledge base containing knowledge about road-maps, roads, locations and trajectories between locations. An overview of those is given in figure 1. Tree 1 shows the primitive-objects: roads, locations and trajectories. There are three categories of roads: local-roads, major-roads and motorways. There are two categories of locations: cities and towns. There are two primitive kinds of trajectories: simple-trajectories and combined-trajectories. We will show how these can be specialized in many ways. Trees 2 and 3 show the sets of instances of locations and roads respectively.

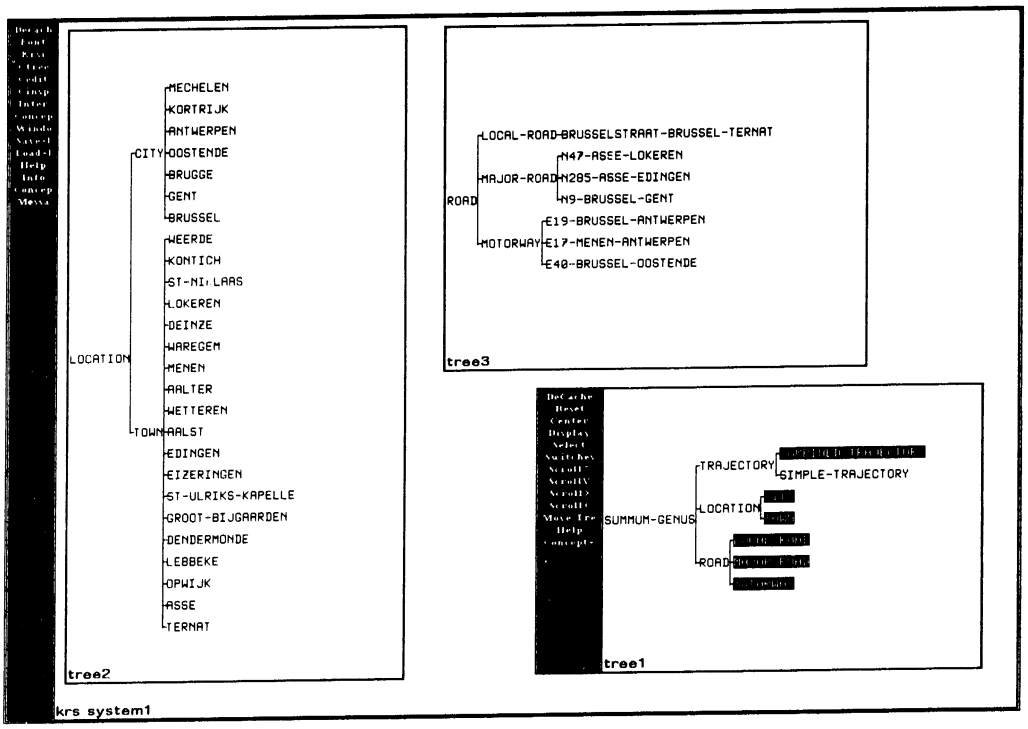


Fig 1: Primitive taxonomies for the road-map application.

We also give some metering information to give some idea of the way the KRS

concept-graph and the FPPD proposition-graph grow, and of the efficiency of the KRS interpreter. Also the effect of caching on the performance of a complex search process is demonstrated.

Finally, also the use of cliches is exemplified.

1.2. The Knowledge Base.

1.2.1. Roads.

A road has an average-speed and a map. The average-speed is used to estimate the duration of following a trajectory. The map is a table containing all locations situated along that road and a kilometer-number indicating where the location is situated.

From its road-map, a road deduces a "from", i.e. the location it starts from, and a "to", i.e. the location where it ends. Given two locations located along a road, the road's distance-subject finds the distance between them by making the difference of the associated kilometer-numbers in the road-map.

```
(defconcept ROAD
  (average-speed (a kms/h))
  (map (a road-map))
  ((from (a subject
           (definition
             [form (caar (>> referent map))])))))
  ((to (a subject
         (definition
           [form (caar (last (>> referent map)))])))))
  ((distance
    (a subject
      (arguments [args ?from ?to])
      (definition [form (a kms
                          (definition
                            [form (abs (- (>> referent (search ?from) map)
                                          (>> referent (search ?to) map)))])))])))))
  (set-type list-of-roads))
```

There are three categories of roads: motorways, major-roads and local-roads. They determine the average-speed of their instances. The average-speed on a motorway for example is hundred kilometers per hour.[13]

[13] This is on a typical Belgian rainy day.

```
(defconcept MOTORWAY
  (a road
    (average-speed [kms/h 100])))

(defconcept MAJOR-ROAD
  (a road
    (average-speed [kms/h 50])))

(defconcept LOCAL-ROAD
  (a road
    (average-speed [kms/h 30])))
```

A list-of-roads is a sequence whose referent is a list of instances of the concept Road.

```
(defconcept LIST-OF-ROADS
  (a sequence
    (element-type road)))
```

A road-map is a table. A table has two columns, the first contains keys, the second values. The keys of a road-map are locations, the values are kilometer-numbers. The elements of a table is a list of all the keys. Given a particular key, its value in a table can be found with the subject "search". A road-map "connects" all its elements.

```
(defconcept ROAD-MAP
  (a table
    (key-type location)
    (value-type km)
    (map-for (a road))
    (connects (>> elements))))


(defconcept TABLE
  (a data-concept
    (meta-interpreter-type table-meta-interpreter)
    (key-type)
    (value-type)
    ((search
       (a subject
         (arguments [args ?key])
         (definition
           [form (cdr (assoc ?key (>> referent)))])))))
    (elements (a (>> set-type key-type)
                 (definition
                   [form (mapcar #'car (>> referent))])))))
```

1.2.2. Locations.

A location is located at a list of roads. Out of this list, the location computes to which other location it is connected. This results in a table of locations and roads, indicating to which other locations this location is connected and via which road.

Out of the "connected-to", the location's "direct-trajectories" are generated. This is a list of simple-trajectories each one going from this location and to one of the

locations it is connected-to, using the road which connects both. A simple-trajectory is a trajectory between two locations via one single road. The direct-trajectories are sorted such that the shorter ones come first.

```
(defconcept LOCATION
  (located-at (a list-of-roads))
  (connected-to
    (a table
     (definition
       [form (mapcan #'(clambda (?road)
                         (mapcan #'(lambda (loc)
                                     (unless (eq loc (>>))
                                       (list (cons loc ?road))))
                                 (>> referent connects map of ?road)))
                     (>> referent located-at))])))
  (direct-trajectories
    (a list-of-trajectories
      (definition
        [form (sort (mapcar #'(lambda (el)
                                #~(a simple-trajectory
                                    (from (>>))
                                    (to ~(car el))
                                    (using ~(cdr el))))
                            (>> referent connected-to))
                    #'(clambda (?x ?y)
                        (< (>> referent distance of ?x)
                           (>> referent distance of ?y))))])))
  (set-type list-of-locations))
```

Locations are categorized into Towns and Cities.

```
(defconcept TOWN
  (a location))
```

```
(defconcept CITY
  (a location))
```

A list-of-locations is a list of instances of the concept Location.

```
(defconcept LIST-OF-LOCATIONS
  (a sequence
    (element-type location)))
```

1.2.3. Trajectories.

Each trajectory has a "from" and a "to". Both are a location. A list-of-trajectories is a list of instances of the concept Trajectory.

```
(defconcept TRAJECTORY
  (from (a location))
  (to (a location))
  (set-type list-of-trajectories))
```

```
(defconcept LIST-OF-TRAJECTORIES
  (a sequence
    (element-type trajectory)))
```

A simple-trajectory is a trajectory via one single road, which is called the "using" of

the trajectory. The distance of the simple-trajectory is the distance between its "from" and its "to" along this road. The trajectory's estimated-duration is computed out of its distance and the average-speed of the road.

```
(defconcept SIMPLE-TRAJECTORY
  (a trajectory
    (meta-interpreter-type simple-trajectory-meta-interpreter)
    (using (a road))
    (distance (>> (distance (>> from) (>> to)) using))
    (estimated-duration
     (a duration
       (definition
         [form (/ (* 60 (>> referent distance))
                  (>> referent average-speed using))]))))))
```

A combined-trajectory uses a list of sub-trajectories. Both its distance and its estimated-duration are computed by adding all the sub-distances or estimated-durations.

```
(defconcept COMBINED-TRAJECTORY
  (a trajectory
    (meta-interpreter-type combined-trajectory-meta-interpreter)
    (using (a list-of-trajectories))
    (distance
     (a kms
       (definition
         [form (apply '+ (mapcar #'(clambda (?trajectory)
                                     (>> referent distance of ?trajectory))
                                 (>> referent using)))])))
    (estimated-duration
     (a duration
       (definition
         [form (apply '+ (mapcar #'(clambda (?trajectory)
                                     (>> referent estimated-duration of ?trajectory))
                                 (>> referent using)))])))))
```

1.2.4. An example.

Within this framework, we encode the road-map which is given in figure 2.

MOTORWAYS

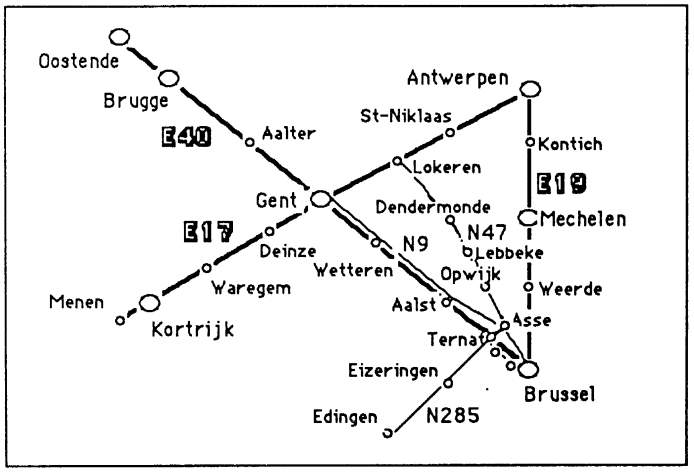


Fig 2: An example road-map.

```
(defconcept E40-BRUSSEL-OOSTENDE
  (a motorway
    (map [road-map (Brussel [km 0])
                   (Ternat [km 8])
                   (Aalst [km 18])
                   (Wetteren [km 43])
                   (Gent [km 54])
                   (Aalter [km 74])
                   (Brugge [km 92])
                   (Oostende [km 116])])))

(defconcept E17-MENEN-ANTWERPEN
  (a motorway
    (map [road-map (Menen [km 0])
                   (Kortrijk [km 7])
                   (Waregem  [km 21])
                   (Deinze [km 34])
                   (Gent [km 47])
                   (Lokeren [km 70])
                   (St-Niklaas [km 83])
                   (Antwerpen [km 102])])))

(defconcept E19-BRUSSEL-ANTWERPEN
  (a motorway
    (map [road-map (Brussel [km 0])
                   (Weerde [km 8])
                   (Mechelen [km 15])
                   (Kontich [km 29])
                   (Antwerpen [km 37])])))
```

MAJOR-ROADS

```
(defconcept N9-BRUSSEL-GENT
  (a major-road
    (map [road-map (Brussel [km 0])
                   (Asse [km 9])
                   (Aalst [km 20])
                   (Gent [Km 43])])))

(defconcept N285-ASSE-EDINGEN
  (a major-road
    (map [road-map (Asse [km 0])
                   (Ternat [km 4])
                   (Eizeringen [km 10])
                   (Edingen [km 30])])))

(defconcept N47-ASSE-LOKEREN
  (a major-road
    (map [road-map (Asse [km 0])
                   (Opwijk [km 5])
                   (Lebbeke [km 10])
                   (Dendermonde [km 14])
                   (Lokeren [km 24])])))
```

LOCAL-ROADS

```
(defconcept BRUSSELSTRAAT-BRUSSEL-TERNAT
  (a local-road
    (map [road-map (Brussel [km 0])
                   (Groot-Bijgaarden [km 1])
                   (St-Ulriks-Kapelle [km 5])
                   (Ternat [km 10])])))
```

CITIES

```
(defconcept BRUSSEL
  (a city
    (located-at [list-of-roads E40-Brussel-Oostende
                               N9-Brussel-Gent
                               Brusselstraat-Brussel-Ternat
                               E19-Brussel-Antwerpen])))

(defconcept GENT
  (a city
    (located-at [list-of-roads E40-Brussel-Oostende
                               N9-Brussel-Gent
                               E17-Menen-Antwerpen])))

(defconcept BRUGGE
  (a city
    (located-at [list-of-roads E40-Brussel-Oostende])))

(defconcept OOSTENDE
  (a city
    (located-at [list-of-roads E40-Brussel-Oostende])))
```

```
(defconcept ANTWERPEN
  (a city
    (located-at [list-of-roads E17-Menen-Antwerpen
                               E19-Brussel-Antwerpen])))

(defconcept KORTRIJK
  (a city
    (located-at [list-of-roads E17-Menen-Antwerpen])))

(defconcept MECHELEN
    (a city
    (located-at [list-of-roads E19-Brussel-Antwerpen])))
```

TOWNS

```
(defconcept TERNAT
  (a town
    (located-at [list-of-roads E40-Brussel-Oostende
                               N285-Asse-Edingen
                               Brusselstraat-Brussel-Ternat])))

(defconcept ASSE
  (a town
    (located-at [list-of-roads N9-Brussel-Gent
                               N285-Asse-Edingen
                               N47-Asse-Lokeren])))

(defconcept OPWIJK
  (a town
    (located-at [list-of-roads N47-Asse-Lokeren])))

(defconcept LEBBEKE
  (a town
    (located-at [list-of-roads N47-Asse-Lokeren])))

(defconcept DENDERMONDE
  (a town
    (located-at [list-of-roads N47-Asse-Lokeren])))

(defconcept GROOT-BIJGAARDEN
  (a town
    (located-at [list-of-roads Brusselstraat-Brussel-Ternat])))

(defconcept ST-ULRIKS-KAPELLE
  (a town
    (located-at [list-of-roads Brusselstraat-Brussel-Ternat])))

(defconcept EIZERINGEN
  (a town
    (located-at [list-of-roads N285-Asse-Edingen])))

(defconcept EDINGEN
  (a town
    (located-at [list-of-roads N285-Asse-Edingen])))
```

```
(defconcept AALST
  (a town
    (located-at [list-of-roads E40-Brussel-Oostende
                               N9-Brussel-Gent])))

(defconcept WETTEREN
  (a town
    (located-at [list-of-roads E40-Brussel-Oostende])))

(defconcept AALTER
  (a town
    (located-at [list-of-roads E40-Brussel-Oostende])))

(defconcept MENEN
  (a town
    (located-at [list-of-roads E17-Menen-Antwerpen])))

(defconcept WAREGEM
  (a town
    (located-at [list-of-roads E17-Menen-Antwerpen])))

(defconcept DEINZE
  (a town
    (located-at [list-of-roads E17-Menen-Antwerpen])))

(defconcept LOKEREN
  (a town
    (located-at [list-of-roads E17-Menen-Antwerpen
                               N47-Asse-Lokeren])))

(defconcept ST-NIKLAAS
  (a town
    (located-at [list-of-roads E17-Menen-Antwerpen])))

(defconcept KONTICH
  (a town
    (located-at [list-of-roads E19-Brussel-Antwerpen])))

(defconcept WEERDE
  (a town
    (located-at [list-of-roads E19-Brussel-Antwerpen])))
```

1.3. The Simulator.

1.3.1. Introduction.

Given a start-location and some additional conditions, the simulator searches a trajectory through the network of roads. The simulator is a search process implemented in LISP, which operates on the knowledge base defined above. The additional conditions it makes use of guide the search: they specify which roads are taken into account, when the search has to consider multiple branches, and how preference between multiple options is determined. The simulator can be thought of as a little

car, which is driving around in the network, duplicating itself when there are multiple possibilities.

1.3.2. Traces.

A trace is a data-structure, containing a qualifier, a list of options and a list of previous-trajectories.

- The previous-trajectories is a list of trajectories already followed by the car to arrive in its current location.
- The options are trajectories all starting from the car's current location. It is among those that the car is now considering how to continue its way.
- The qualifier is a measurement of the degree of success of the trace. For example, if the concern is to find the shortest trajectory between two locations, the qualifier will be computed out of the distances of the individual trajectories. In that case, it is equal to the sum of all the distances of the previous-trajectories, plus the distance of its first option.

The following are macros and functions to create and inspect a trace.

```
(defmacro MAKE-TRACE (qualifier options &optional previous-trajectories)
  `(cons ,qualifier (cons ,options ,previous-trajectories)))

(defmacro CURRENT-OPTION (trace)
  `(caadr ,trace))

(defmacro PREVIOUS-TRAJECTORIES (trace)
  `(cddr ,trace))

(defmacro TRACE-QUALIFIER (trace)
  `(first ,trace))
```

A trace is exhausted if it has no more options. In that case the car has no trajectories left to continue its way. The best trace from a list of traces is the one with the lowest qualifier.

```
(defmacro TRACE-EXHAUSTED (trace)
  `(null (second ,trace)))

(defmacro RULE-OUT-TRACE (trace)
  `(rplaca ,trace :finished))

(defun BEST-TRACE (traces)
  (let ((best (first traces)))
    (dolist (tr (cdr traces))
      (cond
        ((eq (first best) :finished)
         (setq best tr))
        ((eq (first tr) :finished))
        ((< (first tr) (first best))
         (setq best tr))))
    best))
```

Continuing a trace means skipping the first option. The car decides not to take the first option and will thus consider the second now. The qualifier is recomputed such

that the contribution of the former first option is replaced by a contribution of the new first option.

```
(defun CONTINUE-TRACE (trace preference-for)
  (clet ((?option (caadr trace)))
    (rplaca (cdr trace)
            (cdadr trace))
    (if (trace-exhausted trace)
        (rule-out-trace trace)
        (rplaca trace
                (+ (- (car trace)
                      (>> referent ~preference-for of ?option))
                   (>> referent ~preference-for of ~(caadr trace)))))))
```

1.3.3. The function Drive-From.

The simulation is implemented with the function Drive-From. It is an iteration operating over a list of traces. Each trace represents a car which has followed a particular trajectory to get at its current location. Each of the cars is busy evaluating its options, i.e. the trajectories it can take next to continue its way. At each cycle, the iteration choses the most promising trace (the one with the lowest qualifier) and continues this one.

To continue a trace, the function Drive-From distinguishes the following possibilities. If the first option reaches the end-destination, the search can be halted. If the first option reaches a location from where different routes can be tried, a new trace is set up starting in this location. The options of the new trace are all the trajectories it can take from that location, except the ones which do not lead to new trajectories. We assume that a car which arrives in a location via a road X can not take a next trajectory via the same road because these two trajectories can be seen as one.

Drive-From has the following arguments: *End-test* is a function to apply on the first option to decide whether it reaches the destination. *Split-test* is a function to apply on the first option to decide whether a new trace has to start. *Only-drive-on* is a subject-name. It is used to find the list of options a car has when it arrives in a particular location. Those options must already be sorted such that the most preferred comes first. *Preference-for* is a subject-name, indicating which value to take into account to compute the qualifiers.

```
(defun DRIVE-FROM (from end-test split-test only-drive-on preference-for)
  (do* ((all-traces
         (list (make-trace
                (>> referent ~preference-for first ~only-drive-on of ~from)
                (>> referent ~only-drive-on of ~from))))
        (shortest-trace (best-trace all-traces) (best-trace all-traces)))
       (nil)
    (when (eq (first shortest-trace) :finished)
      (error "Drive-From can not find destination."))
    (clet ((?current-option (current-option shortest-trace)))
      (cond
        ((funcall end-test ?current-option)
         (return (reverse (cons ?current-option (previous-trajectories shortest-trace)))))
        ((funcall split-test ?current-option)
         (let ((new-trajectories
                (remove-if #'(clambda (?x)
                               (eq (>> using of ?x)
                                   (>> using of ?current-option)))
                           (>> referent ~only-drive-on to of ?current-option))))
           (push (make-trace (+ (>> referent ~preference-for of ~(first new-trajectories))
                                (first shortest-trace))
                             new-trajectories
                             (cons ?current-option
                                   (previous-trajectories shortest-trace)))
                 all-traces))
         (continue-trace shortest-trace preference-for))
        (t
         (continue-trace shortest-trace preference-for))))))
```

1.3.4. Instantiations of Drive-From.

The function Find-Shortest-Trajectory tries to find the trajectory with the shortest distance between two locations. Its end-condition applies if the "to" of the current-trajectory is equal to the goal-location. The split-test applies if the "to" of a trajectory is located at more than one road. It uses all direct-trajectories of a location, and its preference is determined by the distance of the trajectories.

```
(defun FIND-SHORTEST-TRAJECTORY (from to)
  (drive-from from
              #'(clambda (?tr) (eq (>> to of ?tr) to))
              #'(clambda (?tr) (cdr (>> referent located-at to of ?tr)))
              'direct-trajectories
              'distance))
```

The function Find-Fastest-Trajectory tries to find the trajectory with the lowest estimated-duration between two locations. As in the previous case, its end-condition applies if the "to" of the current-trajectory is equal to the goal-location and its split-test applies if the "to" of a trajectory is located at more than one road. It uses all direct-trajectories of a location, and its preference is determined by the estimated-duration of the trajectories.

To function correctly, the direct-trajectories of a location must now be sorted according to their estimated-duration. Therefore a subject "direct-trajectories-fastest-first" is added.

```
(defun FIND-FASTEST-TRAJECTORY (from to)
  (drive-from from
              #'(clambda (?tr) (eq (>> to of ?tr) to))
              #'(clambda (?tr) (cdr (>> referent located-at to of ?tr)))
              'direct-trajectories-fastest-first
              'estimated-duration))


(defsubject DIRECT-TRAJECTORIES-FASTEST-FIRST of LOCATION
  (a list-of-trajectories
    (definition
      [form (sort (>> referent direct-trajectories)
                  #'(clambda (?x ?y)
                      (< (>> referent estimated-duration of ?x)
                         (>> referent estimated-duration of ?y))))]))))
```

The function Find-Shortest-Trajectory-To-Motorway also searches for a trajectory with minimal distance, but it stops as soon as it finds a location located at a motorway. We add a subject "located-at-motorways" to the concept Location, such that this can be cached for each location.

```
(defun FIND-SHORTEST-TRAJECTORY-TO-MOTORWAY (from)
  (drive-from from
              #'(clambda (?tr) (>> referent located-at-motorways to of ?tr))
              #'(clambda (?tr) (cdr (>> referent located-at to of ?tr)))
              'direct-trajectories
              'distance))


(defsubject LOCATED-AT-MOTORWAYS of LOCATION
  (a list-of-roads
    (definition
      [form (remove-if-not  #'(lambda (road)
                                (subtype-p road (>> of motorway)))
                            (>> referent located-at))])))
```

The function Find-Shortest-Motorway-Trajectory finds the shortest trajectory between two locations while only using motorway-trajectories. A subject "motorway-trajectories" is added to Location filtering out those direct-trajectories which use a motorway.

```
(defun FIND-SHORTEST-MOTORWAY-TRAJECTORY (from to)
  (drive-from from
              #'(clambda (?tr) (eq (>> to of ?tr) to))
              #'(clambda (?tr) (cdr (>> referent located-at-motorways to of ?tr)))
              'motorway-trajectories
              'distance))


(defsubject MOTORWAY-TRAJECTORIES of LOCATION
  (a list-of-trajectories
    (definition
      [form (remove-if-not #'(clambda (?x)
                               (subtype-p (>> using of ?x) (>> of motorway)))
                           (>> referent direct-trajectories))])))
```

1.3.5. A Trace.

We illustrate the functionality of Drive-From by tracing an application of Find-Shortest-Motorway-Trajectory between the locations Aalst and St-Niklaas. At each step in the iteration, the binding of the variable "all-traces" is printed.

Recall that at each cycle, all-traces is bound to the list of active traces, that the first element of each trace is a qualifier, that the trace with the lowest qualifier is continued, that the second element of a trace is the list of current **options**, and that the remaining elements of a trace (if there are such), are the trajectories already followed to arrive in its current location. When presented as a best-first search, the traces are the leaves of the current search-tree. The qualifier determines the most successful leave-node so far. The options are used for the construction of a new subtree under this node. And the previous trajectories is the path followed so far to arrive in this node.

```
--> (find-shortest-motorway-trajectory (>> of aalst) (>> of st-niklaas))
```

In the first cycle, there is one trace. Its current-trajectories are all the trajectories starting from Aalst ordered by their distances. The distance of the first of those is 10 kilometers.

```
((10 (<TRAJECTORY <AALST> <TERNAT> VIA <E40-BRUSSEL-OOSTENDE>>
      <TRAJECTORY <AALST> <BRUSSEL> VIA <E40-BRUSSEL-OOSTENDE>>
      <TRAJECTORY <AALST> <WETTEREN> VIA <E40-BRUSSEL-OOSTENDE>>
      <TRAJECTORY <AALST> <GENT> VIA <E40-BRUSSEL-OOSTENDE>>
      <TRAJECTORY <AALST> <AALTER> VIA <E40-BRUSSEL-OOSTENDE>>
      <TRAJECTORY <AALST> <BRUGGE> VIA <E40-BRUSSEL-OOSTENDE>>
      <TRAJECTORY <AALST> <OOSTENDE> VIA <E40-BRUSSEL-OOSTENDE>>)))
```

The destination of this first trajectory, Ternat, has no connections to other motorways so it is skipped. The new qualifier is computed by subtracting the distance of the old first option from the old qualifier and adding the distance of the new first option.

```
((18 (<TRAJECTORY <AALST> <BRUSSEL> VIA <E40-BRUSSEL-OOSTENDE>>
      <TRAJECTORY <AALST> <WETTEREN> VIA <E40-BRUSSEL-OOSTENDE>>
      <TRAJECTORY <AALST> <GENT> VIA <E40-BRUSSEL-OOSTENDE>>
      <TRAJECTORY <AALST> <AALTER> VIA <E40-BRUSSEL-OOSTENDE>>
      <TRAJECTORY <AALST> <BRUGGE> VIA <E40-BRUSSEL-OOSTENDE>>
      <TRAJECTORY <AALST> <OOSTENDE> VIA <E40-BRUSSEL-OOSTENDE>>)))
```

The first trajectory in the second cycle has as destination Brussel, from which place other motorways can be taken. Hence, a second trace is instantiated.

```
((26 (<TRAJECTORY <BRUSSEL> <WEERDE> VIA <E19-BRUSSEL-ANTWERPEN>>
      <TRAJECTORY <BRUSSEL> <MECHELEN> VIA <E19-BRUSSEL-ANTWERPEN>>
      <TRAJECTORY <BRUSSEL> <KONTICH> VIA <E19-BRUSSEL-ANTWERPEN>>
      <TRAJECTORY <BRUSSEL> <ANTWERPEN> VIA <E19-BRUSSEL-ANTWERPEN>>)
     <TRAJECTORY <AALST> <BRUSSEL> VIA <E40-BRUSSEL-OOSTENDE>>)
 (25 (<TRAJECTORY <AALST> <WETTEREN> VIA <E40-BRUSSEL-OOSTENDE>>
      <TRAJECTORY <AALST> <GENT> VIA <E40-BRUSSEL-OOSTENDE>>
      <TRAJECTORY <AALST> <AALTER> VIA <E40-BRUSSEL-OOSTENDE>>
      <TRAJECTORY <AALST> <BRUGGE> VIA <E40-BRUSSEL-OOSTENDE>>
      <TRAJECTORY <AALST> <OOSTENDE> VIA <E40-BRUSSEL-OOSTENDE>>)))
```

This results in two traces. One trace is the continuation of the previous trace which continues to try all motorway-trajectories starting from Aalst. The other trace tries all trajectories starting from Brussels. This trace has one previous-trajectory, i.e. the trajectory Aalst-Brussels. The qualifier of this trace, 26, is the distance of driving from Aalst to Weerde, which is the sum of the distances of the trajectories Aalst-Brussel and Brussel-Weerde. From now on, in each cycle the trace with the shortest qualifier will be continued. In this case thus, the next trajectory starting from Aalst will be explored. Its destination Wetteren has no other connections so this trajectory is skipped.

```
((26 (<TRAJECTORY <BRUSSEL> <WEERDE> VIA <E19-BRUSSEL-ANTWERPEN>>
      <TRAJECTORY <BRUSSEL> <MECHELEN> VIA <E19-BRUSSEL-ANTWERPEN>>
      <TRAJECTORY <BRUSSEL> <KONTICH> VIA <E19-BRUSSEL-ANTWERPEN>>
      <TRAJECTORY <BRUSSEL> <ANTWERPEN> VIA <E19-BRUSSEL-ANTWERPEN>>)
  <TRAJECTORY <AALST> <BRUSSEL> VIA <E40-BRUSSEL-OOSTENDE>>)
 (36 (<TRAJECTORY <AALST> <GENT> VIA <E40-BRUSSEL-OOSTENDE>>
      <TRAJECTORY <AALST> <AALTER> VIA <E40-BRUSSEL-OOSTENDE>>
      <TRAJECTORY <AALST> <BRUGGE> VIA <E40-BRUSSEL-OOSTENDE>>
      <TRAJECTORY <AALST> <OOSTENDE> VIA <E40-BRUSSEL-OOSTENDE>>)))
```

Skipping Brussel-Weerde ...

```
((33 (<TRAJECTORY <BRUSSEL> <MECHELEN> VIA <E19-BRUSSEL-ANTWERPEN>>
      <TRAJECTORY <BRUSSEL> <KONTICH> VIA <E19-BRUSSEL-ANTWERPEN>>
      <TRAJECTORY <BRUSSEL> <ANTWERPEN> VIA <E19-BRUSSEL-ANTWERPEN>>)
  <TRAJECTORY <AALST> <BRUSSEL> VIA <E40-BRUSSEL-OOSTENDE>>)
 (36 (<TRAJECTORY <AALST> <GENT> VIA <E40-BRUSSEL-OOSTENDE>>
      <TRAJECTORY <AALST> <AALTER> VIA <E40-BRUSSEL-OOSTENDE>>
      <TRAJECTORY <AALST> <BRUGGE> VIA <E40-BRUSSEL-OOSTENDE>>
      <TRAJECTORY <AALST> <OOSTENDE> VIA <E40-BRUSSEL-OOSTENDE>>)))
```

Skipping Brussel-Mechelen ...

```
((47 (<TRAJECTORY <BRUSSEL> <KONTICH> VIA <E19-BRUSSEL-ANTWERPEN>>
      <TRAJECTORY <BRUSSEL> <ANTWERPEN> VIA <E19-BRUSSEL-ANTWERPEN>>)
  <TRAJECTORY <AALST> <BRUSSEL> VIA <E40-BRUSSEL-OOSTENDE>>)
 (36 (<TRAJECTORY <AALST> <GENT> VIA <E40-BRUSSEL-OOSTENDE>>
      <TRAJECTORY <AALST> <AALTER> VIA <E40-BRUSSEL-OOSTENDE>>
      <TRAJECTORY <AALST> <BRUGGE> VIA <E40-BRUSSEL-OOSTENDE>>
      <TRAJECTORY <AALST> <OOSTENDE> VIA <E40-BRUSSEL-OOSTENDE>>)))
```

The next shortest trajectory is the trajectory Aalst-Gent. Since Gent has again connections to other motorways, a third trace must be set up.

```
((49 (<TRAJECTORY <GENT> <DEINZE> VIA <E17-MENEN-ANTWERPEN>>
      <TRAJECTORY <GENT> <LOKEREN> VIA <E17-MENEN-ANTWERPEN>>
      <TRAJECTORY <GENT> <WAREGEM> VIA <E17-MENEN-ANTWERPEN>>
      <TRAJECTORY <GENT> <ST-NIKLAAS> VIA <E17-MENEN-ANTWERPEN>>
      <TRAJECTORY <GENT> <KORTRIJK> VIA <E17-MENEN-ANTWERPEN>>
      <TRAJECTORY <GENT> <MENEN> VIA <E17-MENEN-ANTWERPEN>>
      <TRAJECTORY <GENT> <ANTWERPEN> VIA <E17-MENEN-ANTWERPEN>>)
  <TRAJECTORY <AALST> <GENT> VIA <E40-BRUSSEL-OOSTENDE>>)
 (47 (<TRAJECTORY <BRUSSEL> <KONTICH> VIA <E19-BRUSSEL-ANTWERPEN>>
      <TRAJECTORY <BRUSSEL> <ANTWERPEN> VIA <E19-BRUSSEL-ANTWERPEN>>)
  <TRAJECTORY <AALST> <BRUSSEL> VIA <E40-BRUSSEL-OOSTENDE>>)
 (56 (<TRAJECTORY <AALST> <AALTER> VIA <E40-BRUSSEL-OOSTENDE>>
      <TRAJECTORY <AALST> <BRUGGE> VIA <E40-BRUSSEL-OOSTENDE>>
      <TRAJECTORY <AALST> <OOSTENDE> VIA <E40-BRUSSEL-OOSTENDE>>)))
```

There are now three traces, one still exploring motorway-trajectories starting from Aalst, one exploring motorway-trajectories starting from Brussels, and one exploring motorway-trajectories starting from Gent. The shortest trajectory here is the one going to Kontich. Since there are no possible other directions to take in Kontich, this option is skipped.

```
((49 (<TRAJECTORY <GENT> <DEINZE> VIA <E17-MENEN-ANTWERPEN>>
      <TRAJECTORY <GENT> <LOKEREN> VIA <E17-MENEN-ANTWERPEN>>
      <TRAJECTORY <GENT> <WAREGEM> VIA <E17-MENEN-ANTWERPEN>>
      <TRAJECTORY <GENT> <ST-NIKLAAS> VIA <E17-MENEN-ANTWERPEN>>
      <TRAJECTORY <GENT> <KORTRIJK> VIA <E17-MENEN-ANTWERPEN>>
      <TRAJECTORY <GENT> <MENEN> VIA <E17-MENEN-ANTWERPEN>>
      <TRAJECTORY <GENT> <ANTWERPEN> VIA <E17-MENEN-ANTWERPEN>>)
  <TRAJECTORY <AALST> <GENT> VIA <E40-BRUSSEL-OOSTENDE>>)
 (55 (<TRAJECTORY <BRUSSEL> <ANTWERPEN> VIA <E19-BRUSSEL-ANTWERPEN>>)
  <TRAJECTORY <AALST> <BRUSSEL> VIA <E40-BRUSSEL-OOSTENDE>>)
 (56 (<TRAJECTORY <AALST> <AALTER> VIA <E40-BRUSSEL-OOSTENDE>>
      <TRAJECTORY <AALST> <BRUGGE> VIA <E40-BRUSSEL-OOSTENDE>>
      <TRAJECTORY <AALST> <OOSTENDE> VIA <E40-BRUSSEL-OOSTENDE>>)))
```

Skipping Gent-Deinze ...

```
((59 (<TRAJECTORY <GENT> <LOKEREN> VIA <E17-MENEN-ANTWERPEN>>
      <TRAJECTORY <GENT> <WAREGEM> VIA <E17-MENEN-ANTWERPEN>>
      <TRAJECTORY <GENT> <ST-NIKLAAS> VIA <E17-MENEN-ANTWERPEN>>
      <TRAJECTORY <GENT> <KORTRIJK> VIA <E17-MENEN-ANTWERPEN>>
      <TRAJECTORY <GENT> <MENEN> VIA <E17-MENEN-ANTWERPEN>>
      <TRAJECTORY <GENT> <ANTWERPEN> VIA <E17-MENEN-ANTWERPEN>>)
  <TRAJECTORY <AALST> <GENT> VIA <E40-BRUSSEL-OOSTENDE>>)
 (55 (<TRAJECTORY <BRUSSEL> <ANTWERPEN> VIA <E19-BRUSSEL-ANTWERPEN>>)
  <TRAJECTORY <AALST> <BRUSSEL> VIA <E40-BRUSSEL-OOSTENDE>>)
 (56 (<TRAJECTORY <AALST> <AALTER> VIA <E40-BRUSSEL-OOSTENDE>>
      <TRAJECTORY <AALST> <BRUGGE> VIA <E40-BRUSSEL-OOSTENDE>>
      <TRAJECTORY <AALST> <OOSTENDE> VIA <E40-BRUSSEL-OOSTENDE>>)))
```

The next trajectory to explore goes to Antwerpen. From there, a new trace can be set up trying new trajectories starting in Antwerpen.

```
((74 (<TRAJECTORY <ANTWERPEN> <ST-NIKLAAS> VIA <E17-MENEN-ANTWERPEN>>
      <TRAJECTORY <ANTWERPEN> <LOKEREN> VIA <E17-MENEN-ANTWERPEN>>
      <TRAJECTORY <ANTWERPEN> <GENT> VIA <E17-MENEN-ANTWERPEN>>
      <TRAJECTORY <ANTWERPEN> <DEINZE> VIA <E17-MENEN-ANTWERPEN>>
      <TRAJECTORY <ANTWERPEN> <WAREGEM> VIA <E17-MENEN-ANTWERPEN>>
      <TRAJECTORY <ANTWERPEN> <KORTRIJK> VIA <E17-MENEN-ANTWERPEN>>
      <TRAJECTORY <ANTWERPEN> <MENEN> VIA <E17-MENEN-ANTWERPEN>>)
  <TRAJECTORY <BRUSSEL> <ANTWERPEN> VIA <E19-BRUSSEL-ANTWERPEN>>
  <TRAJECTORY <AALST> <BRUSSEL> VIA <E40-BRUSSEL-OOSTENDE>>)
 (59 (<TRAJECTORY <GENT> <LOKEREN> VIA <E17-MENEN-ANTWERPEN>>
      <TRAJECTORY <GENT> <WAREGEM> VIA <E17-MENEN-ANTWERPEN>>
      <TRAJECTORY <GENT> <ST-NIKLAAS> VIA <E17-MENEN-ANTWERPEN>>
      <TRAJECTORY <GENT> <KORTRIJK> VIA <E17-MENEN-ANTWERPEN>>
      <TRAJECTORY <GENT> <MENEN> VIA <E17-MENEN-ANTWERPEN>>
      <TRAJECTORY <GENT> <ANTWERPEN> VIA <E17-MENEN-ANTWERPEN>>)
  <TRAJECTORY <AALST> <GENT> VIA <E40-BRUSSEL-OOSTENDE>>)
 (:FINISHED NIL <TRAJECTORY <AALST> <BRUSSEL> VIA <E40-BRUSSEL-OOSTENDE>>)
 (56 (<TRAJECTORY <AALST> <AALTER> VIA <E40-BRUSSEL-OOSTENDE>>
      <TRAJECTORY <AALST> <BRUGGE> VIA <E40-BRUSSEL-OOSTENDE>>
      <TRAJECTORY <AALST> <OOSTENDE> VIA <E40-BRUSSEL-OOSTENDE>>)))
```

There are now four traces, one of which is ruled out because there remain no more options. In this cycle, the trajectory Aalst-Aalter must be skipped.

```
((74 (<TRAJECTORY <ANTWERPEN> <ST-NIKLAAS> VIA <E17-MENEN-ANTWERPEN>>
      <TRAJECTORY <ANTWERPEN> <LOKEREN> VIA <E17-MENEN-ANTWERPEN>>
      <TRAJECTORY <ANTWERPEN> <GENT> VIA <E17-MENEN-ANTWERPEN>>
      <TRAJECTORY <ANTWERPEN> <DEINZE> VIA <E17-MENEN-ANTWERPEN>>
      <TRAJECTORY <ANTWERPEN> <WAREGEM> VIA <E17-MENEN-ANTWERPEN>>
      <TRAJECTORY <ANTWERPEN> <KORTRIJK> VIA <E17-MENEN-ANTWERPEN>>
      <TRAJECTORY <ANTWERPEN> <MENEN> VIA <E17-MENEN-ANTWERPEN>>)
  <TRAJECTORY <BRUSSEL> <ANTWERPEN> VIA <E19-BRUSSEL-ANTWERPEN>>
  <TRAJECTORY <AALST> <BRUSSEL> VIA <E40-BRUSSEL-OOSTENDE>>)
 (59 (<TRAJECTORY <GENT> <LOKEREN> VIA <E17-MENEN-ANTWERPEN>>
      <TRAJECTORY <GENT> <WAREGEM> VIA <E17-MENEN-ANTWERPEN>>
      <TRAJECTORY <GENT> <ST-NIKLAAS> VIA <E17-MENEN-ANTWERPEN>>
      <TRAJECTORY <GENT> <KORTRIJK> VIA <E17-MENEN-ANTWERPEN>>
      <TRAJECTORY <GENT> <MENEN> VIA <E17-MENEN-ANTWERPEN>>
      <TRAJECTORY <GENT> <ANTWERPEN> VIA <E17-MENEN-ANTWERPEN>>)
  <TRAJECTORY <AALST> <GENT> VIA <E40-BRUSSEL-OOSTENDE>>)
 (:FINISHED NIL <TRAJECTORY <AALST> <BRUSSEL> VIA <E40-BRUSSEL-OOSTENDE>>)
 (74 (<TRAJECTORY <AALST> <BRUGGE> VIA <E40-BRUSSEL-OOSTENDE>>
      <TRAJECTORY <AALST> <OOSTENDE> VIA <E40-BRUSSEL-OOSTENDE>>)))
```

Skipping Gent-Lokeren ...

```
((74 (<TRAJECTORY <ANTWERPEN> <ST-NIKLAAS> VIA <E17-MENEN-ANTWERPEN>>
      <TRAJECTORY <ANTWERPEN> <LOKEREN> VIA <E17-MENEN-ANTWERPEN>>
      <TRAJECTORY <ANTWERPEN> <GENT> VIA <E17-MENEN-ANTWERPEN>>
      <TRAJECTORY <ANTWERPEN> <DEINZE> VIA <E17-MENEN-ANTWERPEN>>
      <TRAJECTORY <ANTWERPEN> <WAREGEM> VIA <E17-MENEN-ANTWERPEN>>
      <TRAJECTORY <ANTWERPEN> <KORTRIJK> VIA <E17-MENEN-ANTWERPEN>>
      <TRAJECTORY <ANTWERPEN> <MENEN> VIA <E17-MENEN-ANTWERPEN>>)
  <TRAJECTORY <BRUSSEL> <ANTWERPEN> VIA <E19-BRUSSEL-ANTWERPEN>>
  <TRAJECTORY <AALST> <BRUSSEL> VIA <E40-BRUSSEL-OOSTENDE>>)
 (62 (<TRAJECTORY <GENT> <WAREGEM> VIA <E17-MENEN-ANTWERPEN>>
      <TRAJECTORY <GENT> <ST-NIKLAAS> VIA <E17-MENEN-ANTWERPEN>>
      <TRAJECTORY <GENT> <KORTRIJK> VIA <E17-MENEN-ANTWERPEN>>
      <TRAJECTORY <GENT> <MENEN> VIA <E17-MENEN-ANTWERPEN>>
      <TRAJECTORY <GENT> <ANTWERPEN> VIA <E17-MENEN-ANTWERPEN>>)
  <TRAJECTORY <AALST> <GENT> VIA <E40-BRUSSEL-OOSTENDE>>)
 (:FINISHED NIL <TRAJECTORY <AALST> <BRUSSEL> VIA <E40-BRUSSEL-OOSTENDE>>)
 (74 (<TRAJECTORY <AALST> <BRUGGE> VIA <E40-BRUSSEL-OOSTENDE>>
      <TRAJECTORY <AALST> <OOSTENDE> VIA <E40-BRUSSEL-OOSTENDE>>)))
```

Skipping Gent-Waregem ...

```
((74 (<TRAJECTORY <ANTWERPEN> <ST-NIKLAAS> VIA <E17-MENEN-ANTWERPEN>>
      <TRAJECTORY <ANTWERPEN> <LOKEREN> VIA <E17-MENEN-ANTWERPEN>>
      <TRAJECTORY <ANTWERPEN> <GENT> VIA <E17-MENEN-ANTWERPEN>>
      <TRAJECTORY <ANTWERPEN> <DEINZE> VIA <E17-MENEN-ANTWERPEN>>
      <TRAJECTORY <ANTWERPEN> <WAREGEM> VIA <E17-MENEN-ANTWERPEN>>
      <TRAJECTORY <ANTWERPEN> <KORTRIJK> VIA <E17-MENEN-ANTWERPEN>>
      <TRAJECTORY <ANTWERPEN> <MENEN> VIA <E17-MENEN-ANTWERPEN>>)
  <TRAJECTORY <BRUSSEL> <ANTWERPEN> VIA <E19-BRUSSEL-ANTWERPEN>>
  <TRAJECTORY <AALST> <BRUSSEL> VIA <E40-BRUSSEL-OOSTENDE>>)
 (72 (<TRAJECTORY <GENT> <ST-NIKLAAS> VIA <E17-MENEN-ANTWERPEN>>
      <TRAJECTORY <GENT> <KORTRIJK> VIA <E17-MENEN-ANTWERPEN>>
      <TRAJECTORY <GENT> <MENEN> VIA <E17-MENEN-ANTWERPEN>>
      <TRAJECTORY <GENT> <ANTWERPEN> VIA <E17-MENEN-ANTWERPEN>>)
  <TRAJECTORY <AALST> <GENT> VIA <E40-BRUSSEL-OOSTENDE>>)
 (:FINISHED NIL <TRAJECTORY <AALST> <BRUSSEL> VIA <E40-BRUSSEL-OOSTENDE>>)
 (74 (<TRAJECTORY <AALST> <BRUGGE> VIA <E40-BRUSSEL-OOSTENDE>>
      <TRAJECTORY <AALST> <OOSTENDE> VIA <E40-BRUSSEL-OOSTENDE>>)))
```

The best trace we have now arrives in St-Niklaas, which is our end destination. So a list of two trajectories is returned, the first going from Aalst to Gent, the second going from Gent to St-Niklaas.

```
(<TRAJECTORY <AALST> <GENT> VIA <E40-BRUSSEL-OOSTENDE>>
 <TRAJECTORY <GENT> <ST-NIKLAAS> VIA <E17-MENEN-ANTWERPEN>>)
```

1.4. A larger example.

1.4.1. The concepts Shortest-Trajectory and Fastest-Trajectorys.

The concepts Shortest-Trajectory and Fastest-Trajectory are concepts using the functions Find-Shortest-Trajectory and Find-Fastest-Trajectory respectively.

```
(defconcept SHORTEST-TRAJECTORY
  (a combined-trajectory
    (from (a location))
    (to (a location))
    (using (a list-of-trajectories
             (definition [form (find-shortest-trajectory (>> from) (>> to))])))))

(defconcept FASTEST-TRAJECTORY
  (a combined-trajectory
    (from (a location))
    (to (a location))
    (using (a list-of-trajectories
             (definition [form (find-fastest-trajectory (>> from) (>> to))])))))
```

We take four example-trajectories, two being the fastest and the shortest trajectories between Asse and Antwerpen and two being the fastest and the shortest trajectories between Opwijk and Antwerpen.

```
(defconcept SHORTEST-TRAJECTORY-ASSE-ANTWERPEN
  (a shortest-trajectory
    (from asse)
    (to antwerpen)))

(defconcept FASTEST-TRAJECTORY-ASSE-ANTWERPEN
  (a fastest-trajectory
    (from asse)
    (to antwerpen)))

(defconcept SHORTEST-TRAJECTORY-OPWIJK-ANTWERPEN
  (a shortest-trajectory
    (from opwijk)
    (to antwerpen)))

(defconcept FASTEST-TRAJECTORY-OPWIJK-ANTWERPEN
  (a fastest-trajectory
    (from opwijk)
    (to antwerpen)))
```

The first test shows that the shortest trajectory between Asse and Antwerpen is via Brussel. In the second test we see that it is faster to go from Asse to Ternat first and to take the motorway to Brussel there. (The function Inspect-Trajectory is an auxiliary function to print the interesting information about a given trajectory).

```
--> (inspect-trajectory (>> of shortest-trajectory-asse-antwerpen))

Trajectory: <SHORTEST-TRAJECTORY-ASSE-ANTWERPEN>;
distance: <KMS 46>;
estimated-duration: <DURATION 33 '>;
via: (<TRAJECTORY <ASSE> <BRUSSEL> VIA <N9-BRUSSEL-GENT>>
      <TRAJECTORY <BRUSSEL> <ANTWERPEN> VIA <E19-BRUSSEL-ANTWERPEN>>)

--> (inspect-trajectory (>> of fastest-trajectory-asse-antwerpen))

Trajectory: <FASTEST-TRAJECTORY-ASSE-ANTWERPEN>;
distance: <KMS 49>;
estimated-duration: <DURATION 31 '>;
via: (<TRAJECTORY <ASSE> <TERNAT> VIA <N285-ASSE-EDINGEN>>
      <TRAJECTORY <TERNAT> <BRUSSEL> VIA <E40-BRUSSEL-OOSTENDE>>
      <TRAJECTORY <BRUSSEL> <ANTWERPEN> VIA <E19-BRUSSEL-ANTWERPEN>>)
```

The following test shows that the shortest trajectory from Opwijk to Antwerpen goes via Lokeren. The fastest trajectory however is to go first to Asse and to take from there the trajectory Asse-Antwerpen we had above.

```
--> (inspect-trajectory (>> of shortest-trajectory-opwijk-antwerpen))

Trajectory: <SHORTEST-TRAJECTORY-OPWIJK-ANTWERPEN>;
distance: <KMS 51>;
estimated-duration: <DURATION 42 '>;
via: (<TRAJECTORY <OPWIJK> <LOKEREN> VIA <N47-ASSE-LOKEREN>>
      <TRAJECTORY <LOKEREN> <ANTWERPEN> VIA <E17-MENEN-ANTWERPEN>>)

--> (inspect-trajectory (>> of fastest-trajectory-opwijk-antwerpen))

Trajectory: <FASTEST-TRAJECTORY-OPWIJK-ANTWERPEN>;
distance: <KMS 54>;
estimated-duration: <DURATION 37 '>;
via: (<TRAJECTORY <OPWIJK> <ASSE> VIA <N47-ASSE-LOKEREN>>
      <TRAJECTORY <ASSE> <TERNAT> VIA <N285-ASSE-EDINGEN>>
      <TRAJECTORY <TERNAT> <BRUSSEL> VIA <E40-BRUSSEL-OOSTENDE>>
      <TRAJECTORY <BRUSSEL> <ANTWERPEN> VIA <E19-BRUSSEL-ANTWERPEN>>)
```

Next we show the effect of changing the average-speed of the motorway E19-Brussel-Antwerpen on the trajectories shown above.

```
(defsubject AVERAGE-SPEED of E19-BRUSSEL-ANTWERPEN
  [kms/h 50])
```

Obviously the distances of trajectories is not affected by this modification, so that the shortest trajectory remains unchanged (everything remains even cached). The fastest trajectory between Aalst and Antwerpen however goes now no longer via Brussels, but via Lokeren.

```
Trajectory: <SHORTEST-TRAJECTORY-ASSE-ANTWERPEN>;
distance: <KMS 46>;
estimated-duration: <DURATION 55 '>;
via: (<TRAJECTORY <ASSE> <BRUSSEL> VIA <N9-BRUSSEL-GENT>>
      <TRAJECTORY <BRUSSEL> <ANTWERPEN> VIA <E19-BRUSSEL-ANTWERPEN>>)

Trajectory: <FASTEST-TRAJECTORY-ASSE-ANTWERPEN>;
distance: <KMS 56>;
estimated-duration: <DURATION 48 '>;
via: (<TRAJECTORY <ASSE> <LOKEREN> VIA <N47-ASSE-LOKEREN>>
      <TRAJECTORY <LOKEREN> <ANTWERPEN> VIA <E17-MENEN-ANTWERPEN>>)
```

Indeed, we see that the estimated-duration of the previous fastest trajectory is now fifty-four minutes, where it was thirty-one minutes before.

```
(inspect-trajectory
  (a combined-trajectory
    (from asse)
    (to antwerpen)
    (using
      [list-of-trajectories
        (a simple-trajectory
          (from asse) (to ternat) (using n285-asse-edingen))
        (a simple-trajectory
          (from ternat) (to brussel) (using E40-brussel-oostende))
        (a simple-trajectory
          (from brussel) (to antwerpen) (using e19-brussel-antwerpen))])))

Trajectory: <COMBINED-TRAJECTORY <ASSE> <ANTWERPEN>>;
distance: <KMS 49>;
estimated-duration: <DURATION 54 '>;
via: (<TRAJECTORY <ASSE> <TERNAT> VIA <N285-ASSE-EDINGEN>>
      <TRAJECTORY <TERNAT> <BRUSSEL> VIA <E40-BRUSSEL-OOSTENDE>>
      <TRAJECTORY <BRUSSEL> <ANTWERPEN> VIA <E19-BRUSSEL-ANTWERPEN>>)
```

1.4.2. Some Metering Data.

This section provides information about the size of the concept-graph and proposition-graph created by this example. We start the experiments with the following metering-data. There are already 673 concepts created, 3951 FPPD-propositions created, 20554 proposition-value's requested, 1489 proposition-value's computed, and 774 propositions denied.

```
Total number of concepts: 673
Total number of propositions: 3951
Total number of requested values: 20554
Total number of computed values: 1489
Total number of denied values: 774
```

The numbers below show the result of the first computation of the shortest-trajectory between Asse and Antwerpen, i.e. there are no cached values yet in the road-map knowledge-base. The numbers are to be interpreted as follows: this computation took 69 seconds, instantiated 494 new concepts, used 17638 tasks, instantiated 2604 FPPD-propositions, asked 17921 proposition values of which 1107 were not cached, and did deny zero propositions. Notice that the duration number gives only a

indication of the time it takes to reply to the request, since it is measured in real-time while there are multiple processes running.

```
=======================================================================
expression: (INSPECT-TRAJECTORY (>> OF SHORTEST-TRAJECTORY-ASSE-ANTWERPEN))

Trajectory: <SHORTEST-TRAJECTORY-ASSE-ANTWERPEN>;
distance: <KMS 46>;
estimated-duration: <DURATION 33 '>;
via: (<TRAJECTORY <ASSE> <BRUSSEL> VIA <N9-BRUSSEL-GENT>>
      <TRAJECTORY <BRUSSEL> <ANTWERPEN> VIA <E19-BRUSSEL-ANTWERPEN>>)

time: 69.6868 secs. 494 new concepts. 17638 tasks
fppd: 2604 new propositions; 17921 requests; 1107 computed; 0 denied
=======================================================================
```

Afterwards, the fastest trajectory between the same locations is computed. This time, a lot of the needed information is already cached, such that the performance is much better.

```
=======================================================================
expression: (INSPECT-TRAJECTORY (>> OF FASTEST-TRAJECTORY-ASSE-ANTWERPEN))

Trajectory: <FASTEST-TRAJECTORY-ASSE-ANTWERPEN>;
distance: <KMS 49>;
estimated-duration: <DURATION 31 '>;
via: (<TRAJECTORY <ASSE> <TERNAT> VIA <N285-ASSE-EDINGEN>>
      <TRAJECTORY <TERNAT> <BRUSSEL> VIA <E40-BRUSSEL-OOSTENDE>>
      <TRAJECTORY <BRUSSEL> <ANTWERPEN> VIA <E19-BRUSSEL-ANTWERPEN>>)

time: 4.774821 secs. 177 new concepts. 4109 tasks
fppd: 905 new propositions; 4475 requests; 378 computed; 0 denied
=======================================================================
```

Similarly, to compute the shortest and fastest trajectories between Opwijk and Antwerpen, we still benefit from the values cached before.

```
=======================================================================
expression: (INSPECT-TRAJECTORY (>> OF SHORTEST-TRAJECTORY-OPWIJK-ANTWERPEN))

Trajectory: <SHORTEST-TRAJECTORY-OPWIJK-ANTWERPEN>;
distance: <KMS 51>;
estimated-duration: <DURATION 42 '>;
via: (<TRAJECTORY <OPWIJK> <LOKEREN> VIA <N47-ASSE-LOKEREN>>
      <TRAJECTORY <LOKEREN> <ANTWERPEN> VIA <E17-MENEN-ANTWERPEN>>)

time: 13.681679 secs. 61 new concepts. 9124 tasks
fppd: 326 new propositions; 9510 requests; 151 computed; 0 denied
=======================================================================
```

```
=====================================================================
expression: (INSPECT-TRAJECTORY (>> OF FASTEST-TRAJECTORY-OPWIJK-ANTWERPEN))

Trajectory: <FASTEST-TRAJECTORY-OPWIJK-ANTWERPEN>;
distance: <KMS 54>;
estimated-duration: <DURATION 37 '>;
via: (<TRAJECTORY <OPWIJK> <ASSE> VIA <N47-ASSE-LOKEREN>>
      <TRAJECTORY <ASSE> <TERNAT> VIA <N285-ASSE-EDINGEN>>
      <TRAJECTORY <TERNAT> <BRUSSEL> VIA <E40-BRUSSEL-OOSTENDE>>
      <TRAJECTORY <BRUSSEL> <ANTWERPEN> VIA <E19-BRUSSEL-ANTWERPEN>>)

time: 8.995906 secs. 81 new concepts. 3156 tasks
fppd: 409 new propositions; 3393 requests; 176 computed; 0 denied
=====================================================================
```

In the next experiment we measure the result of changing one subject, e.g. the average-speed of a particular motorway. We see that it only causes 24 propositions to be denied.

```
=====================================================================
expression: (defsubject AVERAGE-SPEED of E19-BRUSSEL-ANTWERPEN [kms/h 50])

AVERAGE-SPEED

time: 0.011258956 secs. 0 new concepts. 0 tasks
fppd: 2 new propositions; 1 requests; 0 computed; 24 denied
=====================================================================
```

Computing the new trajectories goes very fast now.

```
=====================================================================
expression: (INSPECT-TRAJECTORY (>> OF SHORTEST-TRAJECTORY-ASSE-ANTWERPEN))

Trajectory: <SHORTEST-TRAJECTORY-ASSE-ANTWERPEN>;
distance: <KMS 46>;
estimated-duration: <DURATION 55 '>;
via: (<TRAJECTORY <ASSE> <BRUSSEL> VIA <N9-BRUSSEL-GENT>>
      <TRAJECTORY <BRUSSEL> <ANTWERPEN> VIA <E19-BRUSSEL-ANTWERPEN>>)

time: 0.9283521 secs. 1 new concepts. 158 tasks
fppd: 3 new propositions; 196 requests; 6 computed; 0 denied
=====================================================================


=====================================================================
expression: (INSPECT-TRAJECTORY (>> OF FASTEST-TRAJECTORY-ASSE-ANTWERPEN))

Trajectory: <FASTEST-TRAJECTORY-ASSE-ANTWERPEN>;
distance: <KMS 56>;
estimated-duration: <DURATION 48 '>;
via: (<TRAJECTORY <ASSE> <LOKEREN> VIA <N47-ASSE-LOKEREN>>
      <TRAJECTORY <LOKEREN> <ANTWERPEN> VIA <E17-MENEN-ANTWERPEN>>)

time: 4.784033 secs. 9 new concepts. 8866 tasks
fppd: 40 new propositions; 9228 requests; 28 computed; 0 denied
=====================================================================
```

After all those experiments, we have the following state of the concept and proposition-graphs:

```
Total number of concepts: 1511
Total number of propositions: 8327
Total number of requested values: 65400
Total number of computed values: 3377
Total number of denied values: 798
```

1.5. The Use of Cliches.

In the previous sections, we added various subjects to the concept Location in order to run the Simulator. We now use these subjects to illustrate the use of cliches.

1.5.1. What are Cliches ?

The use of Cliches was one of KRS's original design objectives. *Cliches* are based on work by Viviane Jonckers [Jonckers87] and by recent tendencies in programming environments [Rich78, Waters81, Abbott87]. It consists of *algorithmic concepts* which can be used to describe a computational operation on input data at a conceptual level. A typical example of an abstract algorithmic concept is a Filter, which takes an input list and a predicate, and which returns a new list in which all elements of the original list which do not satisfy the predicate are removed. A cliche library is described in the manual (section 6.3).

The notion of programming with cliches was already considered in the early KRS design phases [Steels85b] for the following reasons:

- Cliches make programming knowledge explicit. This knowledge can be employed for developments in automatic programming and programming by example.
- Cliches make explicit how a value will be computed. Particular programs may use not only computed values but also how they were computed. Cliches make this information accessible in a systematic way.
- Cliches are useful for rapid prototyping. Though an experienced LISP programmer may write down as easy a LISP-form which is equivalent to the cliche, updating a cliche at the conceptual level is usually faster and more relyable than updating this equivalent LISP-form.

1.5.2. Example.

The first three experiments use the previous concept Location. The first and second experiment both ask for the motorway-trajectory between Mechelen and Kortrijk. Obviously, in the second, everything was cached. The fact that there are still 134 tasks processed is mainly caused by the computation of the pnames of the printed concepts. The third experiment tries a different but similar question. Also here we still benefit a lot from the cached values.

```
=====================================================================
expression: (INSPECT-TRAJECTORY (>> OF MOTORWAY-TRAJECTORY-MECHELEN-KORTRIJK))

Trajectory: <MOTORWAY-TRAJECTORY-MECHELEN-KORTRIJK>;
distance: <KMS 109>;
estimated-duration: <DURATION 1 H. 14 '>;
via: (<TRAJECTORY <MECHELEN> <BRUSSEL> VIA <E19-BRUSSEL-ANTWERPEN>>
      <TRAJECTORY <BRUSSEL> <GENT> VIA <E40-BRUSSEL-OOSTENDE>>
      <TRAJECTORY <GENT> <KORTRIJK> VIA <E17-MENEN-ANTWERPEN>>)

time: 10.172978 secs. 142 new concepts. 2913 tasks
fppd: 766 new propositions; 2989 requests; 345 computed; 0 denied
=====================================================================


=====================================================================
expression: (INSPECT-TRAJECTORY (>> OF MOTORWAY-TRAJECTORY-MECHELEN-KORTRIJK))

Trajectory: <MOTORWAY-TRAJECTORY-MECHELEN-KORTRIJK>;
distance: <KMS 109>;
estimated-duration: <DURATION 1 H. 14 '>;
via: (<TRAJECTORY <MECHELEN> <BRUSSEL> VIA <E19-BRUSSEL-ANTWERPEN>>
      <TRAJECTORY <BRUSSEL> <GENT> VIA <E40-BRUSSEL-OOSTENDE>>
      <TRAJECTORY <GENT> <KORTRIJK> VIA <E17-MENEN-ANTWERPEN>>)

time: 1.1023542 secs. 0 new concepts. 134 tasks
fppd: 0 new propositions; 185 requests; 0 computed; 0 denied
=====================================================================


=====================================================================
expression: (INSPECT-TRAJECTORY (>> OF MOTORWAY-TRAJECTORY-MECHELEN-MENEN))

Trajectory: <MOTORWAY-TRAJECTORY-MECHELEN-MENEN>;
distance: <KMS 116>;
estimated-duration: <DURATION 1 H. 18 '>;
via: (<TRAJECTORY <MECHELEN> <BRUSSEL> VIA <E19-BRUSSEL-ANTWERPEN>>
      <TRAJECTORY <BRUSSEL> <GENT> VIA <E40-BRUSSEL-OOSTENDE>>
      <TRAJECTORY <GENT> <MENEN> VIA <E17-MENEN-ANTWERPEN>>)

time: 1.8198568 secs. 38 new concepts. 861 tasks
fppd: 194 new propositions; 972 requests; 88 computed; 0 denied
=====================================================================
```

We now define the subjects "direct-trajectories-fastest-first", "located-at-motorways" and "motorway-trajectories" using cliches.

```
(defsubject DIRECT-TRAJECTORIES-FASTEST-FIRST of LOCATION
  (a list-of-trajectories
    (definition
      (a sort
        (sort-what (>> direct-trajectories))
        (sort-predicate
          [clambda (?x ?y)
            (>> (smaller-than (>> estimated-duration of ?y))
                estimated-duration of ?x)])))))
```

```
(defsubject LOCATED-AT-MOTORWAYS of LOCATION
  (a list-of-roads
    (definition
      (a filter
        (filter-over (>> located-at))
        (filter-with [variable ?one-road])
        (filter-predicate
          [form (if (subtype-p ?one-road (>> of motorway))
                    (>> of true)
                    (>> of false))])))))

(defsubject MOTORWAY-TRAJECTORIES of LOCATION
 (a list-of-trajectories
   (definition
     (a filter
       (filter-over (>> direct-trajectories))
       (filter-with [variable ?one-trajectory])
       (filter-predicate
         [form (if (subtype-p (>> using of ?one-trajectory) (>> of motorway))
                   (>> of true)
                   (>> of false))])))))
```

The same experiments are repeated with the new version of these subjects. We see in the third experiment that the numbers are hardly higher then in the corresponding test with the old version.

```
========================================================================
expression: (INSPECT-TRAJECTORY (>> OF MOTORWAY-TRAJECTORY-MECHELEN-KORTRIJK))

Trajectory: <MOTORWAY-TRAJECTORY-MECHELEN-KORTRIJK>;
distance: <KMS 109>;
estimated-duration: <DURATION 1 H. 14 '>;
via: (<TRAJECTORY <MECHELEN> <BRUSSEL> VIA <E19-BRUSSEL-ANTWERPEN>>
      <TRAJECTORY <BRUSSEL> <GENT> VIA <E40-BRUSSEL-OOSTENDE>>
      <TRAJECTORY <GENT> <KORTRIJK> VIA <E17-MENEN-ANTWERPEN>>)

time: 4.0204706 secs. 122 new concepts. 1750 tasks
fppd: 660 new propositions; 1950 requests; 264 computed; 0 denied
========================================================================


========================================================================
expression: (INSPECT-TRAJECTORY (>> OF MOTORWAY-TRAJECTORY-MECHELEN-KORTRIJK))

Trajectory: <MOTORWAY-TRAJECTORY-MECHELEN-KORTRIJK>;
distance: <KMS 109>;
estimated-duration: <DURATION 1 H. 14 '>;
via: (<TRAJECTORY <MECHELEN> <BRUSSEL> VIA <E19-BRUSSEL-ANTWERPEN>>
      <TRAJECTORY <BRUSSEL> <GENT> VIA <E40-BRUSSEL-OOSTENDE>>
      <TRAJECTORY <GENT> <KORTRIJK> VIA <E17-MENEN-ANTWERPEN>>)

time: 1.0061413 secs. 0 new concepts. 134 tasks
fppd: 0 new propositions; 185 requests; 0 computed; 0 denied
========================================================================
```

```
=====================================================================
expression: (INSPECT-TRAJECTORY (>> OF MOTORWAY-TRAJECTORY-MECHELEN-MENEN))

Trajectory: <MOTORWAY-TRAJECTORY-MECHELEN-MENEN>;
distance: <KMS 116>;
estimated-duration: <DURATION 1 H. 18 '>;
via: (<TRAJECTORY <MECHELEN> <BRUSSEL> VIA <E19-BRUSSEL-ANTWERPEN>>
      <TRAJECTORY <BRUSSEL> <GENT> VIA <E40-BRUSSEL-OOSTENDE>>
      <TRAJECTORY <GENT> <MENEN> VIA <E17-MENEN-ANTWERPEN>>)

time: 1.408393 secs. 5 new concepts. 770 tasks
fppd: 26 new propositions; 850 requests; 18 computed; 0 denied
=====================================================================
```

The final state of the concept and proposition-graphs is the following:

```
Total number of concepts: 1818
Total number of propositions: 9976
Total number of requested values: 72550
Total number of computed values: 4092
Total number of denied values: 850
```

1.6. Auxiliary Concepts.

This section shows the definitions of some additional concepts which were used throughout the example.

SPECIAL-INSTANCE-CREATOR FOR TABLES.

```
(defconcept TABLE-META-INTERPRETER
  (a data-concept-meta-interpreter
    (special-instance-creator table-sic)))

(defconcept TABLE-SIC
  [function table-sic])

(defun TABLE-SIC (type info ks::distance ks::krs-env ks::lisp-env)
  #~(a ~type
      (definition [form (mapcar #'(lambda (x)
                                    (cons (parse-concept-description (first x)
                                              ks::distance ks::krs-env ks::lisp-env)
                                          (parse-concept-description (second x)
                                              ks::distance ks::krs-env ks::lisp-env)))
                                ~info)])))
```

KM, KMS, KMS/H

```
(defconcept KM
  (a number))

(defconcept KMS
  (a number
    (meta-interpreter-type eager-data-concept-meta-interpreter)))

(defconcept KMS/H
  (a number))
```

DURATION

```
(defconcept DURATION
  (a number
    (meta-interpreter-type duration-meta-interpreter)
    (minutes (a number
               (definition [form (mod (floor (>> referent)) 60)])))
    (hours (a number
             (definition [form (floor (>> referent) 60)])))))

(defconcept DURATION-META-INTERPRETER
  (a data-concept-meta-interpreter
    (pname (a list
             (definition
               [form (let* ((concept-name (>> referent concept-name))
                            (concept-name-in-type
                             (or concept-name (>> referent concept-name-in-type))))
                       (cond
                        (concept-name (list concept-name))
                        (t (let ((min (>> referent minutes referent))
                                 (hours (>> referent hours referent)))
                             (cons concept-name-in-type
                                   (cond
                                    ((zerop hours) (list min #'))
                                    ((zerop min) (list hours 'h.))
                                    (t (list hours 'h. min #')))))))))])))))
```

SORT

```
(defconcept SORT
  (a cliche
    (sort-what (a concept-list))
    (sort-predicate (a function))
    (definition
      [form `(sort (>> referent of ,(>> sort-what))
                   ',(>> referent sort-predicate))])))
```

META-INTERPRETER'S FOR TRAJECTORIES

```
(defconcept SIMPLE-TRAJECTORY-META-INTERPRETER
  (a meta-interpreter
    (pname (a list
             (definition
               [form (let ((concept-name (>> referent concept-name)))
                       (if concept-name
                           (list concept-name)
                           (list 'trajectory (>> from referent)
                                 (>> to referent)
                                 'via
                                 (>> using referent))))])))))
```

```
(defconcept COMBINED-TRAJECTORY-META-INTERPRETER
  (a meta-interpreter
    (pname (a list
             (definition
               [form (let ((concept-name (>> referent concept-name)))
                       (if concept-name
                           (list concept-name)
                           (list (>> referent concept-name-in-type)
                                 (>> from referent)
                                 (>> to referent))))])))))
```

INSPECT-TRAJECTORY

```
(defun INSPECT-TRAJECTORY (trajectory)
  (format t "~&~%Trajectory: ~S; ~%distance: ~S; ~%estimated-duration: ~S;~%via: ~S~% "
          trajectory
          (>> distance of ~trajectory)
          (>> estimated-duration of ~trajectory)
          (>> referent using of ~trajectory))
  '*)
```

1.7. Conclusion.

The most common way to use KRS is for data modeling. There is a direct mapping between the objects in the representation domain and the concepts in the concept-graph. It is in particular useful for the representation of static facts about these objects.

To have an optimal use of KRS's caching mechanism, we use as much named concepts as possible and we to make too many new instances of a concept. When using the same concepts many times, more and more information is cached and easily re-used. By using named concepts, we can easily access the same concepts many times.

We also illustrated the use of Cliches. Cliches are algorithmic concepts, which allow us to define aspects of a concept in terms of other aspects at a very high level.

Finally this example demonstrates how large the concept-graph and the FPPD proposition-graph grow for a typical KRS application, and the ease (= speed) with which the KRS interpreter can deal with those sizes. It was also demonstrated that the use of cliches did not significantly slow down the systems behavior.

2. CLASSIFICATION.

2.1. Introduction.

Inheritance via the type-subject is the standard data-retrieval mechanism in KRS. We demonstrate how different data-retrieval mechanisms can be explicitly built in KRS. We choose to model a classification hierarchy. Classification is a deduction mechanism at the basis of many well known representation languages ([Brachman85a, Hewitt80]).

The mechanism is exemplified in the office-domain, in particular inspired by our own administration. An example classification-hierarchy selects the contract by which a trip can be sponsored. The representation of this hierarchy constitutes the core of this example. Besides this hierarchy, the example also has some knowledge about contracts, persons, locations, sponsors and research-domains.

2.2. The concepts Class and Case.

The classification formalism embodies two generic concepts, i.e. the concept Class and the concept Case. A class is a node in a classification hierarchy, a case is a particular occurrence to be classified.

The specializations of a class are the descendants of the class in the hierarchy. The constraint of a class is a function with one argument. A case which satisfies the constraint belongs to the class. This is tested with the subject "belongs-to". The subject "classify" takes a case as argument and selects to which from its subclasses (itself included) the case belongs. It may return multiple classes. This process is implemented with the LISP-function Classify.

```
(defconcept CLASS
  (specializations [list-of-classes ()])
  (constraint (a function))
  ((classify
     (a subject
       (arguments [args ?case])
       (definition
         [form (a list-of-classes
                 (definition [form (classify ?case (>>))]))])))))
  ((belongs-to
     (a subject
       (arguments [args ?case])
       (definition
         [form (>> (apply ?case) constraint)])))))
```

The subject "classification-structure" of the concept Case contains the top-class by which the case must be classified. The subject matching-classes computes the result of classifying the case in its classification-structure.

```
(defconcept CASE
  (classification-structure (a class))
  (matching-classes (>> (classify (>>)) classification-structure)))
```

The function Classify implements the classification process. Classification is guided by testing whether the case belongs to the classes, i.e. by applying the corresponding constraints to the case. If this application returns false, the classification fails and is not explored deeper. If it returns unknown, the classification is the result of collecting all recursive classifications for all specializations of the class. If the application of the constraint returns true, the classification returns the same result except if all deeper classifications fail. In this case the class itself is returned.

```
(defun CLASSIFY (case class)
  (clet* ((?case case)
          (?class class)
          (?belongs-to (>> (belongs-to ?case) of ?class)))
    (unless (>> false-p of ?belongs-to)
      (or (mapcan #'(lambda (one-specialization)
                      (classify ?case ~one-specialization))
                  (>> referent specializations of ?class))
          (when (>> true-p of ?belongs-to)
            (list class))))))
```

A list of classes is a concept-list with referent a list of class-concepts.

```
(defconcept LIST-OF-CLASSES
  (a concept-list
    (element-type class)))
```

2.3. The Example Classification Hierarchy.

As we have said before, we apply this classification formalism in a domain provided by our own administration. When somebody in the laboratory wants to participate in a conference, it is a complex decision procedure to find a contract to sponsor this trip. A simplified version of this procedure is here implemented with a classification hierarchy.

Figure 2 shows the knowledge. The classification hierarchy is shown in tree-1. The sons of a node in this tree are the elements of the referent of the specializations of the concept represented by the node. Tree-2 shows a part of the type-tree, i.e. the instances of the concept Domain. Then there is also an inspector-window describing a trip by Pattie to New-York, for which the classification process deduced that it will be sponsored by the Impuls Autonomous Agent project.

THE CLASSIFICATION-HIERARCHY

Each trip belongs to the class Sponsored-Trip.

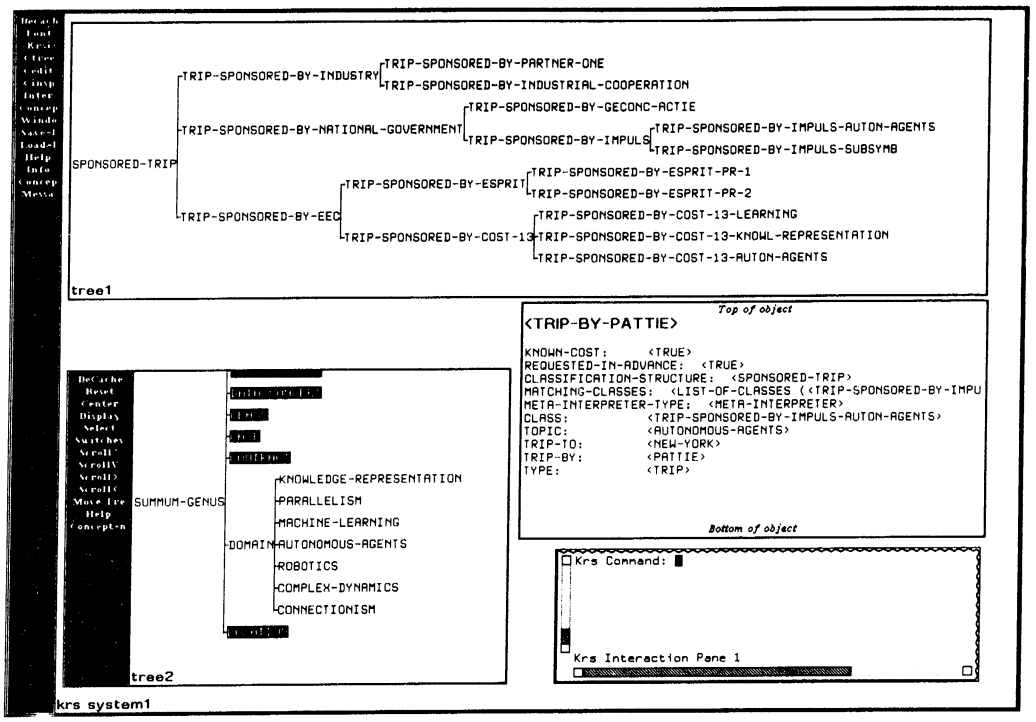


Fig 3: The trip knowledge-base.

```
(defconcept SPONSORED-TRIP
  (a class
    (constraint
      [clambda (?trip) (>> of true)])
    (specializations
      [list-of-classes (trip-sponsored-by-industry
                        trip-sponsored-by-national-government
                        trip-sponsored-by-eec)])))
```

Only trips within Europe can be sponsored by the industry.

```
(defconcept TRIP-SPONSORED-BY-INDUSTRY
  (a class
    (specializations
      [list-of-classes (trip-sponsored-by-partner-one
                        trip-sponsored-by-industrial-cooperation)])
    (constraint
      [clambda (?trip) (>> (and (>> in-europe trip-to of ?trip))
                           of unknown)])))
```

A trip sponsored by the national government must always be requested in advance.

```
(defconcept TRIP-SPONSORED-BY-NATIONAL-GOVERNMENT
  (a class
    (specializations
      [list-of-classes (trip-sponsored-by-geconc-actie
                        trip-sponsored-by-impuls)])
    (constraint
      [clambda (?trip) (>> (and (>> requested-in-advance of ?trip))
                           of unknown)])))
```

```
(defconcept TRIP-SPONSORED-BY-EEC
  (a class
    (specializations
      [list-of-classes (trip-sponsored-by-esprit
                        trip-sponsored-by-cost-13)])
    (constraint
      [clambda (?trip) (>> of unknown)])))
```

A trip can be sponsored by the Partner-One-Contract if the topic fits within the contract, the cost of the trip is known in advance, the traveler is known by the contract and the trip is requested in advance.

```
(defconcept TRIP-SPONSORED-BY-PARTNER-ONE
  (a class
    (contract partner-one-contract)
    (constraint
      [clambda (?trip)
        (>> (and (>> (member (>> topic of ?trip))
                     research-domains contract))
            (and (>> known-cost of ?trip))
            (and (>> (member (>> trip-by of ?trip)) knows-of contract))
            requested-in-advance of ?trip)])))
```

A trip can be sponsored by the Industrial-Cooperation-Contract if the topic of the trip fits with the research-domains specified in the contract, the cost of the trip is known in advance, the visitor of the trip is employed by this contract, and the trip is requested in advance.

```
(defconcept TRIP-SPONSORED-BY-INDUSTRIAL-COOPERATION
  (a class
    (contract industrial-cooperation-contract)
    (constraint
      [clambda (?trip)
        (>> (and (>> (member (>> topic of ?trip))
                     research-domains contract))
            (and (>> known-cost of ?trip))
            (and (>> (member (>> trip-by of ?trip)) employs contract))
            requested-in-advance of ?trip)])))
```

A trip can be sponsored by the contract "geconcerteerde actie" if the topic fits in the contract, the cost is known in advance and the trip was requested in advance.

```
(defconcept TRIP-SPONSORED-BY-GECONC-ACTIE
  (a class
    (contract geconc-actie-contract)
    (constraint
      [clambda (?trip)
        (>> (and (>> (member (>> topic of ?trip))
                     research-domains contract))
            (and (>> known-cost of ?trip))
            requested-in-advance of ?trip)])))
```

```
(defconcept TRIP-SPONSORED-BY-IMPULS
  (a class
    (specializations
      [list-of-classes (trip-sponsored-by-impuls-auton-agents
                        trip-sponsored-by-impuls-subsymb)])
    (constraint
      [clambda (?trip) (>> of unknown)])))
```

A trip can not be sponsored by Esprit if it is outside of Europe.

```
(defconcept TRIP-SPONSORED-BY-ESPRIT
  (a class
    (specializations
      [list-of-classes (trip-sponsored-by-esprit-pr-1
                        trip-sponsored-by-esprit-pr-2)])
    (constraint
      [clambda (?trip) (>> (and (>> in-europe trip-to of ?trip))
                           of unknown)])))
```

```
(defconcept TRIP-SPONSORED-BY-COST-13
  (a class
    (specializations
      [list-of-classes (trip-sponsored-by-cost-13-learning
                        trip-sponsored-by-cost-13-knowl-representation
                        trip-sponsored-by-cost-13-auton-agents)])
    (constraint
      [clambda (?trip) (>> of unknown)])))
```

A trip can be sponsored by the A.A.-impuls-contract if the traveler is known by the contract and the trip is requested in advance. Furthermore, if the traveler is the promoter of the contract, then the topic must fit within the contract.

```
(defconcept TRIP-SPONSORED-BY-IMPULS-AUTON-AGENTS
  (a class
    (contract auton-agent-impuls-contract)
    (constraint
      [clambda (?trip)
        (>> (and (>> (member (>> trip-by of ?trip)) knows-of contract))
            (and (>> requested-in-advance of ?trip))
            (or (>> not (equal (>> trip-by of ?trip))
                       promoter contract))
            (member (>> topic of ?trip))
            research-domains contract)])))
```

A trip can be sponsored by the subsymb-impuls-contract if the traveler is known by the contract and the trip is requested in advance. Furthermore, if the traveler is the promoter of the contract, then the topic must fit within the contract.

```
(defconcept TRIP-SPONSORED-BY-IMPULS-SUBSYMB
  (a class
    (contract subsymb-impuls-contract)
    (constraint
      [clambda (?trip)
        (>> (and (>> (member (>> trip-by of ?trip)) knows-of contract))
            (and (>> requested-in-advance of ?trip))
            (or (>> not (equal (>> trip-by of ?trip))
                       promoter contract))
            (member (>> topic of ?trip))
            research-domains contract)])))
```

A trip can be sponsored by the Esprit-contract PR-1 if it is inside Europe.

```
(defconcept TRIP-SPONSORED-BY-ESPRIT-PR-1
  (a class
    (contract pr-1-esprit-contract)
    (constraint
      [clambda (?trip)
        (>> in-europe trip-to of ?trip)])))
```

A trip can be sponsored by the Esprit-contract PR-2 if it is inside Europe and if it is requested in advance.

```
(defconcept TRIP-SPONSORED-BY-ESPRIT-PR-2
  (a class
    (contract pr-2-esprit-contract)
    (constraint
      [clambda (?trip)
        (>> (and (>> in-europe trip-to of ?trip))
            requested-in-advance of ?trip)])))
```

A trip can be sponsored by the cost-13-machine-learning-contract if the topic fits with the research-domains in the contract.

```
(defconcept TRIP-SPONSORED-BY-COST-13-LEARNING
  (a class
    (contract learning-cost-13-contract)
    (constraint
      [clambda (?trip)
        (>> (member (>> topic of ?trip))
            research-domains contract)])))
```

A trip can be sponsored by the cost-13-contract on knowledge representation if the topic fits with the research-domains in the contract.

```
(defconcept TRIP-SPONSORED-BY-COST-13-KNOWL-REPRESENTATION
  (a class
    (contract knowl-representation-cost-13-contract)
    (constraint
      [clambda (?trip)
        (>> (member (>> topic of ?trip))
            research-domains contract)])))
```

A trip can be sponsored by the cost-13-contract on autonomous agents if the topic fits with the research-domains in the contract.

```
(defconcept TRIP-SPONSORED-BY-COST-13-AUTON-AGENTS
  (a class
    (contract auton-agents-cost-13-contract)
    (constraint
      [clambda (?trip)
        (>> (member (>> topic of ?trip))
            research-domains contract)])))
```

TRIP

A trip is a case. Its classification-structure is the concept Sponsored Trip. The class of a trip is one of its matching-classes, to be selected by the user. The trip-by of a trip is a person and the trip-to is a location. The subjects "known-cost" and "requested-in-advance" are query-subjects. Their filler is asked to the user when they are needed.

```
(defconcept TRIP
  (a case
    (classification-structure sponsored-trip)
    ((class (a query-subject
              (possibilities (>> matching-classes)))))
    (trip-by (a person))
    (trip-to (a location))
    ((known-cost (a query-subject
                   (possibilities [concept-list (true false)])))))
    ((requested-in-advance (a query-subject
                             (possibilities [concept-list (true false)])))))))
```

2.4. Domain Knowledge.

CONTRACTS

```
(defconcept CONTRACT
  (employs (a list-of-persons))
  (promoter (a person))
  (knows-of (>> (cons (>> promoter)) employs))
  (research-domains (a list-of-domains))
  (sponsor (a sponsor)))


(defconcept PARTNER-ONE-CONTRACT
  (a contract
    (employs [list-of-persons Peter])
    (promoter Luc)
    (research-domains [list-of-domains expert-systems])
    (sponsor partner-one)))


(defconcept INDUSTRIAL-COOPERATION-CONTRACT
  (a contract
    (employs [list-of-persons Bernard walter])
    (promoter Luc)
    (research-domains [list-of-domains expert-systems knowledge-representation])
    (sponsor industrial-cooperation)))
```

```
(defconcept GECONC-ACTIE-CONTRACT
  (a contract
    (employs [list-of-persons Eric])
    (promoter Luc)
    (research-domains [list-of-domains parallelism])
    (sponsor national-government)))

(defconcept AUTON-AGENTS-IMPULS-CONTRACT
  (a contract
    (employs [list-of-persons Jo Robin])
    (promoter Pattie)
    (research-domains [list-of-domains autonomous-agents robotics])
    (sponsor impuls)))

(defconcept SUBSYMB-IMPULS-CONTRACT
  (a contract
    (employs [list-of-persons Jan Piet])
    (promoter Luc)
    (research-domains [list-of-domains complex-dynamics connectionism robotics])
    (sponsor impuls)))

(defconcept LEARNING-COST-13-CONTRACT
  (a contract
    (employs [list-of-persons ])
    (promoter walter)
    (research-domains [list-of-domains machine-learning])
    (sponsor cost-13)))

(defconcept KNOWL-REPRESENTATION-COST-13-CONTRACT
  (a contract
    (employs [list-of-persons ])
    (promoter Pattie)
    (research-domains [list-of-domains knowledge-representation parallelism])
    (sponsor cost-13)))

(defconcept AUTON-AGENTS-COST-13-CONTRACT
  (a contract
    (employs [list-of-persons ])
    (promoter Pattie)
    (research-domains [list-of-domains autonomous-agents])
    (sponsor cost-13)))

(defconcept PR-1-ESPRIT-CONTRACT
  (a contract
    (employs [list-of-persons Walter-d Karina])
    (promoter Luc)
    (research-domains [list-of-domains intelligent-office-systems])
    (sponsor esprit)))

(defconcept PR-2-ESPRIT-CONTRACT
  (a contract
    (employs [list-of-persons kris Peter-b Didier])
    (promoter Luc)
    (research-domains [list-of-domains knowledge-representation parallelism])
    (sponsor esprit)))
```

PERSONS

```
(defconcept PERSON
  ((equal (a subject
            (arguments [args ?other])
            (definition [form (if (eq (>>) ?other)
                                  (>> of true)
                                  (>> of false))])))))

(defconcept LIST-OF-PERSONS
  (a sequence
    (element-type person)))

(defconcept JAN
  (a person))

(defconcept PIET
  (a person))

(defconcept PETER
  (a person))

(defconcept BERNARD
  (a person))

(defconcept WALTER
  (a person))

(defconcept LUC
  (a person))

(defconcept ERIC
  (a person))

(defconcept PATTIE
  (a person))

(defconcept JO
  (a person))

(defconcept ROBIN
  (a person))

(defconcept KRIS
  (a person))

(defconcept PETER-B
  (a person))

(defconcept DIDIER
  (a person))

(defconcept WALTER-D
  (a person))

(defconcept KARINA
  (a person))
```

RESEARCH-DOMAINS

```
(defconcept DOMAIN)

(defconcept LIST-OF-DOMAINS
  (a sequence
    (element-type domain)))
```

```
(defconcept EXPERT-SYSTEMS
  (a domain))

(defconcept KNOWLEDGE-REPRESENTATION
  (a domain))

(defconcept PARALLELISM
  (a domain))

(defconcept MACHINE-LEARNING
  (a domain))

(defconcept AUTONOMOUS-AGENTS
  (a domain))
```

```
(defconcept ROBOTICS
  (a domain))

(defconcept COMPLEX-DYNAMICS
  (a domain))

(defconcept CONNECTIONISM
  (a domain))

(defconcept INTELLIGENT-OFFICE-SYSTEMS
  (a domain))
```

LOCATIONS

```
(defconcept LOCATION
  (in-europe unknown))
```

```
(defconcept PARIS
  (a location
    (in-europe true)))
```

```
(defconcept NEW-YORK
  (a location
    (in-europe false)))
```

SPONSORS

```
(defconcept SPONSOR)
```

```
(defconcept PARTNER-ONE
  (a sponsor))

(defconcept INDUSTRIAL-COOPERATION
  (a sponsor))

(defconcept ESPRIT
  (a sponsor))
```

```
(defconcept NATIONAL-GOVERNMENT
  (a sponsor))

(defconcept IMPULS
  (a sponsor))

(defconcept COST-13
  (a sponsor))
```

2.5. A Trace of the Classification Process.

As a first illustration we classify a trip by Kris to Paris concerning intelligent office systems.

```
(>> referent matching-classes of
    (a trip
      (trip-by kris)
      (trip-to paris)
      (topic intelligent-office-systems)))
```

A trace of the classes visited by the classification algorithm is shown below. Notice that while testing the constraint of the class Trip-Sponsored-By-Partner-one, the user is asked to give the fillers of the subjects "requested-in-advance" and "cost-known". The answers are False and True respectively. Notice also that the specializations of the class Trip-Sponsored-By-National-Government are not explored, because the trip

was not requested in advance. The only matching class it finds is the class Trip-Sponsored-By-Esprit-PR-1.

```
Classifying <TRIP #40> in <SPONSORED-TRIP>
Classifying <TRIP #40> in <TRIP-SPONSORED-BY-INDUSTRY>
Classifying <TRIP #40> in <TRIP-SPONSORED-BY-PARTNER-ONE>

   Querying REQUESTED-IN-ADVANCE of <TRIP #40>
   User's reply: <FALSE>

   Querying KNOWN-COST of <TRIP #40>
   User's reply: <TRUE>

Classifying <TRIP #40> in <TRIP-SPONSORED-BY-INDUSTRIAL-COOPERATION>
Classifying <TRIP #40> in <TRIP-SPONSORED-BY-NATIONAL-GOVERNMENT>
Classifying <TRIP #40> in <TRIP-SPONSORED-BY-EEC>
Classifying <TRIP #40> in <TRIP-SPONSORED-BY-ESPRIT>
Classifying <TRIP #40> in <TRIP-SPONSORED-BY-ESPRIT-PR-1>
Classifying <TRIP #40> in <TRIP-SPONSORED-BY-ESPRIT-PR-2>
Classifying <TRIP #40> in <TRIP-SPONSORED-BY-COST-13>
Classifying <TRIP #40> in <TRIP-SPONSORED-BY-COST-13-LEARNING>
Classifying <TRIP #40> in <TRIP-SPONSORED-BY-COST-13-KNOWL-REPRESENTATION>
Classifying <TRIP #40> in <TRIP-SPONSORED-BY-COST-13-AUTON-AGENTS>

(<TRIP-SPONSORED-BY-ESPRIT-PR-1>)
```

The second illustration is an attempt to classify a trip by Kris to New-York, again concerning intelligent office systems.

```
(>> referent matching-classes of
    (a trip
      (trip-by kris)
      (trip-to new-york)
      (topic intelligent-office-systems)))
```

This time neither the specialization of the class Trip-Sponsored-By-Industry nor those of the class Trip-Sponsored-By-Esprit are explored, since it is a trip outside Europe. The only matching class is Sponsored-Trip, i.e. the top of the classification-hierarchy. This means that the trip cannot be sponsored by any of the contracts.

```
Classifying <TRIP #43> in <SPONSORED-TRIP>
Classifying <TRIP #43> in <TRIP-SPONSORED-BY-INDUSTRY>
Classifying <TRIP #43> in <TRIP-SPONSORED-BY-NATIONAL-GOVERNMENT>

   Querying REQUESTED-IN-ADVANCE of <TRIP #43>
   User's reply: <TRUE>

Classifying <TRIP #43> in <TRIP-SPONSORED-BY-GECONC-ACTIE>

   Querying KNOWN-COST of <TRIP #43>
   User's reply: <TRUE>

Classifying <TRIP #43> in <TRIP-SPONSORED-BY-IMPULS>
Classifying <TRIP #43> in <TRIP-SPONSORED-BY-IMPULS-AUTON-AGENTS>
Classifying <TRIP #43> in <TRIP-SPONSORED-BY-IMPULS-SUBSYMB>
Classifying <TRIP #43> in <TRIP-SPONSORED-BY-EEC>
Classifying <TRIP #43> in <TRIP-SPONSORED-BY-ESPRIT>
Classifying <TRIP #43> in <TRIP-SPONSORED-BY-COST-13>
Classifying <TRIP #43> in <TRIP-SPONSORED-BY-COST-13-LEARNING>
Classifying <TRIP #43> in <TRIP-SPONSORED-BY-COST-13-KNOWL-REPRESENTATION>
Classifying <TRIP #43> in <TRIP-SPONSORED-BY-COST-13-AUTON-AGENTS>

(<SPONSORED-TRIP>)
```

A possible remedy for this problem is to change the topic of the trip. Is we say that the trip is about knowledge representation, it can be sponsored by cost-13:

```
(>> referent matching-classes of
    (a trip
      (trip-by kris)
      (trip-to new-york)
      (topic knowledge-representation)))


Classifying <TRIP #46> in <SPONSORED-TRIP>
Classifying <TRIP #46> in <TRIP-SPONSORED-BY-INDUSTRY>
Classifying <TRIP #46> in <TRIP-SPONSORED-BY-NATIONAL-GOVERNMENT>

   Querying REQUESTED-IN-ADVANCE of <TRIP #46>
   User's reply: <FALSE>

Classifying <TRIP #46> in <TRIP-SPONSORED-BY-EEC>
Classifying <TRIP #46> in <TRIP-SPONSORED-BY-ESPRIT>
Classifying <TRIP #46> in <TRIP-SPONSORED-BY-COST-13>
Classifying <TRIP #46> in <TRIP-SPONSORED-BY-COST-13-LEARNING>
Classifying <TRIP #46> in <TRIP-SPONSORED-BY-COST-13-KNOWL-REPRESENTATION>
Classifying <TRIP #46> in <TRIP-SPONSORED-BY-COST-13-AUTON-AGENTS>

(<TRIP-SPONSORED-BY-COST-13-KNOWL-REPRESENTATION>)
```

In the following examples we no longer give the full trace of the classification process. The first case is a trip by Luc to Paris. The topic of the trip is expert systems, and the trip is requested in advance. We see that this gives three possible sponsors.

```
-->  (>> referent matching-classes of
        (a trip
          (trip-by luc)
          (trip-to paris)
          (topic expert-systems)
          (requested-in-advance true)
          (known-cost true)))

<--  (<TRIP-SPONSORED-BY-PARTNER-ONE>
      <TRIP-SPONSORED-BY-ESPRIT-PR-1>
      <TRIP-SPONSORED-BY-ESPRIT-PR-2>)
```

The same trip made by Walter can not be sponsored by PARTNER-ONE, since he is not known by this contract. Instead, this trip can be sponsored by the Industrial-Cooperation-Contract.

```
-->  (>> referent matching-classes of
        (a trip
          (trip-by walter)
          (trip-to paris)
          (topic expert-systems)
          (requested-in-advance true)
          (known-cost true)))

<--  (<TRIP-SPONSORED-BY-INDUSTRIAL-COOPERATION>
      <TRIP-SPONSORED-BY-ESPRIT-PR-1>
      <TRIP-SPONSORED-BY-ESPRIT-PR-2>)
```

2.6. Auxiliary Concepts.

```
(defconcept QUERY-SUBJECT
  (a subject
    (possibilities [concept-list ()])
    (type-in-window query-subject-type-in-window)
    (choose-window
      (a window
        (definition
          [form (tv:make-window 'tv:momentary-menu :label
                                (format nil "Give ~A of ~A: "
                                        (>> referent subject-name)
                                        (>> subject-of))
                                :item-list (mapcar
                                             #'(lambda (one)
                                                 (list (format nil "~A" one) one))
                                             (>> referent possibilities)))])))
    (definition [form (do* ((window (>> referent choose-window))
                            (choice (scl:send window :choose)
                                    (scl:send window :choose)))
                           (choice choice))])))
```

2.7. Conclusion.

The built-in deduction mechanisms in KRS are inheritance and referent-computation. With this example we demonstrated how they can serve to build other deductive mechanisms on top of them. In particular, the example showed the modeling of a classification tree. We used the following strategy.

1 The deduction structure is made explicit. This consists of defining concepts to represent the objects which play a role in the deduction mechanism, e.g. classes and cases for this example.

2 A LISP-function is written to implement the basic search. In this case, the role of this function is to push concepts down the tree as deep as possible and to apply the constraints.

3 This function is called from within a definition. This has two major advantages. First the function will be automatically called when the definition is evaluated, independent of the kind of request by which it was triggered. In our example, the deduction may be invoked by asking for the class of a trip, or by asking for the contract of the class of a trip, or by whatever request that uses the matching classes. The second advantage is that the result will be cached such that it must not be reproduced every time it is needed.

3. DEMONS.

3.1. Introduction.

3.1.1. Outline.

In A.I. terminology, a *demon* typically denotes a procedure which sits around waiting for some event to occur. The procedure automatically starts each time the event occurs. This way, the demon exhibits data-driven computation.

In traditional representation languages, there are but a few predefined kinds of events which can trigger a demon. This is usually the assignment of a value to a slot or the retrieval of a slot's value. In either cases, the demon must be explicitly attached to that slot.

We propose here a more general mechanism for triggering demons. Demons are connected to FPPD's dependency network. This way they can fire (execute the procedure) whenever they detect a mutation.

3.1.2. Demons in Representation Languages.

Demons exist in many forms and under many names in the representation language gallery. They are called monitors, traps, triggers, servants, active-values, etc.

Demons in KRL-0 [Bobrow77], constitute part of the procedural attachment strategy. Procedures can be attached to a slot to be executed as a method to carry out some operation, in which case they call it a servant, or to cause a secondary effect of some event, in which case they call it a demon. A demon can be awakened or triggered when something is about to be done or when something has just been done. All of the primitive data-manipulating operations check for demons, whenever they use or add information.

In FRL [Roberts77] and in KRS-83 [Steels84], slots have if-added and if-changed facets which are filled with a LISP-method. This method is evaluated when the slot's value is initialized or changed respectively. Again, the implementation of the trigger-function is embedded in the primitive functions to set a slot's value (= the absorb-conclusion method in KRS-83).

KEE [Intellicorp84] and LOOPS [Bobrow81] have the notion of active-value. An active-value in KEE is a unit which is again associated to a slot, owned by some other unit. It has itself slots *avput, avget, avadd* and *avrem*. Each of them can contain a method to be invoked in particular circumstances. The avput method is invoked when the value of the monitored slot changes. The avget method is invoked whenever that value is referenced. The avadd method is invoked when the active-

value is attached to the slot it is monitoring and the avrem when it is detached again. The triggering of the active-value is done explicitly by the primitive operators of the slot the active value is attached to. It is therefore put in the *avunits* facet of that slot.

3.1.3. Demons attached to Mutating-Subjects.

Explicitly triggering demons which are attached to a particular slot is straightforwardly done in KRS. We can for example specialize Mutating-Subject such that its mutate-to-subject triggers a set of demons before and after changing the value (1). Similarly, a mutating-subject can be defined which triggers a demon when its value if referenced (2).

```
(defconcept MUTATING-SUBJECT-WITH-MUTATION-DEMONS                          1
  (a mutating-subject
    (before-demons (a list-of-demons))
    (after-demons (a list-of-demons))
    ((mutate-to
       (definition
         [form (prog2 (>> trigger-all before-demons)
                      (>> (redefine-value ?new-value) proposition)
                      (>> trigger-all after-demons))])))))


(defconcept MUTATING-SUBJECT-WITH-REFERENCE-DEMONS                         2
  (a mutating-subject
    (reference-demons (a list-of-demons))
    (definition
      [form (progn (>> trigger-all reference-demons)
                   (>> proposition-value proposition))])))
```

3.2. A Different Solution.

3.2.1. Sketch of the Solution.

We here propose a more general demon-concept. The generality is obtained by not connecting the demon to a particular slot. Instead, it is defined as an independent concept and connected to FPPD's dependency-network.

This connection is realized with an FPPD eager-proposition. An eager-proposition recomputes its value automatically the moment it is denied. In this way, the demon is triggered when it detects a mutation of one of its supports.

This approach gives several advantages.

- It is conceptually cleaner. The demon is separated from the action by which it is to be triggered. Often indeed, both are part of different aspects of the program. When it is not, the demon can still be linked to the action by which it is triggered in the normal KRS way (via a subject).

Take for example dates. In an office-application there may be many actions to be triggered when the current-date changes. However, the demons to be triggered are conceptually part of the office-procedures, and not of the concept Current-Date.

- Our approach is also more flexible. When the demons are attached to slots, they have to be anticipated when these slots are defined. To continue the office-example, all office procedures must be considered the moment Current-Date is modeled. In our approach, monitors are defined in an independent way and can thus be dynamically added any time.
- In the classical approach, demons can only be triggered by something that knows about demons. In KRS terms this would mean that only slots implemented by a Mutating-Subject-With-Mutation-Demons can cause demon triggering. By connecting a demon to the dependency-network, it can be triggered by any mutation, even in an indirect way.
- In the traditional approach, an artificial construct must be set up to define a demon which monitors more than one slot at a time. A demon connected to the dependency network monitors all subjects it is directly or indirectly connected with.

Obviously there is one large disadvantage of this triggering strategy. Just because it has such large capacities, it becomes more difficult to control the triggering of the demon. Therefore, to give a better handle to the demon firing, we add an additional condition named *fire-when*. When it detects a mutation, the demon only fires if this condition is true.

3.2.2. Functional Description.

Demon *monitoring (fire-when (a boolean)) (fire-action (a form)) (deactivated (a boolean))* *[concept]*

A demon is a concept. Once it is activated, it tries to fire whenever it detects a mutation to its monitoring-subject. When a demon tries to fire, it first checks whether its fire-when is True. If so it evaluates its fire-action. Whenever a demon tries to fire, it tests its deactivated-subject. If this is True, the demon is deactivated and can never fire again. The demon also tries to fire once the moment it is activated.

activate *[subject of Demon]*

a subject to activate a demon.

3.2.3. Implementation.

The implementation of the concept Demon is based on an eager-subject. An eager-subject is a subject whose referent is computed using an eager-proposition. Hence

this referent is automatically recomputed whenever it is denied.

The description of the concept Demon is given in (3). The definition of the activate-subject requests among others the filler of the monitoring-subject of the demon. Hence, every mutation which leads to the retraction of this filler also withdraws the referent of the active-subject. Since this is an eager-subject, it will be recomputed immediately.

```
(defconcept DEMON                                                    3
  (monitoring)
  (fire-action (a form))
  (fire-when true)
  (deactivate (a boolean))
  ((activate
     (an eager-subject
       (definition
         [form
           (cond
             ((monitor-deactivated (>>))
              nil)
             ((>> referent fire-when)
              (>> eval referent fire-action)
              (if (>> referent deactivate)
                  (deactivate-monitor (>>))
                  (>> monitoring)))
             ((>> referent deactivate)
              (deactivate-monitor (>>)))
             (t (>> monitoring)))]))))
```

Once the demon is deactivated, it is possible that it tries to fire one more time, caused by dependencies installed during its previous execution. Therefore the first clause in the activate-subject's definition checks whether the demon was not deactivated before (4). This information is stored on the demon's meta-info (5).

```
(defun MONITOR-DEACTIVATED (monitor)                                 4
  (ks::meta-info-element monitor 'ks::monitor-deactivated))


(defun DEACTIVATE-MONITOR (monitor)                                  5
  (ks::new-meta-info-element monitor 'ks::monitor-deactivated t))
```

3.3. Example.

The example models an RS-flip-flop (figure 4). It has two lines called R and S which are normally both True. Each of them can be temporarily set to False. When this happens we say that the line was pulsed.

The Q and the $\bar{Q}$ are two lines remembering which of the lines R or S was pulsed last. Q is True if R was pulsed last, otherwise $\bar{Q}$ is True. Only one of them can be True at the same time.

The two modules in the center are logical-nand-gates. There output is True if one of their inputs is not True. Otherwise it is False.

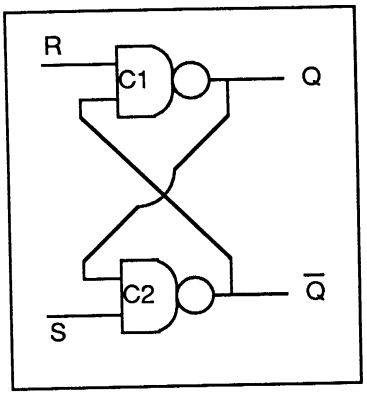


Fig 4: An RS-flip-flop.

Below we define some auxiliary concepts. The concept Line (6) has an input-value and an output-value, both Booleans. The output-value is normally equal to the input-value. The input-value of the out-line of a logical-nand (7) is equal to the not of the and of the output-values of the two in-lines.

```
(defconcept LINE                                                    6
  (input-value (a boolean))
  (output-value (>> input-value)))


(defconcept LOGICAL-NAND                                            7
  (a logical-component
    (inl (a line))
    (in2 (a line))
    (out (a line
           (input-value
             (>> not (and (>> output-value inl)) output-value in2))))))
```

At first sight, the RS-flip-flop could be defined as in (8). This however can not work. To compute the out of Q it needs the out of $\bar{Q}$ and vice versa. This circularity is obviously caused by the feedback-lines which send the output of each nand-gate back as input for the other.

```
(defconcept R-S-FLIP-FLOP                                           8
  (a logical-component
    (r (a line))
    (s (a line))
    (c1 (a logical-nand
          (inl (>> r))
          (in2 (>> q_))))
    (c2 (a logical-nand
          (inl (>> q))
          (in2 (>> s))))
    (q (>> out c1))
    (q_ (>> out c2))))
```

This problem can be solved by simulating a feedback-line. We model a Feedback-Line such that the input-value is explicitly passed to the output-value rather than

computing the output-value from the input-value as a normal line does. We therefore define a demon which detects changes to the input-value and passes this through to the output-value (9).

```
(defconcept FEEDBACK-LINE                                                    9
  (a line
    ((output-value
       (a mutating-subject
         (initial-value true))))
    (feedback-demon
      (a demon
        (monitoring (>> input-value))
        (fire-when [form (>> not (equal (>> input-value)) output-value)])
        (fire-action [form (>> (mutate-to (>> input-value)) handler output-value)])
        (deactivate false)))))
```

The new version of the RS-Flip-Flop has one feedback-line which is the second input-line of the first logical-nand. It is connected to the output-line of the second logical-nand.

```
(defconcept R-S-FLIP-FLOP                                                   10
  (a logical-component
    (r (a line))
    (s (a line))
    (c1 (a logical-nand
          (in1 (>> r))
          (in2 (a feedback-line
                 (input-value (>> output-value out c2))))))
    (c2 (a logical-nand
          (in1 (>> q))
          (in2 (>> s))))
    (q (>> out c1))
    (q_ (>> out c2))))
```

A Pulse-Line is a line which can be pulsed (11). We need it to demonstrate the functioning of the RS-flip-flop.

```
(defconcept PULSE-LINE                                                      11
 (a line
   ((input-value
      (a mutating-subject
        (initial-value true))))
   ((set
     (a subject
       (arguments [args ?new-value])
       (definition [form (>> (mutate-to ?new-value) handler input-value)]))))
   ((pulse
      (an action-subject
        (definition
          [form (progn (>> (set false)) (>> (set true)))]))))))
```

RS-1 is an example flip-flop. Its two input-lines are pulse-lines:

```
(defconcept RS-1
  (a r-s-flip-flop
    (r (a pulse-line))
    (s (a pulse-line))))
```

The function Show-RS prints the state of an RS-flip-flop:

```
(defun show-rs (ct)
  (format t "~&~%Q:   ~A~%Q_:  ~A~%"
          (>> output-value q of ~ct)
          (>> output-value q_ of ~ct)))
```

To function, the feedback-demon must be activated:

```
(>> activate feedback-demon in2 cl of rs-1)"
```

Below we show a trace of the functioning:

```
-->   (>> pulse s of rs-1)
<--   <done>
-->   (show-rs (>> of rs-1))

Q:   <false>
Q_:  <true>

-->   (>> pulse r of rs-1)
<--   <done>
-->   (show-rs (>> of rs-1))

Q:   <true>
Q_:  <false>

-->   (>> pulse s of rs-1)
<--   <done>
-->   (show-rs (>> of rs-1))

Q:   <false>
Q_:  <true>
```

3.4. Conclusion.

Normal KRS reasoning is backward: to respond to a request a definition is evaluated which may result in sending new requests. For a data-driven reasoning mechanism, we can introduce demons.

In this example we showed the implementation of a concept Demon (which is part of the standard KRS concept library). A KRS demon has an innovative KRS triggering mechanism. All existing demons in other representation languages are typically triggered by a primitive operation on a variable or slot (i.e. setting or referencing its value).

A KRS demon fires when it detects a mutation by means of the FPPD dependency network. This has the following advantages:

- It is conceptually cleaner.
- It is more flexible.
- It can be triggered by different sorts of mutations.
- It can detect mutations in an indirect way.

An important disadvantage is that it is more difficult to keep the demon firing under control.

The application of the demons to model an RS-Flip-Flop demonstrated how the two reasoning styles (backward and data-driven) can work together. The feedback-line works in a data-driven way. The moment it detects a mutation of its input-value it recomputes its input-value and it updates its output-value. The rest of the RS-Flip-FLop works in a backward way: subject-fillers are computed the moment they are requested.

The cooperation of the two styles happens in two ways. The backward computation bottoms out when it needs the output-value of a feedback line, because this is always known. The data-driven computation however requests the input-value of the feedback-line, which is deduced in a backward way.

4. A SMALL PRODUCTION SYSTEM.

4.1. Introduction.

This example illustrates how to use KRS to implement an existing knowledge representation formalism, i.e. a production system.

We define simple production-rules with a condition and an action. Production are organized in rule-sets, which are attached to problems. On top of this, a simple forward chaining inference mechanism is implemented with a LISP-function.

Also the memory is explicitly modeled. Data in the memory is stored within FPPD propositions. Hence changing data in the memory requires changing the values of those propositions. This way the conditions of the production-rules may be cached. Cached conditions will be withdrawn if some memory data on which they depend changes. This speeds up the search for the next rule to fire.

A production-rule program is written to do technical diagnosis. The program gives a simplified model of a car's engine. Diagnosis can be done for repairing start problems and problems with the electrical installation.

4.2. A Production-Rule System in KRS.

4.2.1. Modeling Production-Rules.

A production has a condition and an action. The condition is a boolean, and the action is a form. The subject *try-to-fire* asks for the referent of the production's *should-fire* which will be defined below. If this is T, it evaluates the action, after setting the LISP-variable **fires** to T. This LISP-variable will be used by the inference cycle to check whether a rule has fired or not.

```
(defconcept PRODUCTION
  (meta-interpreter-type production-meta-interpreter)
  (condition (a boolean))
  (action (an form))
  ((try-to-fire
    (a subject
      (definition
        [form (when (>> referent should-fire)
                (setq *fires* t)
                (>> eval referent action))]))))))
```

The meta-interpreter of productions contains a special-instance-creator. It takes two elements to define a production. The first is a property-name or a logical combination of property-names. The (hairy) function Make-Lisp-Form-To-Test-Property (12) takes this property or this logical combination of properties as input and generates a KRS request for retrieving these properties from the memory and making the desired logical combination. This request is the referent of the definition of the condition-

subject. The second is the referent of its action (13).

```
(defconcept PRODUCTION-META-INTERPRETER
  (a meta-interpreter
    (special-instance-creator
      [function production-special-instance-creator])))

(defun PRODUCTION-SPECIAL-INSTANCE-CREATOR
       (type info ks::distance ks::krs-env ks::lisp-env)
  #~(a ~type
      (condition ~(make-form-to-test-property (first ~info)))
      (action [form ~(second info)])))


(defun MAKE-FORM-TO-TEST-PROPERTY (property)
  (cond
   ((atom property)
    `(>> (get ,property) memory))
   ((eq (first property) 'not)
    `(>> not ,@(cdr (make-form-to-test-property (second property)))))
   ((not (member (first property) '(and or)))
    (error "Unallowed logical condition ~S in production rule" (first property)))
   ((cddr property)
    `(>> (,(first property) ,(make-form-to-test-property (second property)))
         ,@(cdr (make-form-to-test-property (cons (first property)
                                                  (cddr property))))))
   (t (make-form-to-test-property (second property)))))


(defconcept TEST-PRODUCTION                                                    12
  [production adult-p
              (>> (print [string "John is an adult!"]) of message-window)])

-->  (>> definition handler condition of test-production)
<--  <form (>> (get adult-p) memory)>

-->  (>> action of test-production)
<--  <form (>> (print [string "John is an adult!"]) of message-window)>
```

Three kinds of productions are defined. An If-True is a production which should fire if its condition is True. An If-False is a production which should fire if its condition is False. An If-Unknown is a production which should fire if its condition is Unknown.

```
(defconcept IF-TRUE
  (a production
    (should-fire (>> (equal true) condition))))

(defconcept IF-FALSE
  (a production
    (should-fire (>> (equal false) condition))))

(defconcept IF-UNKNOWN
  (a production
    (should-fire (>> (equal unknown) condition))))
```

A Rule-Set is a sequence with as referent a list of Productions.

```
(defconcept RULE-SET
  (a sequence
    (element-type production)))
```

4.2.2. Modeling a Memory.

A memory contains boolean data. Data can be stored labeled with a given *property*, by which it can again be retrieved. The data is stored within an FPPD proposition, such that the consistency of the rules conditions is preserved when data is changed. We use for this the concept Primitive-Proposition, which has as referent an FPPD primitive-proposition. This concept is described in the manual (section 6.6).

The *for* of a memory is the problem whose associated rules operate upon this memory. The *get-proposition-subject* takes a property and returns the primitive-proposition which is associated with this property in the memory's referent. If this did not yet exist, it is added to it with a destructive operation. This way, from an external point of view, the memory behaves as if all properties which will ever be stored or asked are predefined. The subject *clear* resets all propositions in the memory, i.e. the data is set to Unknown. The subject *ask* queries for a value of a particular property and stores it in the memory. The subject *set* stores a given value for a given property and the subject *get* retrieves the value of a given property.

```
(defconcept MEMORY
  (a list
    (for (a problem))
    (definition
      [form (list '(*working-memory* ,(>> for)))])
    ((get-proposition
       (a subject
         (arguments [args ?property])
         (definition
           [form (get-proposition ?property (>> referent))])))))
    ((clear
       (an action-subject
         (definition
           [form (progn (clear-memory (>> referent)) (>> of done))])))))
    ((ask
      (a subject
        (arguments [args ?property])
        (definition
          [form (>> (set ?property (>> ask of ?property)))])))))
    ((get
      (a subject
        (arguments [args ?property])
        (definition
          [form (>> value (get-proposition ?property))])))))
    ((set
      (a subject
        (arguments [args ?property ?value])
        (definition
          [form (>> (redefine-value ?value) (get-proposition ?property))])))))))
```

```
(defun GET-PROPOSITION (property memory)
  (or (cdr (assoc property (cdr memory)))
      (let ((new (a primitive-proposition
                   (initial-value unknown))))
        (rplacd memory (cons (cons property new) (cdr memory)))
        new)))

(defun CLEAR-MEMORY (memory)
  (mapcar #'(lambda (one-pair)
              (>> (redefine-value unknown) of ~(cdr one-pair)))
          (cdr memory)))
```

4.2.3. Modeling a Problem.

A Problem groups a set of production-rules. Its *diagnose-subject* is an inference-subject, used to invoke forward chaining on the rule-set. Inference-subjects are described below. A problem's *conclusion-subject* takes a solution as input and sets this to the property Conclusion in the problem's memory. The *delegate-subject* delegates the diagnosis to another Problem. This is also the function of the subject *try*, except that this one takes over control again when the other problem did not derive an acceptable solution.

```
(defconcept PROBLEM
  (rules
   (a rule-set))
  ((diagnose
    (an inference-subject
      (domain (>>))))))
  ((conclusion
    (a subject
      (arguments [args ?solution])
      (definition
        [form (>> (set conclusion ?solution) memory)])))))
  ((delegate
    (a subject
      (arguments [args ?problem])
      (definition
        [form  (>> (set conclusion (>> diagnose of ?problem)) memory)])))))
  ((try
    (a subject
      (arguments [args ?problem ?unknown-propert,])
      (definition
        [form (clet ((?try (>> diagnose of ?problem)))
                (when (eq ?try (>> of no-problem))
                   (>> (redefine-value false)
                       (get-proposition ?unknown-property) memory)
                   (>> (set conclusion unknown) memory)
                   (format t "~&~%Returning to ~S :" (>>)))))])))))
```

4.2.4. The Inference Cycle.

An inference-subject starts a simple forward chaining loop on the production-rules of a given problem.

```
(defconcept INFERENCE-SUBJECT
  (a non-caching-subject
     (domain (a problem))
     (definition
       [form (forward-inference-engine (>> domain))])))
```

The function Forward-Inference-Engine contains two nested loops. The outer loop cycles until the property Conclusion in the memory is no longer Unknown. When it stops, it returns the value of this property.

The inner loop cycles through the fireable rules of the problem. These are all the rules whose condition is true. The cycle stops if one of the rules actually fired. A rule actually fires if the referent of its *try-to-fire* subject was not cached. The function sees this because the dynamically scoped variable *fires* is set. Hence, a rule which has fired before, and for which nothing has changed which affected its cached condition, does not fire again.

```
(defun FORWARD-INFERENCE-ENGINE (problem &aux rule-list)
  (clet ((?problem problem))
    (format t "~&~%Diagnosing ~S :" problem)
    (setq rule-list (>> referent rules of ?problem))
    (do ((conclusion (>> (get conclusion) memory of ?problem)
                     (>> (get conclusion) memory of ?problem)))
        ((not (eq conclusion (>> of unknown))) conclusion)
      (let ((fireable-rules
             (remove-if-not #'(lambda (one-rule)
                                (>> referent should-fire of ~one-rule))
                            rule-list)))
        (do ((x fireable-rules (cdr x))
             (*fires* nil))
            ((null x)
             (krs-error "None of the rules in ~S can fire !" problem))
          (declare (special *fires*))
          (>> try-to-fire of ~(first x))
          (when *fires* (return)))))))
```

A Malfunctioning-Object has a Problem and a diagnose-subject. When the filler of the latter subject is requested, it invokes the diagnosis of the problem. When it is finished, it clears the problem's memory. This way, all cached values depending on the specific previous case are withdrawn, and will not disturb the correct diagnoses of later cases.

```
(defconcept MALFUNCTIONING-OBJECT
  (problem (a problem))
  ((diagnose
    (a subject
      (definition
        [form (unwind-protect (>> diagnose problem)
                (>> clear memory problem))]))))))
```

4.2.5. Auxiliary Concepts.

```
(defconcept PROPERTY
  (to-query (a string))
  ((ask (a non-caching-subject
          (definition [form  (let ((answer (y-or-n-p (>> referent to-query))))
                               (if answer (>> of true) (>> of false)))]))))))


(defconcept CONCLUSION
  (a property))


(defconcept SOLUTION)


(defconcept NO-PROBLEM
  (a solution))
```

4.3. A Production Program for Technical Diagnosis of a Car.

4.3.1. The Problems.

A malfunctioning-car is a malfunctioning-object with as problem the concept Car-Problem.

```
(defconcept MALFUNCTIONING-CAR
  (a malfunctioning-object
    (problem car-problem)))
```

Car-Problem-Memory is the memory of Car-Problem.

```
(defconcept CAR-PROBLEM-MEMORY
  (a memory
    (for car-problem)))
```

The rules associated to Car-Problem are used to focus into a particular sub-problem. They can be read as follows:

- If there is no problem with the car then conclude No-Problem.
- If it is not known whether there is a problem then ask.
- If the problem is with starting, then focus on Start-Problem.
- If it is not known whether there is a start problem then ask.
- If the problem concerns the electrical installation, then focus on Electricity-Problem.
- If it is not known whether the problem concerns the electrical installation than ask.

```
(defconcept CAR-PROBLEM
  (a problem
    (memory car-problem-memory)
    (rules
      [rule-set
        [if-false car-problem-p
          (>> (conclusion no-problem))]
        [if-unknown car-problem-p
          (>> (ask car-problem-p) memory)]
        [if-true start-problem-p
          (>> (delegate start-problem))]
        [if-unknown start-problem-p
          (>> (ask start-problem-p) memory)]
        [if-true electricity-problem-p
          (>> (delegate electricity-problem))]
        [if-unknown electricity-problem-p
          (>> (ask electricity-problem-p) memory)]])))
```

The rules associated to Start-Problem figure out a remedy for a car which does not start. Besides the typical focus and query rules we had in the previous rule-set, there are also a few more interesting ones. For example the fourth rule states that if the lights can not burn, the start-problem is in fact caused by an electricity-problem. Rule seven and eight successively try out sub-problems. Rule seven tries whether it is a problem with the starter. Rule eight tries whether it is an ignition-problem.

```
(defconcept START-PROBLEM
  (a car-problem
    (memory car-problem-memory)
    (rules
      [rule-set
        [if-false start-problem-p
          (>> (conclusion no-problem))]
        [if-unknown start-problem-p
          (>> (ask start-problem-p) memory)]
        [if-true electricity-problem-p
          (>> (delegate electricity-problem))]
        [if-false lights-can-burn-p
          (>> (set electricity-problem-p true) memory)]
        [if-true lights-can-burn-p
          (>> (set electricity-problem-p false) memory)]
        [if-unknown lights-can-burn-p
          (>> (ask lights-can-burn-p) memory)]
        [if-unknown starter-problem-p
          (>> (try starter-problem starter-problem-p))]
        [if-unknown ignition-problem-p
          (>> (try ignition-problem ignition-problem-p))]])))
```

Rules associated to Electricity-Problem try to remedy problems with a car's electrical installation. A strange rule here is rule three. It entails a typical heuristic rule, a rule which shortcuts a large part of the search space (and which is not necessarily always valid). It states that if two problems occur, i.e. the car does not start and there is an electricity-problem, then the remedy is to charge the battery.

```
(defconcept ELECTRICITY-PROBLEM
  (a car-problem
    (memory car-problem-memory)
    (rules
      [rule-set
        [if-false electricity-problem-p
          (>> (conclusion no-problem))]
        [if-unknown electricity-problem-p
          (>> (ask electricity-problem-p) memory)]
        [if-true start-problem-p
          (>> (conclusion charge-battery))]
        [if-true lights-can-burn-p
          (>> (conclusion no-problem))]
        [if-unknown lights-can-burn-p
          (>> (ask lights-can-burn-p) memory)]
        [if-unknown start-problem-p
          (>> (ask start-problem-p) memory)]
        [if-false lamps-ok-p
          (>> (conclusion repair-lamps))]
        [if-unknown lamps-ok-p
          (>> (ask lamps-ok-p) memory)]
        [if-false fuses-ok-p
          (>> (conclusion repair-fuses))]
        [if-unknown fuses-ok-p
          (>> (ask fuses-ok-p) memory)]
        [if-true fuses-ok-p
          (>> (conclusion check-wires))]])))
```

Rules associated to Starter-Problem remedy problems with the car's start-engine.

```
(defconcept STARTER-PROBLEM
  (a car-problem
    (memory car-problem-memory)
    (rules
      [rule-set
        [if-false starter-problem-p
          (>> (conclusion no-problem))]
        [if-true starter-turning-p
          (>> (ask starter-transmission-ok-p) memory)]
        [if-false starter-transmission-ok-p
          (>> (conclusion repair-starter-transmission))]
        [if-true starter-transmission-ok-p
          (>> (conclusion no-problem))]
        [if-true starter-turning-when-hit-p
          (>> (conclusion repair-starter))]
        [if-unknown electricity-problem-p
          (>> (ask electricity-problem-p) memory)]
        [if-true electricity-problem-p
          (>> (delegate electricity-problem))]
        [if-false starter-powered-p
          (>> (conclusion check-starter-connection))]
        [if-unknown starter-turning-p
          (>> (ask starter-turning-p) memory)]
        [if-unknown starter-turning-when-hit-p
          (>> (ask starter-turning-when-hit-p) memory)]
        [if-unknown starter-powered-p
          (>> (ask starter-powered-p) memory)]
        [if-unknown starter-problem-p
          (>> (conclusion no-problem) memory)]])))
```

Rules associated to Ignition-Problem diagnose problems with a car's ignition.

```
(defconcept IGNITION-PROBLEM
  (a car-problem
    (memory car-problem-memory)
    (rules
      [rule-set
        [if-false ignition-problem-p
          (>> (conclusion no-problem))]
        [if-unknown electricity-problem-p
          (>> (ask electricity-problem-p) memory)]
        [if-true electricity-problem-p
          (>> (delegate electricity-problem))]
        [if-unknown plug-1-powered-p
          (>> (ask plug-1-powered-p) memory)]
        [if-unknown plug-2-powered-p
          (>> (ask plug-2-powered-p) memory)]
        [if-false (or plug-1-powered-p plug-2-powered-p)
          (>> (set no-plugs-powered-p true) memory)]
        [if-true (and plug-1-powered-p plug-2-powered-p)
          (>> (set both-plugs-powered-p true) memory)]
        [if-true no-plugs-powered-p
          (>> (ask ignition-coil-powered-p) memory)]
        [if-false ignition-coil-powered-p
          (>> (conclusion check-ignition-coil-connection))]
        [if-true ignition-coil-powered-p
          (>> (conclusion repair-ignition-coil))]
        [if-true both-plugs-powered-p
          (>> (conclusion repair-plugs))]
        [if-false plug-1-powered-p
          (>> (conclusion repair-cable-1))]
        [if-false plug-2-powered-p
          (>> (conclusion repair-cable-2))]])))
```

4.3.2. Auxiliary Concepts.

SOLUTIONS

```
(defconcept CAR-PROBLEM-SOLUTION
 (a solution))

(defconcept ELECTRICITY-PROBLEM-SOLUTION
 (a car-problem-solution))

(defconcept CHARGE-BATTERY
 (an electricity-problem-solution))

(defconcept REPAIR-LAMPS
 (an electricity-problem-solution))

(defconcept REPAIR-FUSES
 (an electricity-problem-solution))

(defconcept CHECK-WIRES
 (an electricity-problem-solution))
```

```
(defconcept STARTER-PROBLEM-SOLUTION
 (a car-problem-solution))

(defconcept REPAIR-STARTER-TRANSMISSION
 (a starter-problem-solution))

(defconcept REPAIR-STARTER
 (a starter-problem-solution))

(defconcept CHECK-STARTER-CONNECTION
 (a starter-problem-solution))

(defconcept IGNITION-PROBLEM-SOLUTION
 (a car-problem-solution))

(defconcept REPAIR-CABLE-1
 (an ignition-problem-solution))

(defconcept REPAIR-CABLE-2
 (an ignition-problem-solution))

(defconcept REPAIR-PLUGS
 (an ignition-problem-solution))

(defconcept CHECK-IGNITION-COIL-CONNECTION
 (an ignition-problem-solution))

(defconcept REPAIR-IGNITION-COIL
 (an ignition-problem-solution))
```

PROPERTIES

```
(defconcept CAR-PROBLEM-PROPERTY
 (a property))

(defconcept CAR-PROBLEM-P
 (a car-problem-property
  (to-query [string "Is there a problem with the car ? "])))

(defconcept START-PROBLEM-PROPERTY
 (a car-problem-property))

(defconcept START-PROBLEM-P
 (a start-problem-property
  (to-query [string "Have you problems to start the car ? "])))

(defconcept ELECTRICITY-PROBLEM-PROPERTY
 (a car-problem-property))
```

```
(defconcept ELECTRICITY-PROBLEM-P
 (an electricity-problem-property
  (to-query [string "Is there a problem with the electrical installation ? "])))

(defconcept LIGHTS-CAN-BURN-P
 (an electricity-problem-property
  (to-query [string "Can the lights burn ? "])))

(defconcept LAMPS-OK-P
 (an electricity-problem-property
  (to-query [string "Are the lamps ok ? "])))

(defconcept FUSES-OK-P
 (an electricity-problem-property
  (to-query [string "Are the fuses ok ? "])))

(defconcept STARTER-PROBLEM-PROPERTY
 (a car-problem-property))

(defconcept STARTER-PROBLEM-P
 (an starter-problem-property
  (to-query [string "Is there a problem with the starter ? "])))

(defconcept STARTER-TURNING-P
 (an starter-problem-property
  (to-query [string "Is the starter turning ? "])))

(defconcept STARTER-TRANSMISSION-OK-P
 (an starter-problem-property
  (to-query [string "Is the starter-transmission ok ? "])))

(defconcept STARTER-TURNING-WHEN-HIT-P
 (an starter-problem-property
  (to-query [string "Is the starter turning after hitting it ? "])))

(defconcept STARTER-POWERED-P
 (an starter-problem-property
  (to-query [string "Is the starter powered ? "])))

(defconcept IGNITION-PROBLEM-PROPERTY
 (a car-problem-property))

(defconcept IGNITION-PROBLEM-P
 (an ignition-problem-property
  (to-query [string "Is there an ignition-problem ? "])))

(defconcept PLUG-1-POWERED-P
 (an electricity-problem-property
  (to-query [string "Is plug 1 powered ? "])))

(defconcept PLUG-2-POWERED-P
 (an electricity-problem-property
  (to-query [string "Is plug 2 powered ? "])))
```

```
(defconcept IGNITION-COIL-POWERED-P
 (an electricity-problem-property
  (to-query [string "Is the ignition-coil powered ? "])))

(defconcept BOTH-PLUGS-POWERED-P
 (an electricity-problem-property
  (to-query [string "Are both plugs powered ? "])))

(defconcept NO-PLUGS-POWERED-P
 (an electricity-problem-property
  (to-query [string "Are there no plugs powered ? "])))
```

4.4. A Trace of the Program.

We now show some traces of the car diagnosing. (13) illustrates the simple focusing mechanism. The inference mechanism starts with the Car-Problem. The questions asked are to focus on a specific sub-problem. In the case, it focuses on Electricity-Problem, where it comes to a solution.

```
? (>> diagnose of malfunctioning-car))                                    13

Diagnosing <CAR-PROBLEM> :
Is there a problem with the car ?  (y or n)  y
Have you problems to start the car ?  (y or n)  n
Is there a problem with the electrical installation ?  (y or n)  y

Diagnosing <ELECTRICITY-PROBLEM> :
Can the lights burn ?  (y or n)  n
Are the lamps ok ?  (y or n)  y
Are the fuses ok ?  (y or n)  n
<REPAIR-FUSES>
```

(14) demonstrates focusing two levels. It first focuses to Start-Problem. The question it asks is to find out whether the start-problem could be caused by an electricity-problem. Since this is not the case, it focuses one level deeper to Starter-Problem, where it can deduce a solution.

```
? (>> diagnose of malfunctioning-car))                                    14

Diagnosing <CAR-PROBLEM> :
Is there a problem with the car ?  (y or n)  y
Have you problems to start the car ?  (y or n)  y

Diagnosing <START-PROBLEM> :
Can the lights burn ?  (y or n)  y

Diagnosing <STARTER-PROBLEM> :
Is the starter turning ?  (y or n)  y
Is the starter-transmission ok ?  (y or n)  n
<REPAIR-STARTER-TRANSMISSION>
```

In (15), the mechanism had focussed to the wrong problem: Start-Problem. When it finds out that the problem was actually caused by an Electricity-Problem it changes

its focus. While diagnosing Electricity-Problem, it finds that, since there are both problems to start the car and to switch on the lights, the solution must be to charge the battery.

```
? (>> diagnose of malfunctioning-car))                                    15

Diagnosing <CAR-PROBLEM> :
Is there a problem with the car ?  (y or n)  y
Have you problems to start the car ?  (y or n)  y

Diagnosing <START-PROBLEM> :
Can the lights burn ?  (y or n)  n

Diagnosing <ELECTRICITY-PROBLEM> :
<CHARGE-BATTERY>
```

(16) finally shows how the mechanism can successively try different hypothesis. While diagnosing Start-Problem, it had first attempted to focus on Start-Problem. Since however no acceptable solution was derived there, control is returned to Start-problem. Then it makes a second hypothesis by focusing on Ignition-Problem. Now a solution is derived.

```
? (>> diagnose of malfunctioning-car))                                    16

Diagnosing <CAR-PROBLEM> :
Is there a problem with the car ?  (y or n)  y
Have you problems to start the car ?  (y or n)  y

Diagnosing <START-PROBLEM> :
Can the lights burn ?  (y or n)  y

Diagnosing <STARTER-PROBLEM> :
Is the starter turning ?  (y or n)  y
Is the starter-transmission ok ?  (y or n)  y

Returning to <START-PROBLEM> :

Diagnosing <IGNITION-PROBLEM> :
Is plug 1 powered ?  (y or n)  n
Is plug 2 powered ?  (y or n)  n
Is the ignition-coil powered ?  (y or n)  n
<CHECK-IGNITION-COIL-CONNECTION>
```

4.5. Conclusion.

The construction of a simple production-system on top of the KRS concept-graph was demonstrated. It must serve as an example of integrating KRS with a traditional knowledge representation formalism. We hereby followed the following strategy.

1 All objects playing a role in the representation and the inference mechanisms are modeled explicitly. In this case, the representation objects are the production rules and the problems, and the inference objects are for example memories.

2 A LISP-function is written to implement the basic inference cycle. This function accesses the objects defined in the previous step as much as possible via

their subjects. This way, it can benefit from many cached results.

2 This LISP-function is called in the definition of a subject. This way, the inference can be invoked by a normal KRS request. The advantage of this is that other parts of the concept-graph can just use this request without caring about how the result is achieved.

3 To exploit KRS's consistency mechanism one has to think which mutations will be important to withdraw particular cached results. In this example, this are the mutations within a memory. There are several ways to make those mutations invoke consistency maintenance. One way is to make the mutations by changing existing concepts. Another way is to mutate existing concepts by either defsubjects or mutating-subjects. In here we used a third method. Data was mutated by explicitly redefining the values of FPPD primitive propositions, which were encapsulated by primitive-proposition concepts.

In the example we gave, the conditions and actions only used data from the inference's memory. Notice however that they can as well access any other knowledge represented in KRS, for example knowledge represented in the classification hierarchy given in example two. This is one of the major advantages of modeling the production system on top of the KRS concept-graph.

5. CONCLUSION.

Four extended KRS programs were described in detail: The Road-Map, Classification, Demons and the Small Production-System. They each illustrated a different KRS programming skill. They are summerized below.

DATA MODELING

Straightforward data modeling is the most common KRS programming style. There is a direct mapping between the entities in the representation domain and their representation in the concept-graph.

EXPLICIT DATA RETRIEVAL ARCHITECTURE

An explicit data-retrieval architecture is a sub-network of the concept graph via which concepts can deduce more data. It is analogue (but probably more complex) then inheritance which is the standard data-retrieval mechanism.

DATA DRIVEN PROGRAMMING

Data driven programming constitutes a method to undertake an action the moment an event occurs. It is typically useful in forward reasoning processes. It is complementary to the normal KRS reasoning processes because those are backward.

KNOWLEDGE REPRESENTATION FORMALISMS

By implementing traditional knowledge representation formalisms on top of the KRS concept-graph it is possible to use several of them together and to tailor them for private usage.

The four styles occurring in these examples require increasing KRS programming knowledge. Representation of static knowledge as in the first example is a technique that novice KRS users can apprehend very fast. The last technique however requires a thorough understanding of the KRS mechanism in general and of the consistency mechanism in particular.

By observing the strategies used to set up each of the examples, we deduce the following rules of thumb.

- By using the same concepts many times, one benefits from cached referents. The caching mechanism is less effective is one makes many new instances.[14]
- Using named concepts is a technique to re-use the same concepts many times. Named concepts are indeed easy to access as much as one wants.
- Deduction mechanism are typically implemented by a LISP-function. They can also be implemented using only the KRS concept-graph. The special interface between KRS and LISP however make this method easier for most cases.

[14] Memorizing-Subjects (described in the manual section 6.4) are developed to make it possible to re-use the same subject-fillers many times for subjects with arguments which do normally not cache their filler.

- The LISP-function which implements the reasoning process is typically called from within a definition. This gives a modular interface to other KRS programs and allows the KRS interpreter to cache the deductions made.

Another important technique which was not directly illustrated with this series of examples is the use of definitions to call on other applications implemented in LISP. Our special way of interfacing with LISP implies that any program written in LISP can be easily incorporated. Previous examples have in this way used the mail-system, LM-Prolog [Kahn83] and recently a program to steer a robot arm.

CONCLUSION.

This work was written to meet two fundamental objectives: (i) the communication of KRS know-how and (ii) the clarification of the techniques which were developed for the KRS implementation.

The communication of KRS know-how, involving the KRS definition, its implementation and the way it is used, is meant to support KRS users. KRS is currently used inside and outside the laboratory for research in various domains. This work showed how KRS can be applied in several domains and by people with different levels of KRS expertise.

This work also reported on the techniques which were used in the KRS implementation. It may inspire others to use the same or similar techniques in different areas. Especially FPPD and the Task-Processor can in our view be directly applied for many other purposes. It may also help people who want to extend KRS or people who want to build new representation languages based on KRS experiences to see where the traps are hidden and how we solved them.

THE KRS PROJECT

KRS is a representation language which enhances the computational capacities of the underlying language LISP with better data-modeling features. The KRS project has been central in all early research activities within our laboratory, and has therefore also led to the most concrete product. The importance of the KRS project is demonstrated by the following observations:

- KRS is the most important outcome of the Esprit project P440: "Message Passing Architectures and Description Systems", which is considered as one of the more successful projects within Esprit.
- KRS has been the backbone of many research projects inside and outside our laboratory. Inside our laboratory, it has been used in all large projects done so far. Outside, KRS is for example used at Bull Transac (Paris), Siemens AG (Muenich), Institute Mario Negri (Milan), etc.
- Many doctoral and bachelor's theses have either used or concerned KRS.

KRS SITUATED

KRS is inspired by intensional logic and by representation language languages (ARLO in particular). It is a representation language defined on top of the programming language LISP. A representation language provides additional features for the underlying programming language, to better support A.I. programming. These features have to do with representation, deduction and control aspects.

The primitive for representation in KRS is a large network of concepts, called the concept-graph. The concept-graph is fully reflective, i.e. each component of the graph (each node and each link) is explicitly represented by a different and separate node in that same graph. The concept-graph is defined and inspected by descriptions in a concept-language. The concept-language is lazyly interpreted and lexically scoped to enhance the expressiveness of the descriptions.

KRS incorporates two deductive mechanisms. There is a simple single inheritance mechanism. Its main power originates from its relation to lexical scope. Referent computation is a mechanism to compute a concept's referent from its definition. It provides a way to automatically deduce what a concept is about.

Control in KRS is mostly hierarchical. The control-flow is however largely influenced by the caching and consistency maintenance mechanism, which frees the KRS user from worrying about minimizing the number of times a complex computation is performed.

IMPLEMENTATION TECHNIQUES

FPPD is a data-dependency system which, although it was specially developed for the implementation of KRS, can be applied in many different applications. The basic function of FPPD is to use data-dependencies, which it detects itself, to keep track of results of computations such that they can be re-used as long as they remain valid.

The task-processor implements an inherently recursive process in a continuation based programming style, resulting in an efficient and flexible implementation.

The problems with the inheritance implementation are mainly caused by the lexical scope mechanism. Our solution computes the scope of inherited descriptions by matching patterns of concepts. It is hard to separate this solution from the KRS inheritance and lexical scope mechanisms. However, we feel these two mechanism are translatable to other object-oriented languages and if this happens, their implementation is obviously a relevant (and far from trivial) aspect.

KRS USAGE

The KRS usage was illustrated with four extended examples, each illustrating a specific KRS programming style:

Data Modeling: Straightforward data modeling consists of defining a direct mapping between the entities in the representation domain and their representation in the concept-graph.

Explicit Data Retrieval Architecture: The second example showed how to use KRS for explicitly modeling data-retrieval structures, such as a classification tree.

Data Driven Programming: The third example discussed an innovative strategy for demon triggering, i.e. through FPPD's dependency network.

Knowledge Representation Formalisms: We also showed how traditional knowledge representation formalisms can be implemented in KRS. This example also shows how the user can explicitly exploit FPPD's consistency maintenance behavior.

By the observation of the different examples, we have extracted a few rules of thumb, which can serve as the backbone of a strategy for KRS programming. These rules are listed in the conclusion of chapter three.

FUTURE WORK

The KRS developments can be continued in many ways. Some of these are merely software engineering jobs, others require more or different research to be done.

- KRS currently lacks a dedicated debugger. The primary difficulty in implementing such a debugger is to decide how it must look like. It requires more KRS programming expertise to find out what kind of errors mostly occur, what they are caused by, and how a debugger can support the debugging phase.
- The KRS interface runs at present only on Symbolics™ LISP-machines. It should be rewritten such that it is better portable to different machines. Studies in that direction have been made in the context of Esprit project P440.
- It can be investigated whether the incorporation of different features within the KRS kernel can enhance representation or deduction capacities. Possible candidates are a matcher or a built-in notion of relations (= two-way subjects).
- KRS fundamentally supports symbolic knowledge representation. The relation with currently emerging knowledge representation developments such as connectionism or complex dynamics can be investigated. New KRS deduction mechanism can possibly interact with subsymbolic mechanisms of this kind.
- The possibility to adapt the KRS interpreter to become a full reflective interpreter as proposed in Pattie Maes's doctoral thesis is currently investigated. It appears that this is possible to some extent, but not in a systematic and robust way. There are two reasons for this. First there are many interdependencies between all KRS features. Second the KRS interpreter was not written with a full

reflective version in mind. Research along this line could be continued either by reimplementing KRS directly with reflection in mind or by making a weaker KRS design which keeps all features more independent from each other.

References

Abbott87.

Abbott, R.J., "Knowledge Abstraction.," *Communications of the ACM*, vol. 30, no. 8, August 1987.

America87.

America, P., "Inheritance and Subtyping in a Parallel Object-Oriented Language.," *Proc. of ECOOP'87: European Conference on Object-Oriented Programming*, Paris, France, June 1987.

Bobrow77.

Bobrow, D.G. and Winograd, T., "An overview of KRL, a Knowledge Representation Language," *Cognitive Science*, vol. 1, pp. 3-46, 1977.

Bobrow81.

Bobrow, D.G. and Stefik, M., *The LOOPS Manual*, Palo Alto, 1981. Working Paper KB-VLSI-81-13 Xerox Parc

Borning86.

Borning, A., "Classes versus Prototypes in Object-Oriented Languages.," *Proceedings of the ACM / IEEE Fall Joint Computer Conference*, Dallas, TX, November, 1986.

Bowen84.

Bowen, K.A., "Expert systems in metaPROLOG.," *unpublished manuscript*, 1984.

Brachman85a.

Brachman, R.J. and Smolze, J.G., "An overview of the KL-ONE knowledge representation system," *Cognitive Science*, vol. 9(2), pp. 171-216, 1985.

Brachman85b.

Brachman, R.J., "I lied about the trees," *AI Magazine*, pp. 80-93, Fall 1985.

Briot86.

Briot, J.P. and Cointe, P., "The OBJVLISP Project: definition of a uniform self-described and extensible object oriented language," *Proc. 7th European Conference on Artificial Intelligence (ECAI-7)*, Brighton, UK, 1986.

Brownston85.

Brownston, L., Farrell, R., Kant, E., and Martin, N., *Programming Expert Systems in OPS5*, Addison Wesley , Massachusettes, 1985.

Byers87.

Byers, G. et. al., *Allegro Common Lisp for the Macintosh*, 1987.

Carnap56.

Carnap, R., *Meaning and Necessity.*, University of Chicago Press, Chicago, 1956.

Charniak80.

Charniak, E., Riesbeck, C. K., and McDermott, D. V., *Artificial Intelligence*

Programming., Lawrence Erlbaum Associates, Publishers, Hillsdale, New-Jersey, 1980.

Charniak87.

Charniak, E., Riesbeck, C. K., McDermott, D. V., and Meehan J.R., *Artificial Intelligence Programming. (second edition)*, Lawrence Erlbaum Associates, Publishers, Hillsdale, New-Jersey, 1987.

Clayton85.

Clayton, B.D., "ART: programming primer," *Inference Corporation*, Los Angeles, 1985.

Cointe86.

Cointe, P., "The ObjVlisp Kernal: a Reflexive Lisp Architecture to define a Uniform Object Oriented System," *Proc. first workshop on Reflection and Meta-level Architectures*, Alghero, Italy, 1986.

Cook88.

Cook, S., Van Marcke, K., Moss, E., Loomis, M., Nieratrasz, O., Kransner, G., and Brown, P., "Workshop on Object-Oriented-Programming. ECOOP 1987, Paris, June 18, 1987.," *Sigplan Notices*, vol. 23, no. 1, January 1988.

Daelemans87.

Daelemans, W., "Studies in Language Technology. An Object-Oriented Computer Model of Morphonological Aspects of Dutch," *Doctoral Thesis*, 1987.

Data86.

Neuron Data, inc., *NEXPERT object*, 1986.

de Kleer77.

de Kleer, J., Doyle, J., Steele, G.L., and Sussman, G.J., "AMORD: Explicit control of reasoning.," *Proc. Symp. on AI and Programming Languages (SIGART Newsletter No. 64)*, 1977.

de Kleer86a.

de Kleer, J., "An Assumption-Based TMS," *Artificial Intelligence Journal*, vol. 28, no. 2, North Holland, Amsterdam, 1986.

de Kleer86b.

de Kleer, J., "Problem solving with the ATMS," *Artificial Intelligence Journal*, vol. 28, no. 2, North Holland, Amsterdam, 1986.

Di Primio85.

Di Primio, F. and Brewka, G., "BABYLON - Kernel system of an integrated environment for expert system development & operation," *Proc. 5th International Workshop on Expert Systems and their Applications*, Avignon, France, 1985.

Doyle79.

Doyle, J., "A Truth Maintenance System," *Artificial Intelligence Journal*, vol. 12, pp. 231-272, North Holland, Amsterdam, 1979.

Fahlman.

Fahlman, S.E., *NETL: A system for representing and Using Real-World Knowledge.*, The MIT Press, Cambridge, MA, 1979

Feigenbaum71.

Feigenbaum, E., Buchanan, B., and Lederberg, J., "On Generality and Problem Solving: A case study using the DENDRAL program.," in *Machine Intelligence 6*, ed. B. Meltzer & D. Michie., pp. 165-190., American Elsevier., 1971.

Ferber84.

Ferber, J., "Mering: An Open-Ended Object Oriented Language for Knowledge Representation.," *Proc. 6th European Conference on Artificial Intelligence (ECAI-6)*, Pisa, Italy, 1984.

Furukawa84.

Furukawa, K., Takeuchi, A., Kunifuji, S., Yasukawa, H., Ohki, M., and Ueda, K., "Mandala: A Logic Based Knowledge programming system.," *In Proc. International conference on Fifth Generation Computer Systems.*, 1984.

Gallaire82.

Gallaire, H. and Lasserre, C., "A Control Metalanguage for Logic Programming.," in *Logic Programming*, ed. K.L. Clark & S.-A. Tarnlund, Acedemic Press, 1982.

Goldberg83.

Goldberg, A. and Robson, D., *Smalltalk-80: The Language and its Implementation.*, Addison-Wesley, 1983.

Greiner80.

Greiner, R., *RLL-1: A Representation Language Language*, Stanford, CA, 1980. Working Paper HPP-80-9 Stanford University

Haase86.

Haase, K., "ARLO - Another Representation Language Offer," *MIT Bachelor's Thesis*, October 1986.

Hewitt76.

Hewitt, C., *Viewing Control Structures as Patterns of Passing Messages*, Cambridge, MA, December 1976.

Hewitt80.

Hewitt, C., Attardi, G., and Simi, M., "Knowledge Embedding in the Description System Omega.," *Proc. of the first AAAI conference*, Stanford, 1980.

Hillis85.

Hillis, D., *The Connection Machine,* MIT Press, 1985.

Intellicorp84.

Intellicorp, *KEE, Software Development Systems. User's Manual*, 1984. Intellicorp document: 2.0-UZ-1

Jonckers87.

Jonckers, V., "A Framework for modeling Programming Knowledge," *VUB AI-Lab*, 1987. Doctoral Thesis

Kahn83.

Kahn, K.M. and Carlsson, M., "How to implement PROLOG on a LISP Machine.," *Uppsala Programming Methodology and Artificial Intelligence Laboratory.*, 1983.

Kauffmann86.

Kauffmann, H. and Grumbach, A., "MULTILOG: MULTIple worlds in LOGic programming.," *Report*, 1986.

Lenat76.

Lenat, D.B., *AM: An Artificial Intelligence Approach to Discovery in Mathematics as Heuristic Search.*, Stanford, CA, July 1976. STAN-CS-76-814 (HPP-80-17)

Lieberman86a.

Lieberman, H., "Using Prototypical Objects to Implement Shared Behavior in Object Oriented Systems," *Proc. of OOPSLA '86*, Portland. Oregon., 1986.

Lieberman86b.

Lieberman, H, "Delegation and Inheritance, two modular mechanisms," *3rd Workshop on OOP*, Paris - Centre Georges Pompidou, 1986.

Lindsay80.

Lindsay, R., Buchanan, B.G., Feigenbaum, E.A., and Lederberg, J., *DENDRAL.*, McGraw-Hill, New YUork, 1980.

Machines86.

Thinking Machines, "The Essential *Lisp Manual," *Thinking Machines TR 86.15*, June, 1986.

Maes87.

Maes, P., "Computational Reflection," *VUB AI-Lab TR 87-2*, 1987. Doctoral Thesis

McDermott83.

McDermott, D.V., "Contexts and Data Dependencies: A Synthesis," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. PAMI-5, no. 3, May 1983.

Moon86.

Moon, D.A., "Object-Oriented Programming with Flavors.," *Proc. of OOPSLA '86*, Portland. Oregon., 1986.

Pereira82.

Pereira, L.M., "Logic Control with Logic.," *Proc. First International Logic Programming Conference*, 1982.

Rees86.

Rees, J.A. and Clinger, W., "The Revised Report on the Algorithmic Language Scheme.," *Sigplan Notices 21*, December 1986.

Rich78.

Rich, C., "Initial Report on a LISP Programmer's Apprentice.," *IEEE Transactions on Software Engineering.* , vol. 5, no. 6, 1978.

Roberts77.

Roberts, R.B. and Goldstein, I.P., "The FRL Manual," *MIT AI Memo No 409*, Cambridge, MA, 1977.

Shapiro83.

Shapiro, E., "The Bagel: A systolic concurrent prolog machine," *The weizmann Institut of Science*, Rehovot, Israel, 1983.

Shapiro87.

Shapiro, E., *Encyclopedia of Artificial Intelligence.*, Wiley-Interscience, 1987.

Stallman77.

Stallman, R.M. and Sussman, G.J., "Forward Reasoning and dependency-directed backtracking in a system for computed aided circuit analyses.," *Artificial Intelligence Journal*, vol. 9, no. 2, North Holland , Amsterdam, 1977.

Steele84.

Steele, G., *Common LISP: The Language.*, Digital Press, 1984.

Steels84.

Steels, L., "Object Oriented Knowledge Representation in KRS," *Proc. 6th European Conference on Artificial Intelligence (ECAI-6)*, Pisa, Italy, 1984.

Steels85a.

Steels, L., "Second Generation Expert Systems.," *Future Generation Computer Systems*, vol. 1, no. 4, North-Holland Pub. , Amsterdam, 1985.

Steels85b.

Steels, L., "Algorithmic Concepts in KRS.," *Proc. NGI-SION symposium*, 1985.

Steels86.

Steels, L., "The KRS concept system," *VUB AI-Lab Technical Report 86-1*, 1986.

Steels87.

Steels, L., "Artificial Intelligence and Complex Dynamics.," *presented at IFIP workshop on tools for knowledge-based systems.*, Mount Fuji, Japan., November, 1987.

Steels88.

Steels, L., "The Explicit Representation of Meaning.," in *Meta-Level Architectures and Reflection*, ed. P. Maes & D. Nardi, Norh-Holland , February, 1988.

Stein87.
Stein, L.A., "Delegation Is Inheritance.," *Proc. of OOPSLA '87*, Orlando, Florida, 1987.

Sterling84.
Sterling, L., "Expert System = Knowledge + Meta-Interpreter.," *Tech. Rep. CS84-17. Weizmann Institute of Science*, Rehovot, Israel, 1984.

Strickx87.
Strickx, P., "Extrak: A Second Generation Expert System in KRS.," *VUB AI-Memo 87-7*, 1987.

Sussman71.
Sussman, G.J., Winograd. T., and Charniak, E., "Micro-Planner reference manual.," *MIT AI-memo 203 A*, Cambridge, MA, 1971.

Sussman72.
Sussman, G.J. and McDermott, D.V., "Why Conniving is better than planning.," *MIT AI-memo 255 A*, Cambridge, MA, 1972.

Sussman75.
Sussman, G.J. and Steele, G, "Scheme: an Interpreter for Extended Lambda Calculus.," *MIT AI-memo 349*, Cambridge, MA, December, 1975.

Symbolics87.
Symbolics, *GENERA Common-Lisp Manual*, 1987.

Touretzky86.
Touretzky, D.S., *The Mathematics of Inheritance Systems,* Pitman, London, 1986.

Ungar87.
Ungar, D. and Smith, R.B., "Self: The Power of Simplicity.," *Proc. of OOPSLA '87*, Orlando, Florida, 1987.

Van de Velde88.
Van de Velde, W., "Learning from Experience.," *VUB AI-Lab*, 1988. Doctoral Thesis

Van Marcke86a.
Van Marcke, K., "A Parallel Algorithm for Consistency Maintenance in Knowledge Representation," *Proc. 7th European Conference on Artificial Intelligence (ECAI-7)*, Brighton, UK, 1986.

Van Marcke86b.
Van Marcke, K., "FPPD: A Consistency Maintenance System based on Forward Propagation of Proposition Denials," *VUB AI-Memo 86-1*, 1986.

Van Marcke87a.
Van Marcke, K., "Context Determination through Inheritance in KRS," *VUB AI-Memo 87-1*, 1987.

Van Marcke87b.
Van Marcke, K., Jonckers, V., and Daelemans, W., "Representation Aspects of Knowledge-Based Office Systems," *ESPRIT technical week.*, 1987.

Van Marcke88a.
Van Marcke, K., "KRS: An Object-Oriented Representation Language.," *Revue d'Intelligence Artificielle*, vol. 1, no. 4, 1988.

Van Marcke88b.
Van Marcke, K., "Towards Explicit Inheritance Schemes," *Proceedings of Hawaii International Conference On System Sciences.*, Januari, 1988.

Van Melle81.
Van Melle, W., Scott, A.C., Bernett, J.S., and Peairs, M., *The Emycin Manual*, Stanford, CA, 1981. Report STAN-CS-81-885 Stanford University

Waters81.
Waters, R.C., "The Programmer's Apprentice: A Session with KBEmacs.," *IEEE Transactions on Software Engineering.* , vol. SE-11, no. 11, 1981.

Weinreb81.
Weinreb, D, and Moon, D., "The Lisp-Machine Manual," *Symbolics Inc.*, 1981.

Yuasa84.
Yuasa, T. et al., "Kyoto Common Lisp Report.," *Research Institute for Mathematical Sciences, Kyoto Univerity*, 1984.

Zeigler86.
Zeigler, B.P. and De Wael, L., "Towards a knowledge based implementation of multifacetted modeling methodology.," in *Artificial Intelligence in Simulation: State of the Art.*, ed. Van Steenkiste, Kerckhoffs, Zeigler, SCS publications, 1986.