A Retrospective of a Pioneering Project. Earlier than XML, Other Than SGML, Still Going: CuProS Metadata for Deeply Nested Relations and Navigating for Retrieval in RAFFAELLO

Ephraim Nissan[1] and Jihad El-Sana[2]

[1] Department of Computing, Goldsmiths College, University of London, 25–27 St. James Street, New Cross, London SE14 6NW, England, U.K.
ephraim.nissan@hotmail.co.uk
[2] Department of Computer Science, Ben-Gurion University of the Negev, Beer-Sheva, Israel
el-sana@cs.bgu.ac.il

Abstract. The spread of XML vindicated a project that the present authors developed earlier, independently of SGML, the parent-language of XML (SGML was originally devised for communicating structured documents). Our own project was the NAVIGATION component of the RAFFAELLO retrieval system. The latter was originally devised for managing the deeply nested, flexible relations that constituted the lexical database of the ONOMATURGE expert system for Hebrew word-formation. Whereas RAFFAELLO within ONOMATURGE was a simpler version, such that retrieval was done by means of retrieval functions implemented in LISP and reflecting the actual structure of the nested relations as known beforehand, the version of NAVIGATION that was implemented by El-Sana under Nissan's supervision used a metadata schema. The syntax for describing metadata in RAFFAELLO was in a language defined by Nissan, CUPROS (short for *Customization Production System*). Whereas a few articles about RAFFAELLO were published, none explained CUPROS and the NAVIGATION tool as implemented by El-Sana. The unabated interest in XML, Web technology, and ontologies (an area relevant for the database of ONOMATURGE) have vindicated Nissan's contention that *nested relations* (a research area in database design) should be allowed not only unlimited depth of nesting, but also extreme flexibility of structure. This is now taken for granted because of how metadata are defined for XML, but the feasibility of the idea was shown by El-Sana's managing to implement retrieval as based on metadata description, along the lines of CUPROS syntax. In this article, apart from explaining the syntax of CUPROS and also describing El-Sana's implementation of retrieval, we also illustrate the approach to metarepresentation through an exemplification from the structure of lexical frames in ONOMATURGE. In particular, we discuss variants of a given derivational pattern of word-formation, and we also discuss the evolution of terminology for given lexical concepts, throughout historical strata of Hebrew. We show how

this is handled in nested relations, but a fuller discussion is provided in appendices. The approach is also exemplified based on a project applied to Italy's regional constitutions.
Keywords. Metadata, knowledge-representation, retrieval, RAFFAELLO, CUPROS, ONOMATURGE, ontologies, nested relations, XML, lexicography, hospitality management, legal computing, history of computing.

Table of Contents of the Article

1 Background of the Project

1.1 A Lexicographical Database of Deeply Nested Relations

ONOMATURGE is an expert system developed by Nissan in 1983–1987. Its area of expertise is word-formation, within computational linguistics. It task was to generate and propose a gamut of new Hebrew terms (candidate neologisms that express a concept whose definition was entered as input), and to evaluate how 'good' those terms would seem to native or fluent speakers of Hebrew. That is to say, the program also had to produce an estimate of the psychosemantic transparency of the coinages it generated, and according to that estimate, it would rank them. The control of ONOMATURGE was described by Nissan in [68], whereas the structure of the database was dealt with in [69]. In the present volume, see [79].

The lexical database of ONOMATURGE consists of a collection of large chunks of lexicographic knowledge; each entry is structured as a deeply nested database relation.[3] A refined structure was envisaged in a sequel multilingual project, which is described in another article in the present volume [76]. In the multilingual project, lexical frames are separate from semantic frames, and moreover there are onomasiological frames (e.g., about how given lexical concepts are named across languages; in particular, semantic shifts are described in such frames).

In the database of ONOMATURGE, retrieval is performed by a tool, RAFFAELLO, whose early versions were described in part of Nissan's PhD thesis [72, Sec. 7 and Appendices E and F], and then, in a more limited fashion, in the articles [65, 70, 71]. In ONOMATURGE, a procedural control component accesses nested relations

[3] Nissan's earlier research in database design had adopted the *universal relation* approach instead [61, 62, 66]. In the universal relation approach to the design of relational databases, there is a conceptual phase such that there virtually is only one relation. The latter is global, is defined by all attributes that are relevant for the database, and its set of attributes forms a space in which the values are coordinates on the respective attribute's axis, thus identifying all those individuals (points in the space) that match a query. In the words of Ullman [103], in the universal relation approach, "the objects are [...] the smallest sets of attributes for which a significant connection is represented in the database", but even though object-based semantics of universal relations have been defined in those terms, it is the opposite approach with respect to nested relations. The universal relation's goal is relieving users from having to use schemata; for certain applications, this approach remains very interesting. However, mapping complex objects onto the universal relation model is a very elaborate task, as early as the database design stage. As Ullman [103], in polemic with Kent [34], recognised, the universal relation model does not improve, with respect to previous relational models, in requiring that attribute-names be made unambiguous by unnatural-sounding strings; this may be inconvenient for objects with a deep part-explosion hierarchy. The universe of attributes should be flattened since the design phase, whereas the nested-relation model keeps the object's structure explicit. Because of all the mapping involved, universal relations don't seem adequate for situations where nested relations could give the best.

and then, based on values found there, further nested relations are sometimes accessed, and rules fire (that in turn execute programs in derivational morphology); that is, substantially, control first accesses frames, and then it accesses rules.

Importantly, within ONOMATURGE, not only retrieval but also a few other functions were associated with RAFFAELLO, the main of these being having the word-formation patterns triggered across several programming languages.[4] This was described in [72, Subsec. 7.3.1, 7.3.2, 7.3.3], and will not be dealt with in the present article, because it is outtside the scope of showing how we combined a rule-driven metarepresentation language with an object-level representation in

[4] RAFFAELLO allows to access and exploit parts of the knowledge-base that are coded in languages different from LISP. This applies to frames simulated under INGRES, but — and this was especially important for the version of ONOMATURGE that was in actual use — this applies also to program-bases heterogeneously implemented: for ONOMATURGE, the collection of derivational programs. Some of them were implemented in LISP (according to different programming approaches), but most of them were implemented in the Shell or CShell languages of the UNIX operating system. "Some Unix systems support more than one Shell. For example, on the Berkeley UNIX system both the Bourne Shell and the C-Shell are available, as the programs **sh** and **csh** respectively." [54, p. 417]

What were the respective advantages of LISP and of Shell or CShell, in the implementation of derivation programs? On the one hand, LISP was found to be best suited for those kinds of inflection where *string explosion* is needed. On the other hand, when *concatenation* (but not explosion) was needed, Shell or CShell implementations proved to be *compact* and relatively *readable*. Moreover, a relevant ruleset was already implemented by Nissan in Shell or CShell, before he finally decided to focus on LISP, while implementing ONOMATURGE.

Rules in ONOMATURGE are specialised in Hebrew word-formation, because the task of the expert system was to generate, evaluate, rank, and propose candidate neologisms in Hebrew. Some such word-formation patterns (patterns of derivation rather than compounding) are suffixes, but usually, they are *free-place formulae* — which is typical of the morphology of Semitic languages. Some rules of ONOMATURGE were coded in LISP (following one out of several possible conceptions and implementations in the same language), or in the Shell or CShell command languages of UNIX. The control programs of ONOMATURGE ask RAFFAELLO to trigger a certain rule, whose *name* was retrieved from some frame instance, but without knowing where it belongs to, from the implementational viewpoint. The format of arguments in different implementations (e.g., in different programming languages) may happen to differ. The input arguments are transformed as suited, and then executed. RAFFAELLO returns the result to the program that invoked it. Unlike `exec` — a FRANZ LISP function that accesses Shell and executes Shell commands from inside LISP, Nissan's interface suitably converts the format of argument-lists, special characters, and so forth. This was performed according to information introduced by the user in the *clone* (i.e., modified copy: a copy of the file, with some modifications introduced) of the interface that was customised for the given application. This provides more flexibility in the integration, as it wouldn't be realistic to assume that heterogeneous implementations (that possibly were developed differently, without the future need of their integration in mind), would share *a priori* conventions on lists of arguments.

nested relations with no conceptual limit on depth and flexibility, well before this has become commonplace with XML.

1.2 A Project for a Corpus in Constitutional Law

A similar encoding in nested relations was applied by Nissan to Italy's regional constitutions (*Statuti Regionali*) [99], trying to exploit their similarities. The description of that project as can be found in Nissan's paper [73] pays mych attention to the metarepresentation whose object-level representation is constituted by the nested relations inside which, the regional constitutional provisions are encoded.

In that project, it was envisaged that nested relations be used at an intermediate level of the knowledge representation, a level unsuitable for interfacing the user, but suitable for knowledge-engineering and -acquisition. It was also envisaged that another part of the knowledge representation (actually, its lower level), as well as possibly queries, would be logic-based, and coded in PROLOG; for example, a rule about the Regional Executive Council (*Giunta Regionale*) of Lombardy in 1972–1976 was formulated as follows:

```
in_office(  giunta_regionale,
            [  [ president,  tizio ]  ,
               [ aldermen,   [ caio, sempronio, etc ]  ]
            ]
         ) :- region_is( lombardia ), period_is( 1972, to, 1976 ).
```

In this article, part of the exemplification (in Subsec. 4.4) is going to be drawn from work originally done for this application to Italy's regional constitutions. This is worthwhile, as the metarepresentation that defines and regulates the nested relations of this project had interesting aspects that made good use of the syntax of the CuProS metarepresentation language, which itself had been originally defined by Nissan for it to subserve the development of the Onomaturge expert system for word-formation.

1.3 Another Application, in Retrospect a Precursor

This subsection describes the second application of the approach to database structure that had been inaugurated with ONOMATURGE. This second application was a program whose role was as an adviser for terminal food-processing: meal configuration under constraints on ingredient quantities, and so forth. Implementors of various phases (in 1986 and 1987)

included Edith Marmelstein and Gabi Shahar; Tzafrir Kagan and Benny Trushinsky; Gilla Shluha, Tzvi Getzel, and Adi Gavish; Dorit Nogah and Nava Sha'ya. They were supervised by Nissan.

A database of deeply nested relations, again similar to the database of ONOMATURGE, was applied to the database of this expert system prototype

for planning in large kitchens, thus the application domain was within hospitality management [67]: FIDEL GASTRO (so named after *gastronomy*) plans cooking or terminal food-processing (its intended use being at institutional or hotel kitchens) by configuring meals, as constrained by available quantities of (mandatory or optional) ingredients, and by various kinds of other constraints. Kinds of food, and ingredients, are represented in an inheritance network whose vertices are deeply nested relations. the procedural control component of FIDEL GASTRO first accesses heuristics (that can be conceived of

as rules operating on sets), and then accesses frames (and then again it accesses more frames).

Even though FIDEL GASTRO was a relatively minor project that Nissan had his undergraduate students implement in 1987, it was considered valuable enough for a journal in hospitality management to publish a paper on this application to large kitchens. Moreover, it finds now a parallel in a function of the so-called Microsoft Home, introduced in 2004, and first inhabited by Bill Gates and his family. Even though the Microsoft Home uses radio-frequency identification tags for inventory tracking, the basic function is like that of the 1987 project, FIDEL GASTRO. James Barron [6], describing the Microsoft Home, wrote: "The network knows, for example, what ingredients are available in the kitchen. If the makings for chocolate chip cookies are not at hand, it will suggest a recipe for oatmeal cookies, assuming these ingredients are on the shelf" — and this is precisely what FIDEL GASTRO was doing.

While representation in the frame-bases of ONOMATURGE and FIDEL GASTRO is similar, the architecture of the *dynamics* (the control flow) of the two expert systems is different (and somewhat "upside down" with respect to each other). In fact, in ONOMATURGE, a large procedural control component intensively accesses frames, identifies — the process — suitable rules in a ruleset, and triggers modules that perform the action part of the rule. In FIDEL GASTRO instead, frames are accessed by a procedural control, whose structural regularities suggests macro-based reimplementation in terms of a deterministic production-system storing meta-level knowledge on the dynamics.

In ONOMATURGE, a procedural and retrieval-intensive control accesses a database of frames. Among other things, upon retrieving from the frame-base certain properties that indicate rules of word-formation that are suitable in the current context, the control heuristics of Onomaturge access a program-base of modules, implementing procedural rules, and which is heterogeneous. Relevant rules fire, while a scoring procedure evaluates the application of rules to data instances by retrieving declarative knowledge on rules from another frame-base, whose frames describe single rules.

By contrast, in FIDEL GASTRO, kinds of food, ingredients, and so forth are described in an *inheritance network* of frame-instances. The frame-base is accessed by a procedural control. Control heuristics generate and constrain the search-space where a solution is sought for the problem proposed by the task. A procedural program coded in LISP fulfilled the role of control. However, this code could be easily rewritten as decomposed into a production-system of con-

trol heuristics: the ruleset would access frames, whereas in ONOMATURGE, frames cause the ruleset elements to fire. Besides, FIDEL GASTRO's control component could be repeated for different tasks, and task-oriented versions would anyway access the general frame-base on food. The first task implemented proposed possible meals, once it had learnt about ingredients or dishes that are available in a kitchen. Incrementally, it could conceivably be expanded with new constraints, that in turn would possibly need further accesses in frames, or would instead perform only computations that are not specific of *semantic* knowledge about certain kinds of food, but involve *contingent* data (i.e., *episodic* knowledge). For example, such contingent (session-dependent) constraints may involve quantities: an operations research problem should be solved, in trials to optimise heuristically (or, anyway, to enhance) the exploitment of ingredients.

In FIDEL GASTRO, one can identify two macro-components: the *database* — where detailed general information is stored about kinds of food and ingredients — and *task-oriented programs.* These programs embody a set of rather modular *heuristics* for performing a given task, and a general *control* that orders and organises the application of the heuristics to the data. Heuristics access the database intensively. Besides, they apply to input data. The task implemented consists of checking qualities and quantities of stocks available in a kitchen, and then in proposing the kinds of food that could be prepared with the basic or intermediate ingredients available, in order to constitute a meal, under constraints.

Constraints may be *quantitative:* you cannot use twice the same flour. They may be temporal, and involve sequences of actions or absolute amounts of time, in presence of a limited amount of cooking facilities. On the other hand, constraints may be *qualitative,* and refer to any out of several domains: social etiquette, dietetic, medical, religious, and so forth

The control programs of FIDEL GASTRO tackle the hierarchy of ingredients to prepare a given candidate kind of food that is being considered by the program; limited availability of ingredients is accounted for. The organization of knowledge chunks in the database as implemented includes many attributes that can be exploited to impose constraints: actually, more than the control as implemented was actually exploiting (the same is also true of ONOMATURGE). Chunks of knowledge about single kinds of food (or of ingredients that cannot be served as food on their own) are organised in data-structures of the same kind of frames in ONOMATURGE.

Let us consider, now, the *statics* of FIDEL GASTRO as reflected in the structure of single entries in its database. Afterwards, we are going to look at the *dynamics* of the system, as embodied in the control programs and in the heuristics they apply.

The control programs of FIDEL GASTRO exploit only a small part of the information that can be stored in frames, according to the following organisation. Inside each frame that describes a kind of food, an attribute `IS_A` lists such concepts of which the current concept is a subclass. It is useless to state, inside the frame of a concept, C_1, i.e., such properties that are coincident with properties

as found in frames of concepts

$$\{C_2, C_3, \ldots\}$$

of which C_1 is a subclass. In fact, those properties are enforced for C_1 as well, unless the frame of C_1 explicitly states different values that supersede the inherited property. On the other hand, the attribute `PARTICULAR_CASES` lists subclasses belonging to the current concept.

An attribute `IS_INGREDIENT_FOR` states for what kinds of food the present entry could be used as ingredient. Often, food that can be served on its own have no such value of `IS_INGREDIENT_FOR`. Knowledge stored under the attribute `INGREDIENTS` lists various properties, subdivided according to the kind of `MANDATORY_INGREDIENTS` or `OPTIONAL_INGREDIENTS`.

In terminal kinds of food (as opposed to general classes), the `QUANTITY` of ingredients is stated. Besides, the `SPECIFICITY` is tentatively quantified: this is resorted to by the control programs, in order to avoid looking for candidate dishes first according to those ingredients (such as salt or water) that do not typify only a reasonably small set of kinds of food of which they are an ingredient. For optional ingredients, a tentative quantification how the extent to which they may be renounced is also stated. Besides, standard units of measure are defined for different kinds of food.

Social information is gathered in properties nested inside the property `SOCIAL_INFO`: information on suitable occasions (events, or calendar recurrences) for serving a given kind of food fits in the property `OCCASION_STATUS`. Information on the social status of food fits in the property `SOCIAL_STATUS`. Price information fits in properties nested inside the property `PRICE`. Medical constraints and fitness constraints fit, respectively, in properties nested inside `MEDICAL_CONSTRAINTS` (subdivided by pathological condition, drug-dependent prohibitions therapy-dependent prohibitions, and so forth), and `FITNESS_CONSTRAINTS` (subdivided by kind of diet). `RELIGIOUS_CONSTRAINTS` are subdivided by religious denomination, and include constraints that depend on a particular status of a person, on place, on calendar time, on time past since the consumption of other given kinds of food, on contemporaneousness of consumption, and so forth.

`MORAL_CONSTRAINTS` is an attribute actually more closely related to lifestyle than to ethos, and includes information on `VEGETARIAN` habits (of various kinds: e.g., admitting fish, admitting eggs, excluding meat but not animal fats, etc., with default assumptions), on `ANTI_HUNT`, or `ANTI_SMELL` (thus excluding garlic), or `FLORA_CONSERVATION` considerations and sentiments (e.g., either persons, or legal systems, object to palm cabbage, as one palm is killed for each cabbage). `LEGAL_CONSTRAINTS` can be of various kinds: e.g., the prescribed addition of sesam-oil to ersatz to make it easily identifiable, or other laws against adulteration, or protectionist or prohibitionist norms, as dictated by economic considerations or by health hazards. Etiquette is described also under the attribute `HOW_TO_SERVE`. Other attributes refer to `CALORIES`, `CONSERVATION`, `TASTE`, `ODOR`, `CONSISTENCE`, and so forth. `PREPARATION_PROTOCOL` lists the attributes `TOTAL_TIME`, `MACHINES`, and `PROCEDURE`.

Let us turn to the control flow of FIDEL GASTRO. At the beginning of each session, the program interacts with the user, ideally a cook the manager of an institutional kitchen. The control of FIDEL GASTRO takes a list of stocks (with the respective quantities) that are available in a kitchen, and then it proposes possible lists of dishes to be prepared.

First of all, for each kind of stock, the system determines whether it can be served as food on its own, or it is just a possible ingredient. (This is done according to the properties `IS_INGREDIENT_FOR` and `HOW_TO_SERVE`, as found in the frame of each item in the list.) Then, FIDEL GASTRO tries to figure out what dishes could be prepared by means of the list of ingredients it has in the kitchen. It should not begin by considering ingredients that are very often used, such as salt. Instead, FIDEL GASTRO tries to direct its search into small classes of candidate dishes. Thus, for each item in the list of available ingredients, the control program checks its `IS_INGREDIENT_FOR` property, and begins by considering those ingredients that could fit only in few dishes. Besides, this check is done recursively; that is, if you have flour, yeast, and water in the kitchen, then you would find out that you can prepare dough (an intermediate ingredient), but then, those kinds of food that can be prepared by means of dough should also be looked for.

Mandatory ingredients in the list of candidate dishes are retrieved from the frame of each item, and this is done (recursively) also for intermediate ingredients. FIDEL GASTRO checks whether it has, in the kitchen, all of the ingredients necessary to prepare the candidate currently considered. After having checked the actual presence of all of the ingredients required, FIDEL GASTRO checks also whether the available quantities of ingredients: not only the initial quantities should be available for a single considered candidate dish, but interactions are also considered between the requirements of different candidate dishes as considered for inclusion in the same menu: it may happen indeed that if you employ a certain quantity of a certain ingredient for the first candidate dish in the current candidate menu, then an insufficient quantity would be left to prepare a given dish that was also candidate for inclusion in the menu. Feasible candidate menus are then displayed.

1.4 The Nested Relations Approach in Database Research

When the *nested relation* approach appeared, in database design research, nested relations were also called *non-1NF relations* (where 1NF refers to Codd's First Normal Form of database relations), or *relation-valued relations*. The latter name highlights the recursive structure of nested relations. Namely, values may themselves be relations. At the time, in the 1980s, it looked like the ascent of knowledge-based systems (expert systems, or, more in general, computer systems that exploit knowledge-representation), as well as the need to model complex objects (a need subserved also, in another manner, by object-oriented programming) was vindicating a school, in relational-database research, that during the 1970s already existed but was by far in the minority: this was the school advocating processing relations as being nested (this was felt by proponents to be a

natural description of objects in the world as modeled), without transforming them into shallow tables through Codd normalisations. In fact, expert database systems, CAD/CAM knowledge-bases, and so forth, it was already felt in the 1980s, need to represent complex objects, whose part-explosion representation naturally leads to nested relations. In relational databases with shallow relations, the universe of attributes is "flattened" as early as the design phase, whereas the nested-relation model instead keeps the object's structure explicit.

Deeply nested relations allow to define attributes by decomposing and refining the universe of notions of a given application-domain. Nissan's approach in the database of ONOMATURGE concentrated on static properties of complex objects (as opposed to active networks of objects), and thus on the very identification of attributes that fit into the database schema as nested inside each other: according to the intuitive semantics of instances of nested attributes, whenever the need was felt to articulate information associated with a given attribute, Nissan nested inside that attribute further attributes, which classify the concerned information according to a hierarchical structure. More detail involves deeper nesting just as it may involve "broadening" the set of attributes found on the same hierarchical level.

Nested relations emerged, in relational database design research, out of the desire to generalise flat relations, so that hierarchically structured objects could be modelled directly. Paper collections appeared [1, 85]. Normal forms were introduced for nested relations [86]. Mark Levene introduced important variants of nested relations [41–44].

In particular, he combined the universal relation model (that allows the user to view the database as though it was just one flat relation, in the space of all attributes of the database), with nested relations, into an approach called the *nested universal relation model* [41]. That model offered the following advantages [41, Abstract]:

> Functional data dependencies and the classical notion of lossless decomposition are extended to nested relations and an extended chase procedure is defined to test the satisfaction of the data dependencies. The nested UR model is defined, and the classical UR
>
> model is shown to be a special case of the nested model. This implies that an UR interface can be implemented by using the nested UR model, thus gaining the full advantages of nested relations over flat relations

Levene's formalisation incorporated null values into the model. More recent work on nested relations includes [7, 22, 23]. Georgia Garani [23] distinguishes nested attributes as decomposable and non-decomposable. She has proven that — overcoming a once popular maxim in nested relations research ("Unnesting and then nesting on the same attribute of a nested relation does not always yield the original relation") — "for all nested relations, unnesting and then renesting on the same attribute yields the original relation subject only to the elimination of duplicate data" (in the wording of the abstract of [23]).

1.5 "Denormalized Relations" in Cornell University's Uto-Aztecan Project

In the 1980s, it looked like in computational lexicography there were developments being published (already in the early years of that decade), that were well suited by the nested-relation approach. Grimes, who was researching the lexicon of Huichol, a Native American language in Mexico (one result was a dictionary, [30]), had developed an approach to lexicographical databases that was based on what Grimes called *denormalized relations* [29], that is to say, on nested relations. The ones proposed by Grimes were not deeply nested however, unlike the nested relations in ONOMATURGE. In 1984, Grimes [29] expressed his opinion that relations in Codd normalization are not suited for the representation of natural-language dictionaries: "It appears as if the better the dictionary in terms of lexicographic theory, the more awkward it is to fit relational constraints".

An application developed by Grimes and others was [30], the already mentioned dictionary of Huichol. Definitions were in Spanish, and the lexicon was that of a Uto-Aztecan language of Mexico. However, since the work as reported in 1984 [29], that had adopted denormalised relations, Grimes had been reconsidering the feasibility of "traditional", normalised relations for lexicography (personal communication from Grimes to Nissan, 1985). By contrast to Grimes, who was a linguist affiliated with a linguistics department at Cornell University, Nissan's main concern was with the computer science aspect of his own project in word-formation (after all, his doctoral project was in computer science), even as he strove for a linguistically rigorous treatment. Therefore, Nissan did not shy away from sticking with his deeply nested relations, as being applied to representing the lexicon, for the needs of ONOMATURGE and beyond.

2 Nested Relations, and Passive Frames vs. Active Frames

A nested relation is a *tree of properties.* In the lexical database exploited by ONOMATURGE, each entry is organised as a nested relation, and is owned by a word, a collocational compound,[5] or a lexical root (i.e., a consonantal stem as typical of semitic languages, as ONOMATURGE is applied to Hebrew word-formation). Besides, derivational morphemes as well are allowed to have a nested relation associated, whose attributes often are the same as those of lexical entries. In ONOMATURGE, chunks of knowledge are large: tens or hundreds of (indented and possibly commented) lines can correspond to a single entry, if much information is stated.

For the purposes of the expert system using the database, each entry in the database is a *passive* frame. A frame is a representation of an object in terms of its properties; it is passive if no *procedural attachment* is found that is automatically triggered on access: in programs such as ONOMATURGE and FIDEL

[5] An interesting thing about collocations, is that sometimes usage flexibilises them; see e.g. [24].

Gastro (the latter was described in Subsec. 1.3 above), it is the responsibility of control programs, exploiting the database, to perform suitable actions after consultation; in particular, the action taken could be the execution of some retrieved value. But even though from the viewpoint of knowledge engineering, these nested relations are basically passive frames (accessed by an external control), conceivably a control component in a program exploiting such a database of nested relations could exploit them as embodying active data, too: actions firing as soon as they are read.

3 Raffaello: A Toolkit for Retrieval from Nested Relations

3.1 A History of the Raffaello Project

As mentioned, from late 1983 to 1987, Nissan developed by himself Onomaturge, an expert system for Hebrew word-formation, in which a lexical database of deeply nested relations was a component. This was the original background for all subsequent work done on Raffaello, whose potential for application was of course believed to be wider. Retrieval functions of Raffaello's various versions, always applied to deeply nested relations such as those of Onomaturge, were first implemented by Nissan in the Franz Lisp programming language, as part of the Onomaturge project. Moreover, also in 1984–1986, he had different groups of students also do some work on Raffaello by using Franz Lisp. Afterwards, Nissan had his students develop further versions of Raffaello, and these versions were implemented in either Prolog (this was the version developed by Jihad El-Sana in 1989), or the C programming language (which was the case of a version developed in 1989 especially by Ziad Ashkar).

As said, he earliest version of Raffaello was developed by Nissan, who implemented retrieval functions in Franz Lisp: these functions carried out retrieval from the nested relations of a lexical database, and were invoked by the control component of Nissan's Onomaturge expert system. Between 1984 and 1990, his students who were involved in the development of the Raffaello toolkit for retrieval from deeply nested relations were, at first, Gai Gecht and Avishay Silbeschatz, and next, for different versions,

- on the one hand Jihad El-Sana, who developed in Prolog a version of Raffaello that carries out retrieval from nested relations by consulting a metarepresentation coded in the sc CuProS language as defined by Nissan,
- and on the other hand Ziad Ashkar, Ilia 'Ablini, Muataz Abusalah, and Fadel Fakher-Eldeen. These other students developed, in the C programming language, a version of Raffaello that does not consult the metarepresentation.

In 1985, Yehudit Shakhi and 'Irit Dahan implemented a nested-relations based representation of semantic componential analysis (in lexical semantics). This project was inspired by Nissan's concomitant work in computational linguistics, but did not contribute to Onomaturge itself. By 1988, Nissan was developing

a multilingual schema for the lexis and semantics. In 1989, his student Barak Shemesh developed a program transforming a nested subrelation into a visual representation, by applying this to diachronic onomasiological relations, that is to say, to how a given lexical concept came to be expressed by a different set of terminology over time, in different historical strata of the same language. This application is discussed in Appendix B, at whose end Barak Shemesh's project is briefly described.

In 1989, Nissan designed an adaptation of deeply nested relations to the representation of legal or consistutional concepts, namely, a representation that exploited similarities among Italy's regional constitutions: some of our exemplification in the present article is taken from that project in legal computing, as there were niceties to the metarepresentation of the nested relations that it is worthwhile to reproduce here.

3.2 Operational Aspects of of RAFFAELLO

The PROLOG version of RAFFAELLO converts the syntax of the nested relations (originally developed so it would be handled in a LISP environment) into nested lists as in a syntax that PROLOG can handle; then, retrieval from the nested relation is performed, according to a database schema (i.e., a metarepresentation) that specifies legal nesting of properties. The PROLOG version of RAFFAELLO stands out among the other version, in that it navigates inside the nested relations by using the metarepresentation to guide it.

A query addressing the nested-relation representation is a predicate that takes a list as argument; the list is the *retrieval path* through the nested levels of the frame: the path consists of the name of a frame, followed by a sequence of names of nested attributes, from the outermost, to the innermost interested. Incomplete but unambiguous paths are also allowed. As for traversing an unnamed chunk, instead of naming the attribute (that in this case is implicit), one has to state the key value (that is identified as such because unlike names of attributes, it includes at least one lower-case letter). As for the case when repeated chunks of different attributes occur — that is, in the generalised multiset case — the path has to include the name of the attribute, followed by the key value that identifies the particular chunk.

The PROLOG-coded retrieval tool consults the metarepresentation, and is able to analyse its syntax: Nissan defined a language, named CUPROS, short for *Customization Production System,* as it is rule-based. Writing down a metarepresentation coded in CUPROS customises the kind of knowledge representation for the given application. Likewise, using that metarepresentation coded in CUPROS customises the retrieval tool for the particular application. Each rule describes legal nesting in a given attribute, that is named in the left side; implicit attributes (those with unnamed chunks) have a conventional name, in use only inside the metarepresentation, and written (at least partly) in lower case.

4 The Metarepresentation

4.1 Preliminary Remarks

As early as the ONOMATURGE project, and regardless of the fact that the kind of retrieval used inside that expert system did not consult the metarepresentation while accessing the nested relations of the lexical database, the relation between the nested relations and the formalism by which their structure is regulated was clearly defined. The nested relation itself is at the *object-level* of representation, as opposed to the *meta-level representation* (or *metarepresentation*). Each kind of nested relation may have several instances at the object level, and one metarepresentation.

What was special (before the widespread adoption of XML) about our kind of frames (beside their being passive rather than active, i.e., beside the absence of *procedural attachments* being triggered on access), is these frames being structured as deeply nested relations. The nesting schema of attributes is flexible: its instances in different frames are likely to vary even considerably. This makes is relatively easy to add new kinds and new formats of information.

Control, as subserving the specific task of ONOMATURGE, exploits information found under several attributes inside frames. However, attributes as actually found — with values filled in — in implemented instances of database entries, are only a subset of attributes that are allowed to appear, according to a *meta-level representation* (or *metarepresentation*) stating legal nesting of attributes, and expressed in terms of a *production system* augmented with the syntax of a special language, CUPROS. This metarepresentation reflects a decomposition of the knowledge-domain into a large set of attributes, and flexibility as allowed is far-reaching.

Meta-level representation, as opposed — in frame-systems and knowledge-bases — to the *object-level representation* that it defines and regulates, is the equivalent of what a *data-dictionary* is for a traditional *database.* Metarepresentation, in knowledge-based systems, is not necessarily limited to describing the structure of the database component: in fact, a meta-level description can be given as well of sets of heuristic rules, or even of the control component. Our retrieval tool, Raffaello, resorts however to metarepresentation (in the one RAFFAELLO version that does access the metarepresentation indeed) only as defining the database of nested relations.

4.2 General Aspects of the Object-Level Representation

Bear in mind that from the viewpoint of *Lisp*, a nested relation (no matter how deeply nested it is) is a recursive list of lists. It is only the semantics, to the human beholder, that interprets such nested relations as being trees of properties. By convention, each property is identified by an attribute, that usually is explicitly named in upper-case, but that can be implicit, in certain contexts that are specified in the metarepresentation. Each property is delimited by the parenthesis enclosing it This parenthesis includes the name of the attribute first,

followed either by one or more terminal values, or by one or more inner properties. Comments may be inserted, in the part of a line that follows a semicolon. Consider the following example (the first of a seven, taken from [73]):

Example 1.

```
(ATTRIBUTE_1
      (ATTRIBUTE_2   ( value_a   value_b )  )
      (ATTRIBUTE_3   ( value_c )
      (ATTRIBUTE_4
              (ATTRIBUTE_5   ( value_d  value_e  )  )
              (ATTRIBUTE_6
                    (ATTRIBUTE_7  ..........    ;  further
                                  ..........    ;  nesting.
                    )
                    ........................
              )
      (ATTRIBUTE_8   ( textual_value )  )
)
```

These nested properties can be all different (every kind occurs once), so they are explicitly identified by the upper-case name of their respective attribute. But sometimes chunks with the same subschema of attributes can be repeated after each other, appearing with more or less the same attributes, but with different values. That is to say, the nested properties can constitute a sequence of one or more instances of one or more attributes. If several instances of just one attribute occur in the same sequence, then it is superfluous to state the name of the attribute explicitly: each property will be an *unnamed chunk,* that can contain either terminal values, or further nested properties. For example, in the following piece of code, two unnamed chunks are nested inside the property named after `ATTRIBUTE_1`, and they each has the same structure, in the sense that all or some of the attribute nested inside one chunk, occur (with possibly different values) in the following chunk.

Example 2.

```
(ATTRIBUTE_1
      (  (ATTRIBUTE_2    (  value_a    value_b  )  )
         (ATTRIBUTE_3    (  value_c  )  )
         (ATTRIBUTE_4    .........      )
         (ATTRIBUTE_5    .........      )
      )
      (  (ATTRIBUTE_2    (  value_d  )  )
         (ATTRIBUTE_3    (  value_e  )  )
         (ATTRIBUTE_5    .........      )
)     )
```

One or more of the attributes have their value providing the identification *key* of the repeated chunk. For example, let `ATTRIBUTE_3` in the example be the key, but a key could be even multiple, and involve several (or even all) of the attributes occurring in a given chunk.

Unnamed chunks introduce, in our syntax, *multisets* (as opposed to *simple sets,* that is sets of such elements that can occur just once). In fact, unnamed chunks allow several instances of the same (implicit) attribute to be nested side by side. However, the limitation is that several occurrences are allowed for just one attribute. The most general case of multiset is allowed by the following syntactic feature; in fact, we are interested in admitting repeatable chunk-patterns even as instances of one out of a set of attributes; in such cases, it is obvious that the name of the relevant attribute — notwithstanding the repetition — must be stated explicitly (except at most one of the attributes), as in the following sample code:

Example 3.

```
(ATTRIBUTE_1
      (ATTRIBUTE_2  (ATTRIBUTE_3   (  value_a  )  )
                    (ATTRIBUTE_4   (  value_c  )  )
                    (ATTRIBUTE_5    .........     )
                    (ATTRIBUTE_6    .........     )
      )
      (ATTRIBUTE_2  (ATTRIBUTE_3   (  value_d  )  )
                    (ATTRIBUTE_4   (  value_e  )  )
                    (ATTRIBUTE_6    .........     )
      )
      (ATTRIBUTE_7  (ATTRIBUTE_8   (  value_f  )  )
                    (ATTRIBUTE_9   (  value_g  )  )
      )
      (ATTRIBUTE_2  (ATTRIBUTE_3   (  value_h  )  )
                    (ATTRIBUTE_5    .........     )
      )
      (ATTRIBUTE_7  (ATTRIBUTE_8   (  value_i  )  )
                    (ATTRIBUTE_9   (  value_j  )  )
)     )
```

We could have omitted the name of either `ATTRIBUTE_2` or `ATTRIBUTE_7`, because the occurrence of two open parenthesis characters, successively with no other characters between (excepts blanks, newlines, or comments following a semicolon), at the level nested immediately inside `ATTRIBUTE_1`, ensures that the first parenthesis opens an unnamed chunk, to be associated with the one implicit attribute allowed.

4.3 From the Object-Level, to the Meta-Level

Example 4 is a piece of metarepresentation, concerning the object-level code of Example 1.

Example 4.

```
(ATTRIBUTE_1       (  ATTRIBUTE_2
                         ATTRIBUTE_3
                         ATTRIBUTE_4
                         ATTRIBUTE_8
     )                 )
     (ATTRIBUTE_4      (  ATTRIBUTE_5
                         ATTRIBUTE_6
     )                 )
     (ATTRIBUTE_6      (  ATTRIBUTE_7
                         ...........
     )                 )
```

As for Example 2, its metarepresentation is in the following Example 5:

Example 5.

```
     (ATTRIBUTE_1    (  n:  chunk_under_Attribute_1 )  )

     (chunk_under_Attribute_1   (  ATTRIBUTE_2
                                   i:  ATTRIBUTE_3
                                   ATTRIBUTE_4
                                   ATTRIBUTE_5
     )                          )
```

In the latter example, one can see that the right-side part of rules can be interspersed with labels (ending by a colon, and typically being a letter); these labels are meant to convey information concerning the following attribute, or — as the case is of other labels, such as `x:` (which corresponds to an *exclusive or* of substructures of frame) — of a sub-unit of the right-side part of the rule, listing one or more attributes.

The label `n:` indicates that the following attribute can have repeated chunks; in the example, that attribute has its name containing lower-case letters (only one letter happens to be capitalised), thus the repeatable chunk is an unnamed chunk, at the object level.

The label `i:` indicates that the attribute that follows it, is the key attribute of the chunk-pattern. Instead, according to the code in the following Example 6, the key includes two attributes, preceded by the label `i2:` (and being unmistakably a label, because of its ending in a colon).

Example 6.

```
(ATTRIBUTE_1    (  n:  chunk_under_Attribute_1 ) )

(chunk_under_Attribute_1   (  i2:  ATTRIBUTE_2
                              i2:  ATTRIBUTE_3
                              ATTRIBUTE_4
                              ATTRIBUTE_5
)                          )
```

The code in the following Example 7 is the metarepresentation of the object-level code of Example 3:

Example 7.

```
(ATTRIBUTE_1      (  n:  ATTRIBUTE_2
                     n:  ATTRIBUTE_7
)                 )
(ATTRIBUTE_2       (  ATTRIBUTE_3
                      ATTRIBUTE_4
                      ATTRIBUTE_5
                      ATTRIBUTE_6
)                  )
(ATTRIBUTE_7       (  ATTRIBUTE_8
                      ATTRIBUTE_9
)                  )
```

4.4 A Concrete Example of Object-Level vs. Meta-Level Structure

Recall the project, mentioned earlier (in Subsec. 1.2), that encoded in nested relations Italy's regional constitutions (*Statuti Regionali*), trying to exploit their similarities [73]. The following Example 8 is the nested relation of the concept `Giunta_regionale`, that is, 'Regional Executive Council'. In the code shown, we are going to provide in detail only the representation of knowledge on how many members (i.e., regional aldermen, in Italian *assessori alla Regione*) belong to the Executive Councils of the various Regions:

Example 8.

```
(assign_frame         Giunta_regionale
    (  (DEFINITION   ( ( Every region has a Giunta_regionale )
                       ( The Giunta_regionale is the
                         Executive_organ of the Region       )
                       ( The Giunta_regionale has
                         one Presidente_della_Giunta and
                         many an Assessore_alla_Regione
```

```
                  (SEE  ATTRIBUTE  ALDERMEN  )        )
                ( The Giunta_regionale usually has
                  one Vice_Presidente_della_Giunta    )
             )
(APPOINTMENT_MODALITIES      ......................  )
(COMPETENCES                 ......................  )
(ALDERMEN
   (HOW_MANY
      (  (IF           ( Region is Abruzzo )  )
         (VALUE_IS     (        10         )  )
      )
      (  (IF           ( Region is Basilicata )  )
         (VALUE_IS     (          6          )  )
      )
      (  (IF           ( Region is Calabria
                         or  Emilia-Romagna
         )             )
         (VALUE_IS
           (AND
               ( >=  8 )  ;  the same as:  ( at least 8 )
               ( <= 12 )  ;  the same as:  ( at most 12 )
      )  ) )
      (  (IF        ( Region is Campania or Toscana )  )
         (VALUE_IS
           (AND
              ( >=   #X / 10  )
              ( <=   #X /  5  )
              (DEFINE
                 (  #X  is  how many a
                        Consigliere_regionale
                        the Region has
      )  ) )  )  )
      (  (IF           ( Region is Lazio or Piemonte )  )
         (VALUE_IS     (         at most  12         )  )
      )
      (  (IF           ( Region is Liguria )  )
         (VALUE_IS     (    at most  9     )  )
      )
      (  (IF           ( Region is Lombardia )  )
         (VALUE_IS     (     at most  16     )  )
      )
      (  (IF           ( Region is Marche or Molise )  )
         (VALUE_IS     (         at most  8         )  )
      )
      (  (IF           ( Region is Puglia )  )
```

```
               (VALUE_IS     (        12        )  )
            )
            (  (IF           ( Region is Umbria )  )
               (VALUE_IS     (         8        )  )
            )
            (  (IF           ( Region is Veneto )  )
               (VALUE_IS     ( at most one fifth of
                               how many a Consigliere_regionale
                               the Region has
            )  )             )
               ..................................
         )  ;   end of  HOW_MANY
               ..................................
       )     ;   end of  ALDERMEN
               ..................................
)   )        ;   end of frame.
```

A relatively more concise representation, that, however, would require a separate formulation of the criterion of interpretation, is the following, in Example 9:

Example 9.

```
(ALDERMEN
  (HOW_MANY
    ( (CRITERION               (  fixed number  )  )
      (SCHEMA_OF_ATTRIBUTES    (  ( Region  )  ( Value  )  )  )
      (TYPE_OF_ATTRIBUTES      (  ( strings )  ( number )  )  )
      (RELATION
                   (   (  Abruzzo   )     (  10  )   )
                   (   ( Basilicata )     (   6  )   )
                   (   (   Puglia   )     (  12  )   )
                   (   (   Umbria   )     (   8  )   )
    ) )
    ( (CRITERION               (  at most  )  )
      (SCHEMA_OF_ATTRIBUTES    (  ( Region  )  ( at most )  )  )
      (TYPE_OF_ATTRIBUTES      (  ( strings )  ( number  )  )  )
      (RELATION
                   (   (  Lazio  Piemonte  )     (  12  )   )
                   (   (     Liguria      )     (   9  )   )
                   (   (    Lombardia     )     (  16  )   )
                   (   (  Marche   Molise  )     (   8  )   )
    ) )
    ( (CRITERION            ( constantly delimited range )  )
      (SCHEMA_OF_ATTRIBUTES ( (Region ) (at least) (at most) ) )
      (TYPE_OF_ATTRIBUTES   ( (strings) ( number ) (number ) ) )
      (RELATION    (  ( Calabria
```

```
                        Emilia-Romagna )   ( 8 )    ( 12 )   )
    ) )
    ( (CRITERION   (  depends on variable  )   )
      (DEFINE      (  #X  is how many a Consigliere_regionale
                             the Region has
      )            )
      (SCHEMA_OF_ATTRIBUTES  ( (Region ) (at least) (at most) ) )
      (TYPE_OF_ATTRIBUTES    ( (strings) (  text  ) ( text  ) ) )
      (RELATION     (   (  Campania  Toscana  )
                        (  one tenth  of  #X  )
                        (  one fifth  of  #X  )
                    )
                    (   (  Veneto  )
                        (  unspecified  )
                        (  one fifth  of  #X  )
) ) ) )             )
```

In the case of the criterion `constantly delimited range`, only one instance — the chunk comprising the Calabria and Emilia-Romagna regions — occurs under `RELATION`; thus, having to define three more properties (`CRITERION`, `SCHEMA_OF_ATTRIBUTES`, and `TYPE_OF_ATTRIBUTES`) according to the new format of Example 9, yields, for the instance considered, a representation that is even more prolix than in Example 8.

Then, for the situation when the number of regional aldermen is constantly and explicitly delimited by a lower and an upper bound, the chunk identified by the key property

```
( (CRITERION ( constantly delimited range ) )
```

and repeated hereby for the reader's convenience (**Example 9′**):

```
( (CRITERION            ( constantly delimited range )  )
  (SCHEMA_OF_ATTRIBUTES ( ( Region  ) ( at least ) ( at most ) ) )
  (TYPE_OF_ATTRIBUTES   ( ( strings ) (  number  ) ( number  ) ) )
  (RELATION    (  ( Calabria  Emilia-Romagna )  ( 8 )   ( 12 )   )
) )
```

could be usefully replaced by (**Example 10**):

```
     (  (IF        ( Region is Calabria or Emilia-Romagna )  )
        (VALUE_IS  (AND  ( at least 8 )   ( at most 12 )  )  )
     )
```

in the framework of the code of Example 9, but according to the format of Example 8. According to the syntax of the CuProS metarepresentation language we defined, such flexibility in the format of the frame portion under `HOW_MANY` is expressed, in a global metarepresentation file, as follows (Example 11):

Example 11.

```
(executive_organism            ;   It is the kind of frame that
   ( DEFINITION                ;   Giunta_regionale belongs to:
     APPOINTMENT_MODALITIES    ;   the attributes may appear at
     COMPETENCES               ;   the upper level of nesting.
     x:  ALDERMEN       ;   x:  stands for   "either...  or..."
     x:  MINISTERS      ;   and delimits one or more attributes.
)  )

(ALDERMEN          (  HOW_MANY    ..........  )   )

(HOW_MANY          (  n:  if-value_chunk
                      n:  criterion-dependent_chunk  )  )

(if-value_chunk    (  i:  IF                ;  key.
                      VALUE_IS        )  )

(criterion-dependent_chunk
  ( i:  CRITERION   ;  key.
    DEFINE
    par: SCHEMA_OF_ATTRIBUTES  (applies_to: RELATION)
    par: TYPE_OF_ATTRIBUTES    (applies_to: SCHEMA_OF_ATTRIBUTES)
    multi-par:  RELATION
) )
```

The `par:` labels, in the last rule of Example 11, state that the ordered set of elements given, in the object-level representation, as value of the attributes `SCHEMA_OF_ATTRIBUTES` and `TYPE_OF_ATTRIBUTES`, have to be considered as parallel. We have seen indeed, for instance, in Example 9′, that we have the correspondence:

```
(SCHEMA_OF_ATTRIBUTES  ( ( Region  ) ( at least ) ( at most ) ) )
(TYPE_OF_ATTRIBUTES    ( ( strings ) (  number  ) ( number  ) ) )
```

where the second property defines the type of the elements listed under the first property, that, in turn, is the schema of a tabular relation, stated under the `RELATION` attribute. By contrast, the label `multi-par:`)that, in the metarepresentation formulated in the last rule of Example 11, precedes the attribute `RELATION`), states that, in the object-level representation, the property `RELATION` consists of the name of the attribute `RELATION` followed by a sequence of rows in a table, that is, followed by a list of elements, each being parallel to those properties, whose attribute name in the same metarepresentation rule is preceded by the label `par:`

To specify some semantics for the attributes of the right-side part of the rule, the optional, parenthesised statement `applies_to:` that accompanies the

attributes labelled with `par:` stipulates, in the last rule of Example 11, that the values of `TYPE_OF_ATTRIBUTES` are applied to the respective elements of `SCHEMA_OF_ATTRIBUTES` that, in turn, are applied to to the respective elements of the rows of elements (rows) of `RELATION`. The inner level (elements of elements, instead of elements) is conventionally due to the label `multi-par:` that appears before RELATION, instead of the `par:` label.

The intricacy of the `par:` and `multi-par:` syntax is due to the fact that they define the metarepresentation of properties that, in turn, are a local metarepresentation inserted in object-level representation code (indeed, the attribute `SCHEMA_OF_ATTRIBUTES` defines a syntactic schema for attributes included in the same chunk of object-level representation, while `TYPE_OF_ATTRIBUTES` defines some *data semantics,* corresponding to the local syntactic metarepresentation provided by `SCHEMA_OF_ATTRIBUTES`). These features of our metarepresentation language are meant to enhance its expressive flexibility, but are not indispensable, if one adopts our representation formalism: indeed, one can adopt the schema of Example 8, which is simpler than the schema of Example 9. Further details of the metarepresentation in Nissan's *Statuti Regionali* project can be found in [73] and — in the present Choueka Jubilee set of volumes — in [82, Sec. 3].

5 XML Comes into the Picture

5.1 General Considerations

XML, the Extensible Markup Language, is currently a very popular representation language. It is at the foundation of the field known now as the *semantic Web.* Now XML is even appearing as a data type in some other languages. The popularity of XML stems from its role as a standard within Web technology (it was endorsed institutionally), and is reflected in a multitude of languages and applications making use of it, as well as in a literature comprising textbooks or reference books [50, 32, 59], conference proceedings [21, 8, 116], and dissertations [55], as well as in reports in the information media.

The project described in this article, Raffaello, originated in the 1980s, independently of SGML (the Standard Generalized Markup Language), that itself was later simplified into XML, with which our approach is compatible, and of which it is a precursor. Most of XML comes from SGML unchanged. Our own project in knowledge-representation in deeply nested relations, flexibly structured, with no limit on the depth of the nesting, and with retrieval guided by metadata, originated separately from XML's parent language, namely, SGML [26, 9, 98, 97], that was also designed in the 1980s, and is defined in ISO Standard 8879. SGML enabled a formal definition of the component parts of a publishable data set, and when designed, it was primarily intended as a tool enabling publishable data (such as technical documentation) to be interchanged between, e.g., authors or originators, and publishers. By the late 1980s, early digital media publishers had understood the versatility of SGML for dynamic information

display. The main historical interest now of SGML is that it was simplified into XML. This ought to be enough for preserving interest in SGML.

5.2 The Significance of the Rise of XML for the Precursor Status of Raffaello and the Deeply Nested relations from Which it Retrieves

The spread of XML vindicated the project that is the subject of the present paper and that the present authors developed earlier, independently of SGML, the parent-language of XML (SGML was originally devised for communicating structured documents). At the apex of our own project, there was the Navigation component of the Raffaello retrieval system. The latter was originally devised for managing the deeply nested, flexible relations that constituted the lexical database of the Onomaturge expert system for Hebrew word-formation. Whereas Raffaello within Onomaturge was a simpler version, such that retrieval was done by means of retrieval functions implemented in Lisp and reflecting the actual structure of the nested relations as known beforehand, the version of Navigation that was implemented by El-Sana under Nissan's supervision used a metadata schema.[6]

The unabated interest in XML, Web technology, and ontologies (an area relevant for the database of Onomaturge) have vindicated Nissan's contention that *nested relations* (a research area in database design) should be allowed not only unlimited depth of nesting, but also extreme flexibility of structure. This is now taken for granted because of how metadata are defined for XML, but the feasibility of the idea was shown by El-Sana's managing to implement retrieval as based on metadata description, along the lines of CuProS syntax: ss seen, the syntax for describing metadata in Raffaello was in a language defined by Nissan, CuProS (short for *Customization Production System*).

5.3 Well-formedness in XML and the Simple Navigation Version of Raffaello, vs. the Validity of an XML Document and the CuProS-Directed Navigation Version of Raffaello

There is one aspect of XML that was first formalised in XML, but had been first successfully implemented in a number of projects, one of these being in lexicography: the software from the University of Waterloo New Oxford English Dictionary Project. That aspect is the notion of well-formedness, as opposed to validity. Well-formedness enables parsing without a schema. By contrast, an XML document being valid means that it contains a reference to a schema (a Document Type Definition, or DTD), and abides by that schema. The schema itself is a grammar. There are various schema languages for XML. The oldest of these is DTD, that XML inherited from SGML. But XML DTDs are simpler than SGML DTDs.

[6] Whereas a few articles about Raffaello were published, none explained CuProS and the Navigation tool as implemented by El-Sana.

The distinction between well-formedness and validity in XML is at the heart of the difference between the various versions of RAFFAELLO. A well-formed XML document can be parsed without a schema. The versions of RAFFAELLO that carry out retrieval from deeply nested relations according to an input retrieval path, and do not consult the metarepresentation, are able to do so because the nested relation they access is well-formed. These were the SIMPLE NAVIGATION versions of RAFFAELLO.

By contrast, the NAVIGATION version of RAFFAELLO that was implemented by El-Sana carries out the retrieval by being directed by the metarepresentation (a grammar) coded in CUPROS and which the retrieval program consults, in order to be able to navigate correctly. Thus, it is essential that the nested relation abides by the metarepresentation, and this foregrounds the notion of validity, as being now familiar from XML.

It must be said however that (as can be seen from the PROLOG code in Appendix C) even the NAVIGATION version of RAFFAELLO was exclusively guided by the metarepresentation: some of the PROLOG predicates were customised for features in the structure of a particular terminological database. Nevertheless, NAVIGATION was the most important step forward, in generalising RAFFAELLO. The next step would have been to produce a more sophisticated version, which would replace that subset of the NAVIGATION predicates with ones which would not be tailored for an attribute taken to be a constant, but could treat it instead as a variable, within a context entirely dictated by the metarepresentation.

5.4 A Project for Traslating Our Representation into XML

In 2001, Monica Larussa did as a BSc project in London — under the supervision of Ala Al-Zobaidie and Ephraim Nissan — some preparatory work [49] (including some preliminary code in JAVA) for the project building a translator into XML from nested relations as described by a metarepresentation coded in CUPROS. Because of reasons other than any technical problem inherent in the project (it rather was the departure of one of the co-investigators), this did not develop further.

And yet, if the kind of representation that is associated with the lexical frames of ONOMATURGE and with a sequel multilingual project [76] hopefully comes to be known better, it stands to reason that developing such a translator would be a worthwhile undertaking. What is more, it would be a relatively simple software project.

A possibility that was envisaged was to resort to YACC, as being a widespread tool for the construction of compilers for programming languages, in order to handle at least part of the translation from CUPROS-regulated nested-relation representations into XML, and from the CUPROS-coded metarepresentation itself into the *Document Type Definition* (*DTD*) that describes admissible structures of the semantic attributes with respect to each other, the way they can appear inside the XML document.

6 For Comparison: Data and Metadata in XML

It is typical for webpages to be coded in HTML: labels of HTML are interspersed within the textual content, and the identifiers of graphic material. HTML handles presentation, not content, which is what XML is supposed to do. If one was to search an HTML Web page for content, one could only look for the occurrences of given strings. This is because the only function of the syntax of HTML is to structure the presentation, regardless of the meaning of the content. In XML, instead, the syntax is meant to describe the semantics of the content, whereas a separate module, called a stylesheet, states how the syntactic units extracted from an XML document are to be displayed. It is not the case that the entire content of the stored data has to be displayed, only their layout will be different. In fact, it may be possible to only extract some of the information, as enclosed by given tags which constitute semantic attributes; it's the data retrieved in response to a query that will be displayed as instructed by the stylesheet, and the query itself is actually specified inside the stylesheet along with the specification of how to display the results. Furthermore, a separate module, called DTD (for *Document Type Definition*) will describe admissible structures of the semantic attributes with respect to each other, the way they can appear inside the XML document. That is to say, the DTD is the schema of a database, whose instances are the respective XML documents.

There are some minor problems with terminology, when dealing with both XML and databases, because the term 'attribute' and the term 'entity' are used differently in the XML literature and in the database literature. Here is a standard XML definition for 'attribute': "A name-value pair, separated by an equals sign, included inside a tagged element that modifies certain features of the element. All attribute values, including things like size and width, are in fact text strings and not numbers. For XML, all values must be enclosed in quotation marks". This is very much like some HTML declarations of tables, or of the inclusion of images, and the like. What is meant by 'attribute' in database terminology, is called an 'element' in the jargon of XML, and what in database terminology is 'the value of the attribute', in XML is the content of the element. If a string, this is referred to as data of the PCDATA type. If, instead, something that is nested inside the current element, then this aggregate is made of sub-elements of that element.

Consider a database relation schema, whose attributes ("attributes" as meant in database terminology, that is, "elements" in XML) are: `FORENAMES`, `SURNAME`, `EMPLOYMENT`, `PLACE_OF_RESIDENCE`, `MARITAL_STATUS`, `DEPENDENTS`, and `REFERENCES`.

One straightforward way to represent in XML such a relation for the given individual, Donald Duck, would be to simply enclose the value of each relation attribute between two tags, `<FORENAME>` and `</FORENAME>`, and so forth:

```
<PERSON>
     <FORENAMES>       Donald          </FORENAMES>
     <SURNAME>         Duck            </SURNAME>
```

```
    <EMPLOYMENT>      unemployed      </EMPLOYMENT>
    <PLACE_OF_RESIDENCE>  Duckburg  </PLACE_OF_RESIDENCE>
    <PLACE_OF_BIRTH>  Granny McDuck's Farm  </PLACE_OF_BIRTH>
    <MARITAL_STATUS>  unmarried  </MARITAL_STATUS>
    <DEPENDENTS>      Hewey               </DEPENDENTS>
    <DEPENDENTS>      Louie               </DEPENDENTS>
    <DEPENDENTS>      Dewey               </DEPENDENTS>
    <REFERENCES>      Scrooge McDuck      </REFERENCES>
    <REFERENCES>      Daisy Duck          </REFERENCES>
    <REFERENCES>      Mickey Mouse        </REFERENCES>
</PERSON>
```

Such a slavish rendering into XML of the database relation lifted from a relational database grossly underexploits the capabilities of XML. To start with, mainstream relational database technology stores the data in flat relations, that is, tables. In the 1980s and early 1990s, several database researchers tried to promote an alternative kind of relational databases, in which tables could be nested inside each other, or, to say it otherwise, attributes may be nested as being the value of another attribute. In artificial intelligence representations, too, frames usually are hierarchies of properties: the values are the terminal nodes in a tree with, say, three generations. Conceptually, such were some of the ideas that can be detected behind the rise of XML, as being a coding in which you can actually nest any levels of attributes inside each other. For example, the same XML file could store a much richer relation than the above, still for the individual whose name is Donald Duck:

```
<PERSON>
          <NAME>
              <FORENAME>    Donald   </FORENAME>
              <SURNAME>
                      <CURRENT>    Duck     </CURRENT>
                      <AS_RECORDED_AT_BIRTH>
              </SURNAME>
          </NAME>
          <EMPLOYMENT_RECORD>
              <CURRENT_EMPLOYMENT>
                     unemployed
              </CURRENT_EMPLOYMENT>
              <PREVIOUS_EMPLOYMENT>
                     <CATEGORY>    sailor      </CATEGORY>
                     <CATEGORY>    farm hand   </CATEGORY>
                     <FROM>        1936        </FROM>
                     <UNTIL>       1937        </UNTIL>
              </PREVIOUS_EMPLOYMENT>
              <PREVIOUS_EMPLOYMENT>
                     <CATEGORY>  factotum       </CATEGORY>
```

```
                    <EMPLOYER>  Scrooge McDuck </EMPLOYER>
                    <WHEN>      often          </WHEN>
                    <MODE> short-term informal contract </MODE>
                </PREVIOUS_EMPLOYMENT>
            </EMPLOYMENT_RECORD>
            <PLACE_OF_RESIDENCE>
                    <TOWN>         Duckburg      </TOWN>
                    <STATE>        California    </STATE>
            </PLACE_OF_RESIDENCE>
            <PLACE_OF_BIRTH>
                    <PLACE>  Granny McDuck's Farm  </PLACE>
                    <COUNTY>       Duckburg        </COUNTY>
                    <STATE>        California      </STATE>
            </PLACE_OF_BIRTH>
            <MARITAL_STATUS>   unmarried   </MARITAL_STATUS>
            <DEPENDENTS>
                 <MINOR>
                        <NEPHEW>     Hewey    </NEPHEW>
                        <NEPHEW>     Louie    </NEPHEW>
                        <NEPHEW>     Dewey    </NEPHEW>
                 </MINOR>
                 <SENIOR>
                        <UNCLE>      Scrooge McDuck    </UNCLE>
                 </SENIOR>
            </DEPENDENTS>
            <REFERENCES>          Daisy Duck      </REFERENCES>
            <REFERENCES>          Mickey Mouse    </REFERENCES>
</PERSON>
```

The syntax of XML requires that the hierarchical structure have one and just one root, which here is `PERSON`. This happens to be the case with nested relations from which retrieval is handled by Raffaello. This was the case of lexical frames in the database of the Onomaturge expert system for word-formation, as well as of a few other applications of the database concept.

Let the XML code of the example given above be stored in a file called duck1.xml It will have to be preceded by two initial XML statements, respectively specifying under which version of XML and with which alphanumeric encoding that file has to be processed; and to which stylesheet type and given stylesheet file the given XML file should be linked:

```
<?xml version="1.0" encoding="UTF-8"?>
<?xml-stylesheet type="text/xsl"  href="nicestyle3.xsl"?>
```

Now, let us define the DTD for the XML nested relation in which we stored information about Donald Duck. We want to state that:

– The root element is `PERSON`

- The root element contains one NAME element
- The root element contains zero or one elements of each of the following: EMPLOYMENT_RECORD, PLACE_OF_RESIDENCE, PLACE_OF_BIRTH (it may be absent for some reason), MARITAL_STATUS (it also may be absent, for example if unknown, or not relevant, or even deliberately withheld), and MARITAL_STATUS may or may not include an element SPOUSE.
- The root element contains zero or more elements of each of the following: DEPENDENTS (not anybody has other people legally depending on him or her), REFERENCES.
- Each NAME element is expected to contain either one or more strings (the symbol standing for a string in a DTD is #PCDATA), or one or more FORENAME and one or more SURNAME, and moreover, these may contain just a value, or, then, zero or more CURRENT, PREVIOUS, and AS_RECORDED_AT_BIRTH. The symbol | stands for 'or'.
- Element EMPLOYMENT_RECORD may include one or more strings, or then, zero or more elements for CURRENT_EMPLOYMENT and for PREVIOUS_EMPLOYMENT, and these possibly contain a further level of nesting, including zero or more CATEGORY elements and zero or one FROM, UNTIL, or WHEN elements
- Elements PLACE_OF_RESIDENCE and PLACE_OF_BIRTH include some string, or zero or one PLACE, COUNTY, STATE, and COUNTRY.
- DEPENDENTS may include some string, or zero or more of: MINOR, SENIOR.
- SPOUSE, REFERENCES, or elements inside MINOR or SENIOR may each include just a string, or a NAME element, or a LINK_TO_PERSON element.
- MINOR may include one or more of: CHILD, GRANDCHILD, NEPHEW, GREATNEPHEW, and COUSIN.
- SENIOR may include zero or more of: PARENT, GRANDPARENT, UNCLE, GREATUNCLE, and COUSIN (a child may be grown-up and thus no longer or minor age, yet be otherwise legally a minor)

The DTD is coded as follows.

```
<?xml version="1.0"?>
<!ELEMENT   PERSON  (NAME, EMPLOYMENT_RECORD+,
                     PLACE_OF_RESIDENCE+,  PLACE_OF_BIRTH+,
                     MARITAL_STATUS+, DEPENDENTS*, REFERENCES*)>
<!ELEMENT   NAME      (#PCDATA+ | (FORENAME*, SURNAME*))>
<!ELEMENT   FORENAME
                (#PCDATA+ | (CURRENT*, PREVIOUS*,
                             AS_RECORDED_AT_BIRTH*))>
<!ELEMENT   SURNAME   (#PCDATA+ | (CURRENT*,
                      PREVIOUS*, AS_RECORDED_AT_BIRTH*))>
<!ELEMENT   CURRENT   (#PCDATA*)>
<!ELEMENT   PREVIOUS  (#PCDATA*)>
<!ELEMENT   AS_RECORDED_AT_BIRTH   (#PCDATA*)>
<!ELEMENT   EMPLOYMENT_RECORD
               (#PCDATA+ | (CURRENT_EMPLOYMENT*,
                            PREVIOUS_EMPLOYMENT*))>
<!ELEMENT   CURRENT_EMPLOYMENT
```

```
            (#PCDATA+ | (CATEGORY*, FROM+, UNTIL+, WHEN+))>
<!ELEMENT   PREVIOUS_EMPLOYMENT
            (#PCDATA+ | (CATEGORY*, FROM+, UNTIL+, WHEN+))>
<!ELEMENT   PLACE_OF_RESIDENCE
            (#PCDATA+ | (PLACE+, COUNTY+, STATE+, COUNTRY+))>
<!ELEMENT   PLACE_OF_BIRTH
            (#PCDATA+ | (PLACE+, COUNTY+, STATE+, COUNTRY+))>
<!ELEMENT   MARITAL_STATUS  (#PCDATA*, SPOUSE*)>
<!ELEMENT   DEPENDENTS  (#PCDATA*, MINOR*, SENIOR*)>
<!ELEMENT   MINOR  (#PCDATA*, CHILD*, GRANDCHILD*,
                    NEPHEW*, GREATNEPHEW*, COUSIN*)>
<!ELEMENT   SENIOR (#PCDATA*, PARENT*, GRANDPARENT*,
                    UNCLE*, GREATUNCLE*, COUSIN*)>
<!ELEMENT   SPOUSE    (#PCDATA*, NAME+, LINK_TO_PERSON)>
<!ELEMENT   LINK_TO_PERSON    (#PCDATA*)>
<!ELEMENT   CHILDN    (#PCDATA*, NAME+, LINK_TO_PERSON)>
<!ELEMENT   PARENT    (#PCDATA*, NAME+, LINK_TO_PERSON)>
<!ELEMENT   GRANDPARENT  (#PCDATA*, NAME+, LINK_TO_PERSON)>
<!ELEMENT   UNCLE   (#PCDATA*, NAME+, LINK_TO_PERSON)>
<!ELEMENT   GREATUNCLE   (#PCDATA*, NAME+, LINK_TO_PERSON)>
<!ELEMENT   NEPHEW    (#PCDATA*, NAME+, LINK_TO_PERSON)>
<!ELEMENT   GEATNEPHEW   (#PCDATA*, NAME+, LINK_TO_PERSON)>
<!ELEMENT   COUSIN    (#PCDATA*, NAME+, LINK_TO_PERSON)>
<!ELEMENT   PLACE     (#PCDATA*)>
<!ELEMENT   COUNTY    (#PCDATA*)>
<!ELEMENT   STATE     (#PCDATA*)>
<!ELEMENT   COUNTRY   (#PCDATA*)>
<!ELEMENT   CATEGORY  (#PCDATA*)>
<!ELEMENT   FROM      (#PCDATA*)>
<!ELEMENT   UNTIL     (#PCDATA*)>
<!ELEMENT   WHEN      (#PCDATA*)>
```

The meaning of the notation can be understood by comparing this code with the explanation with which it was foreworded. The following is an example of nested relation coded in XML, and representing an aspect of the content of this quotation from the very beginning of a paper in literary studies, "Falstaff's Monster", by Clayton Mackenzie [51], concerning the plot of a famous Shakeaspeare play:

> In 1 Henry IV, II.iv, Falstaff presents his account of the Gadshill fiasco. What begins as 'two rogues in buckram suits' (line 184) grows to four (lines 188–9), then to seven (line 194), then to nine (line 204) and ends as a veritable army of eleven men in buckram (line 210) all assailing the beleaguered hero.

As an exercise, we represent the content of this quotation in XML, by analyzing its sense into hierarchy of XML elements rooted in `account`. We are only interested in Falstaff's account, and nothing else, focusing on its consecutive

variants by which the number of rogues is increased from time to time.[7] Note the use we are making of linking values such as `Tag1` and the like, to which we are giving a meaning as though they were variables, which, however, is something XML does not know about. It's only the user, and the user's queries, that will have to take this meaning into account.

```
<?xml version="1.0" encoding="UTF-8"?>
<?xml-stylesheet type="text/xsl"  href="nicestyle3.xsl"?>

<account>
   <about>
        <event>            Gadshill fiasco     </event>
        <protagonist>      Falstaff            </protagonist>
   </about>
   <reported_by>           Falstaff     </reported_by>
   <textual_reference>
        <identifying_tag>  Tag1         </identifying_tag>
        <play>
                 <title>   Henry the Fourth    </title>
                 <part>    1                   </part>
                 <act>     2                   </act>
                 <scene>   4                   </scene>
        </play>
   </textual_reference>
   <content>
        <sub-event>
            <identifying_tag>   Tag2     </identifying_tag>
            <action>
                    <is>        assault      </is>
                    <is>        fight        </is>
                    <agent>     Tag3         </agent>
                    <object>    Tag4         </object>
            </action>
            <action>
                    <is>        fight back   </is>
                    <agent>     Tag4         </agent>
                    <object>    Tag3         </object>
            </action>
        </sub-event>
```

[7] Actually, the dialogue in the play is even more complex. All of the following is taken from Falstaff's own words: "a hundred upon poor four of us", "at half-sword with a dozen of them two hours together", "Sixteen, at least, my lord", and so forth. The rogues clad in clothing of buckram are the supposed assailants whom Falstaff claims to have killed in a fight. He actually was trying to rob innocent travellers, but this is irrelevant for the encoding to ne generated, as this exercise is only interested in an aspect of his own account.

```
     <sequence>  Tag2    <contains/>   Tag5   </sequence>
     <sub-event>
         <identifying_tag>   Tag5      </identifying_tag>
         <action>
                 <is>        kill         </is>
                 <agent>     Falstaff     </agent>
                 <object>    Tag6         </object>
         </action>
     </sub-event>
</content>
<explain_the_tags>
    <explain_this_tag>
            <tag_is>       Tag3         </tag_is>
            <tag_means>
                          <some/> rogues
            </tag_means>
    </explain_this_tag>
    <explain_this_tag>
            <tag_is>       Tag4         </tag_is>
            <tag_means>
                       Tag4  <contains/> Falstaff
            </tag_means>
    </explain_this_tag>
    <explain_this_tag>
            <tag_is>       Tag6         </tag_is>
            <tag_means>
                       Tag3  <contains/> Tag6
            </tag_means>
            <tag_means>
                          <some/> men
            </tag_means>
            <tag_means>
                     <description>
                             clad in buckram suits
                     </description>
                     <how_many>
                         <claim>
                                <number>  2  </number>
                                <according_to>
                                     <version_of_account> 1
                                     </version_of_account>
                                     <textual_reference>
                                         <in>  Tag1 </in>
                                         <line> 184 </line>
                                     </textual_reference>
```

```
       </according_to>
</claim>
<claim>
       <number>  4  </number>
       <according_to>
          <version_of_account> 2
          </version_of_account>
          <textual_reference>
            <in> Tag1 </in>
            <line> 188-189 </line>
          </textual_reference>
       </according_to>
</claim>
<claim>
       <number>  7  </number>
       <according_to>
            <version_of_account> 3
            </version_of_account>
            <textual_reference>
               <in> Tag1 </in>
               <line> 194 </line>
            </textual_reference>
       </according_to>
</claim>
<claim>
       <number>  9  </number>
       <according_to>
            <version_of_account> 4
            </version_of_account>
            <textual_reference>
               <in> Tag1 </in>
               <line> 204 </line>
            </textual_reference>
       </according_to>
</claim>
<claim>
       <number>  11  </number>
       <according_to>
            <version_of_account> 5
            </version_of_account>
            <textual_reference>
               <in> Tag1 </in>
               <line> 210 </line>
            </textual_reference>
       </according_to>
```

```
                                </claim>
                            </how_many>
                    </tag_means>
            </explain_this_tag>
<account>
```

7 Early Versions of Raffaello

7.1 The Basics: Raffaello's Original Tasks

The retrieval component is the main component of Raffaello, so much so that we have been describing the latter as being a toolkit for retrieval. Within Onomaturge, all aspects of handling the nested relations were conceptually associated with Raffaello. Originally, sc Raffaello was developed in Franz Lisp, for it to perform some given tasks supporting the sc Onomaturge expert system. The main component of Raffaello is the one that, invoked by the Lisp control of an expert system, retrieves the requested data from the proper location inside deeply nested frames. In Onomaturge, this component is intensively resorted to, in order to access the frame-base. Another component of Raffaello had the task of interfacing the Lisp control of an expert system with a possibly outer program-base, through the C and Shell or CShell languages of Unix. In Onomaturge, this serves the purposes of one of its architectural components, the *Morphological Coiner* when it needs to have lexical derivation or compounding programs triggered (disregarding the fact they are functions defined in Lisp, or Shell or CShell programs).

There also was a third task, ascribed to a particular component of Raffaello, and conceived with Onomaturge in mind, but not integrated in that expert system. That component was concerned with output-acquisition, as being a form of machine-learning. Thus, as seen, the pool of tasks of Raffaello within the Onomaturge project was broader that merely carrying out retrieval. In a sense, those functions of Onomaturge that were of general interest for knowledge-based systems, and were not specific for computational linguistics, were ideally moved within the delimitation of Raffaello.

7.2 Why a Production-System as Metarepresentation, While the Object-Level is in Nested-Relation Frames?

The very choice of how to structure the relations in the database, the decision to have a meta-level representation, and the decision to have it in the form of a ruleset (in practice, a grammar), was taken by Nissan early in his doctoral project.[8] By that time, the very idea of *knowledge-representation metadescriptive languages* was already found in the scholarly literature (in [28, Sec. 6], and in [25]). Greiner [28, Sec. 6] termed such a language, a *representation language language* (or *rll* for short); the language RLL-1 he introduced and described was

[8] This subsection is based on [72, Subsec. 7.2.2.8].

intended to be a *proto-rll,* that is the first of a class of knowledge-representation metalanguages. According to Greiner, "every part of an rll system must be visible", that is: the rll should be self-describing; every component of an rll (slots, data types, modes of inheritance, data-accessing function) must be explicitly represented in the system [28, Subsec. 6.1].

Even earlier, in the 1970s, in databases, attributes were already being described described by means of a *data dictionary.* Integrated *metadata* management provides a sort of self-description in the database [52]. In the database literature, this is the equivalent of metarepresentation as found in the knowledge representation sector. The difference mainly stems from the difference in the object-level representation models.

More in general, *metamodelling* investigates reasoning and knowledge acquisition processes as used in *modelling,* that is to say, in the representation of reality to obtain knowledge for problem solving. Not limited to computing, *metamodelling* fits in the systemic approach to organization and management, and concerns the design of systems to control banking or financial systems, health care systems, the environment, and so forth.[9]

According to Greiner's 1980 approach to metarepresentation, frames are described by frames. Each attribute is also the owner of a frame (at a metalevel), where information is found that describes the conventions about the attribute. Yet, Greiner mentioned pedagogic difficulties with the use of levels of frame-representations.

In Nissan's CuProS approach, a production system states the legal class of hierarchical structures inside frames. Each rule expresses the legal childset of a given attribute.[10]

---

[9] For example, van Gigch had been publishing epistemic papers in metamodelling [106, 107]. But this was within general systems theory [104], a rather unprestigious area of scholarship, arguably for good reason. Conferences in the domain certainly used to be omnia gatherum, if not anything goes. At any rate, [108] is a book on metamodelling in decision making. Van Gigch [105] discussed the relation between *metadesign* and system failure, and also elaborated about the difference between modelling and metamodelling. At the time, van Gigch was affiliated with the School of Business and Public Administration at California State University in Sacramento.

[10] It can be shown that CuProS production systems are related to *tree-grammars.* As soon as 1973, Barry K. Rosen defined *subtree replacement systems* [89]. In formal language theory (this is well known), *grammars* are *generating devices*, whereas *automata* are *accepting devices.* Metarepresentation with Raffaello as being automatically processed in its version called Navigation (that, anyway, consults it, does not compile it), concerns the *accepting* side of the medal. As being dual with *graph-grammars*, Witt [114] formally defined *finite graph-automata* as *graph-acceptors.* By contrast, our implementations (Nissan's in Lisp, and El-Sana's in prolog) of Navigation was *ad hoc*, with no sound formal support. John Barnden, at a conference talk, subdivided scholars and practitioners of artificial intelligence into two branches: the *messies*, and the *neaties.* The neaties are the theorists. The others are the messies, and the term is not used with any derogatory intention. CuProS (like most of Nissan's work in artificial intelligence) fits in the messies' camp. Not

A rule expressed in CuProS may be considered as being an attribute/value tree with only one level: the left-hand side and the right-hand side may be considered as being, each, a property. In order to describe the semantics of attributes, one should augment the internal structure of each rule in the metarepresentation, with *meta-attributes* other than `FATHER` (that is, the left-hand side) and `CHILDSET` (that is, the right-hand side). Thus, the metarepresentation would be brought back to metalevel frames.

Nissan had found out however that — from the software engineering viewpoint — keeping all of the "genealogical" information on attributes in one structure (a production system) is an easy style of metaknowledge acquisition. Further, semantic details may follow (possibly in another structure). Strings that synonymously refer to the same attribute are listed in a different structure (a list of synonym-lists, in Raffaello). We preferred to *separate* such structures that are meant t store those secondary details: stating such full-fledged metaknowledge details would cause the advancing front of the knowledge analysis (it was feared) to slow down. In Nissan's experience with frame-bases, *attribute genealogy* is the proper carrier of the thrust forward, in this analysis. Now, a *production system* as metarepresentation allows to *focus* on attribute genealogy. By contrast, metarepresentational *frames* would lead you to *schedule* entering secondary details early, together with the genealogic information for each attribute: Nissan believed that this is undesirable. Good metarepresentation structures ought to elicit a proper style of metarepresentation.

7.3 Simple Navigation vs. Navigation

The earlier three versions of Raffaello were implemented in 1984–1986. None of those versions actually required the metarepresentation, that was defined nevertheless within Onomaturge, in order to regulate the structure of its nested relations, and enable it to be flexible. During that stage, the metarepresentation was still only a tool for humans, rather than a tool for an automated component to make use of it.

We have already seen that at the metalevel of representation, a schema — a metarepresentation — stating what is valid nesting among attributes, is drawn in terms of a *production system* (a ruleset, in practice a formal grammar) augmented with the syntax of the CuProS language of metarepresentation; that language was specifically defined by Nissan in the framework of the Raffaello project. That syntax was defined especially in 1984–1985, in connection with the first application to Onomaturge.

As opposed to the *metalevel,* the *object level* of representation is constituted by instances of nested relations, whose internal organization of attributes fits in a space of possibilities as allowed by the metarepresentation. Instances describing the same kind of object are allowed to look even very differently, as

---

that the two camps have been doing war on each other. Rather, they have been complementary, in the history of the discipline.

to the hierarchy of nested attributes: however, such freedom is constrained by the specifications in the metarepresentation.

The retrieval component comes in two branches of versions: SIMPLE NAVIGATION, vs. NAVIGATION. Of the latter, a partial implementation was first done in LISP by Nissan, but the fuller specification of NAVIGATION only came to fuller fruition when El-Sana implemented it. SIMPLE NAVIGATION. is the version that Nissan integrated into ONOMATURGE as being his own doctoral project. In 1984–1985, Nissan implemented low-level parts of SIMPLE NAVIGATION: basic retrieval functions, attribute-synonymy management, and so forth. In 1984–1986, SIMPLE NAVIGATION was developed and applied to ONOMATURGE, as being (*per se*) a clonable program-base specialized in the internal topology of frames structured as deeply nested relations. In 1986–1987, SIMPLE NAVIGATION was applied to FIDEL GASTRO, an expert system for short-term planning in large kitchens.

A more sophisticated version is NAVIGATION. Retrieval, as performed by NAVIGATION, proceeds by consulting the query, the metarepresentation, and the concerned instances of nested relations. This way, NAVIGATION is independent from the application-domain, from which it is uncoupled by means of the application-dependent metarepresentation.

On the other hand, SIMPLE NAVIGATION is an alternative version, consisting of a modular program-base that can be cloned for specific applications. A customisation of SIMPLE NAVIGATION is easily constructed by the human developer consulting the metarepresentation that was previously drawn as a specification, design and reference tool, but the programs themselves do not consult the metarepresentation. The human developer consults it, in order to adapt or compose retrieval functions needed by the control. In fact, SIMPLE NAVIGATION is a module-base of retrieval functions, specialised for a taxonomy of topological situations of property-nesting inside frames. Only the low-level functions of SIMPLE NAVIGATION exhibit some intricacy, and once Nissan had coded them, there was no need to touch them again; rather, those functions that are defined to retrieve the value (or the subframe) found in a specific attribute described in the metarepresentation were very simple, composable, and easily reused in new applications.[11]

The ideal respective role of NAVIGATION and SIMPLE NAVIGATION suits different phases of the development of an expert system. NAVIGATION is suited for early phases when the creative impetus should not be fettered by rote tasks of

[11] In the application to ONOMATURGE, the retrieval functions of SIMPLE NAVIGATION were stored in several directories (i.e., folders). Each directory includes one or several retrieval functions, being either simple compositions of other functions, or simple functions that are "drains" (terminal "leaves") in the direct acyclic graph of invoking among the functions of the customisation of SIMPLE NAVIGATION for ONOMATURGE. Terminal functions of the customisation invoke lower-level functions of SIMPLE NAVIGATION, that is, such functions that their code does not need to be touched when customising SIMPLE NAVIGATION for a knowledge-based system.

providing an (efficient) access from the control programs of the expert system being developed, to knowledge chunks that are being developed.

In such perspective, SIMPLE NAVIGATION can be customised for the application in later, calmer moments in the development process. Once the control programs of the expert system have been shaped, and it is clear what information is needed in what points, NAVIGATION queries can be replaced by invocations to modules of the customized SIMPLE NAVIGATION, that no longer involve the steps taken in the general case by NAVIGATION, but instead are specialized for given paths of retrieval as preferred by the control programs of the particular application.

One of the motivations of our approach to metarepresentation applies not only in the framework of NAVIGATION (that has to consult it), but more in general in RAFFAELLO. Metarepresentation is a handy tool during the the specification, knowledge analysis, design, and knowledge-acquisition phases (including instance coding). The metarepresentation specifies the statics of the knowledge-base. The control of the application is woven by referring to the set of attributes that are already present in the metarepresentation, and by motivating the insertion of new attributes.

In the SIMPLE NAVIGATION version, alternative to NAVIGATION, the metarepresentation is not consulted by that program while it is running, but the customisation of SIMPLE NAVIGATION for a given application is developed according to the specific needs of the control programs of the application, and the programmer constantly refers to the metarepresentation while s/he composes modules to retrieve specifically along preferred paths.

See in Fig. 1 how the control of NAVIGATION proceeds, on processing a query, once the involved frame has been loaded. The top level of frames is processed in a slightly different manner with respect to nested levels, because on processing the top level, low-level functions invoked resort to property-retrieval functions FRANZ LISP, but eventually El-Sana implemented the system in PROLOG). On processing inner levels of nesting, instead, NAVIGATION has to make headway on its own, recursively for each level. As shown in Fig. 1, three data-structures are considered concomitantly: the path provided in the query, the metarepresentation, and the frame-instance as reduced to a currently relevant subtree.

Different kinds of actions to be taken, whenever the currently considered token in the path provided by the query is an attribute-name, as opposed to a value (interpreted as identifying a slotless facet). The test is: "Consult the next token, $Y := TOKEN(j)$, in the path provided by the query". If the token is an attribute, then consult production, $P(j)$, with $TOKEN(j)$ as left-hand side. But if the token is a value instead (interpreted as identifying a slotless facet), then Y is needed to identify the proper facet instance in a sequence of nested facets with a similar nesting-schema (i.e., the same production).

Besides, one should consider also that data-structures in RAFFAELLO include a list of lists of synonymy among attribute-names. For example, `MORPHO_CAT` is the same as `MORPHOLOGICAL_CATEGORY` in the lexical frames of ONOMATURGE.

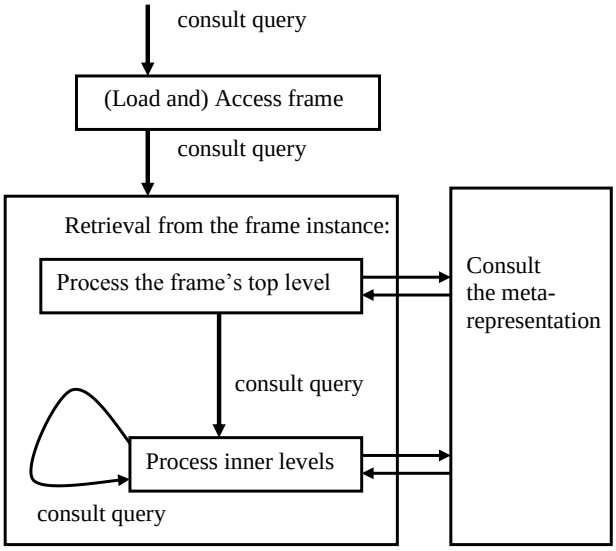


Fig. 1. An outline of the control flow in NAVIGATION

.

Such equivalent attribute-names are handled by SIMPLE NAVIGATION. The same feature could be added to NAVIGATION.

In 1986, a core of NAVIGATION was implemented that tackles the major syntactic features of the CUPROS metarepresentation language; implementers are students of the *Expert Systems* course taught at the time by Nissan; those students were Ofer Hasson, Orly Priva, Ilan Shaked and Ronen Varsano. A fuller version was implemented from scratch by El-Sana, that also resorted to a different programming language]in order to do so.

There is also a third version, a rather early one, of the retrieval component of RAFFAELLO: it retrieves from nested relations that are reconstructed after having been mapped onto INGRES flat relations.[12]

However, this version was very slow, as one virtual step of retrieval corresponds to several actual steps. By comparison, the INGRES-oriented version indicated the advantages of direct manipulation of nested relations. The INGRES-oriented version was implemented in 1985 by two students of the *Expert Systems* course taught by Nissan, namely, Gai Gecht and Avishay Silberschatz. This INGRES-oriented version is described in Sec. 9.

[12] INGRES is a relational database management system (DBMS) available under the UNIX operating system: see [17, 115].

7.4 A Protocol Integrating NAVIGATION and SIMPLE NAVIGATION

This subsection is based on [72, Subsec. 7.2.3.2]. NAVIGATION is an "expert" version, and a tool independent from the application-domain: NAVIGATION is expert in looking into a frame in a "new" application-domain for which it wasn't specifically coded (which is the case of SIMPLE NAVIGATION instead). On penetrating frames, NAVIGATION consults their legal schema, like a tourist that normally lives in a city, and therefore knows how to go around in an urban environment, but does not know the particular city he or she is visiting, and nevertheless manages to visit it, by consulting a map. NAVIGATION is "expert" itself, at least in the sense that its *data, rules,* and *control* are separate, and that control determines the application of rules to data according to the context (as was expected of expert systems).

- In fact, as we already know, the *ruleset* accessed by NAVIGATION is the CUPROS metarepresentation of the application.
- By contrast, the attribute-names of the application are the *data* of NAVIGATION, and are described in the metarepresentation. Frame-instances accessed are also data.
- *Control* performs *conflict resolution,* in order to disambiguate attribute-names that refer to the roots of different hierarchies (nesting-schemata). Control has to disambiguate attribute-names according to the context ("ancestry"), which is the same as having to select rules among a collection of rules with the same lefthand side. The need for this stems from the fact that the syntax of CUPROS is as liberal as to allow the same attribute-name to be the father of different subschemata, according to its own ancestry in the metarepresentation, that is in different contexts.

In terms of computational steps, NAVIGATION "thinks a lot", and therefore is slower than SIMPLE NAVIGATION. This clearly was the case of the LISP version, the earlier one implemented. By contrast, SIMPLE NAVIGATION is not an "expert" program itself, as it just executes an algorithm with no distinction between a general control and rules and data. (Let us consider again the tourist metaphor: SIMPLE NAVIGATION could not visit a foreign city, as it has no general knowledge of "walking". It can only follow a very specific path.)

With NAVIGATION, you place an invocation to *one* standard primitive function, independent from the application-domain, whenever, in the control programs of the application, you need knowledge to be retrieved from frames. On invoking that NAVIGATION function, you give it, as argument, the name of the frame (or a variable representing that value), followed by a list of attributes: a path along which NAVIGATION is going to visit the tree of properties, and retrieve what it finds at the end of the path. In order to do that, NAVIGATION consults a ruleset describing legal structures of frames for the given application (terminology, in the case of ONOMATURGE), and it retrieves the required values (possibly, not just a value, but a subtree of properties: a subframe).

By contrast, with SIMPLE NAVIGATION, we use not one general function, but several different functions for each path of retrieval, in the specific schema

of attributes of your application; therefore, you need to invest time to develop this gamut of functions. True, it is easily done, but if you do that in the initial phases of implementing a prototype of the control programs of your application, devoting time to the customisation of a SIMPLE NAVIGATION version — that is to do trivial work — would slow you down, right when you should concentrate on creative work, that is on modeling expertise. This is why it is preferable to begin developing a prototype of the control programs of an expert system, by using NAVIGATION instead of SIMPLE NAVIGATION. Afterwards, once you find some spare time, you can install SIMPLE NAVIGATION instead. Typically, this moment will come once the schema of legal frames — that is to say, the metarepresentation — of the given application is settled reasonably, as one can tell because the growth of the schema, and especially the modification rate, has slowed down, and because the preferred invocations of retrieval in your control programs have been individuated (by then, the control programs exploiting the frame-base are already in a rather advanced phase of development.)

Retrieval efficiency enhancement can be obtained by programmers, by replacing the "expert" NAVIGATION with the faster but "dumb" SIMPLE NAVIGATION. By using a metaphor from the domain of programming language compilers, turning to the SIMPLE NAVIGATION option is a kind of *manual* knowledge compilation, if you'd pass us this expression.[13] Think of a human being who, on becoming skilled in a given task, acquires habits and shortcuts (or in a learning machine compiling its heuristics for more particular, but frequently occurring classes of situations). Similarly, the difference between how NAVIGATION and then SIMPLE NAVIGATION work, is that certain heuristics are no longer thought about, steps are skipped, and performance is faster.

Now, let us consider in further detail the *protocol* of project management integrating the use of NAVIGATION and SIMPLE NAVIGATION. Using both these alternative versions of the retrieval component of RAFFAELLO, according to the protocol, is meant to enhance *prototyping.* SIMPLE NAVIGATION is (we have seen) a modular collection of retrieval functions, specialised for the metarepresentation instance — describing attribute-nesting inside frames — of the knowledge-based system for some particular application. of application. It is quite easy, even though not a creative phase in projects, to apply that software again: modules may be *reused* as they are, to construct composites, but in order to maintain identifiers in the source code semantically adherent to the application, Nissan around 1987–1988 used to *clone* modules, and in copies he just modified attribute-names. NAVIGATION is domain-independent, and metarepresentation-driven.

[13] In artificial intelligence, if an artificial agent is able to optimise its heuristics by itself (thus, unlike RAFFAELLO), for a class of situations that recur often, then we speak about self-improving automatic learning, and, in particular, about *knowledge compilation* into ready patterns. (*Compilation* is the term, as opposed to a thoughtful, step-by-step, *interpretive,* and therefore slow execution of rational heuristics). We should stress that this does *not* mean that RAFFAELLO is a *learning* machine: the optimisation of retrieval is not performed automatically; it is human intervention on the programs that replaces NAVIGATION with SIMPLE NAVIGATION.

NAVIGATION is able to visit new topologies (we used earlier the metaphor of a tourist who is visiting a foreign city and consults a guide), and this is useful to avoid a long preparation of access to frames on the part of control: this way, you are allowed to prototype control expeditiously. Functions in SIMPLE NAVIGATION are specialised for *paths of intensive retrieval.*

The programmer consults the metarepresentation, while building the SIMPLE NAVIGATION instance: this is a shortcut with respect to the way NAVIGATION (that is to say, the metarepresentation-driven version) performs retrieval, and thus spares execution time, but such "optimisation" is in at the expenses of coding-time. This way, a way was found for transferring the investment in terms of coding time: using NAVIGATION first, allows *rapid prototyping.* See below a list of the steps of the protocol, which schedules first the most creative work.

Step 1. First, one draws the metarepresentation, by trying to identify relevant parameters or objects in the considered area of knowledge, writing a production, and then refining the analysis recursively. The bulk of attribute-genealogy elicitation comes first; secondary details follow (e.g., attribute-name synonyms, or any non-genealogic metaknowledge). The production system stemming out of top-down refinement may be integrated with productions generated by considering specific examples, that is by a bottom-up analysis. According to needs, the metarepresentation may be augmented or corrected even during the remaining steps.

Step 2. Implement the control programs of the application, where NAVIGATION is invoked by means of the call

```
(navigate_and_retrieve <frame-name> <attribute> ... <attribute>)
```

which states a path for visiting the frame's internal hierarchy. You see the prototype running on few sample object-level frames.

Step 3. Fill object-level knowledge in more frames, and possibly rulesets.

Step 4. When you have time, you may "optimise" (anctually, just enhance the efficiency of) the retrieval, by adapting a SIMPLE NAVIGATION module-base for your application, and by replacing the application-control's calls to NAVIGATION, with calls to SIMPLE NAVIGATION functions. Step 4 is easy, but not creative.

The protocol we described allows to concentrate one's efforts on creative work first. The knowledge structures are defined in the metarepresentation, and then work proceeds with control. Seeing the *prototype* running is not delayed beyond what is strictly necessary.[14] Enhancing access efficiency is allowed to be delayed until it has become clear what are the accesses that control needs,

[14] Approaches to *rapid prototyping* for expert systems were described, e.g., in 1988 by Shaw et al. [95].

or needs more intensively. If and when a group of personnel takes over, and is not acquainted with the metarepresentation (e.g., because the latter was developed by someone else), then one or more *system rebirth sessions* with a trainer may be useful: he or she would simulate regenerating the metarepresentation, along the already threaded path of knowledge analysis. The group would penetrate the structure organically, and thus profitably. This protocol wasn't applied to ONOMATURGE: when Nissan began to build the frame-base of ONOMATURGE, he built the SIMPLE NAVIGATION component of RAFFAELLO, whereas the NAVIGATION component became available only later. The principle that Nissan was nevertheless advocating, of using NAVIGATION first, was a lesson he drew *a posteriori*. It was hoped that it would become useful for future new applications of RAFFAELLO.

8 Queries

8.1 The Specification of NAVIGATION's Queries

This subsection is based on [72, Subsec. 7.2.2.2]. It defined the format of queries, as addressed to the knowledge-base by the control programs of an expert system, by invoking the main function of NAVIGATION, namely, the function `Navigate_and_retrieve`. A simple query, consulting just one frame, and selecting there information that is located by means of a given path, can be described by the following grammar:

```
<simple_query> ::=  (navigate_and_retrieve
                           '( <frame-instance_name>
                              <path>
                    )       )

<path>  ::=  <facet-identifier>  |  <facet-identifier> <path>

<facet-identifier> ::= <attribute-name>
                       | <slotless-facet_identification_value>
```

Conventionally, attribute-names in our frames include at least one capital letter, and never include a character in lower case. digits and special character can be included, except characters that are special for LISP (e.g., a dot).

In the framework of the lexicographical database of ONOMATURGE, specifications for NAVIGATION prescribed that one should be able to formulate a query like the following:

```
(navigate_and_retrieve
      '(gid~ul ACCEPTATIONS ARE tumor RELATED_TO RULES
        SUFFIXES  omt  RELEVANCE_ORDINAL)
)
```

This query consults just one frame-instance: the one for `gid~ul` (the noun *gidduúl* in Hebrew has the acceptations 'tumour' and 'growth'). That acceptation had this subframe in ONOMATURGE's database:

```
( RELATED_TO
    ( RULES
       ( SUFFIXES
           ( ( IS  ( omt ) )
             ( RELEVANCE_ORDINAL  ( 1.5 ) )
       )  )
       ( COMPOUND-FORMATION_PATTERNS
           ( ( IS  (giddul_ba-X) )  ;  X is a metasymbol.
           (RELEVANCE_ORDINAL   ( 1 ) )
    )  )
    ( LIKELY_CONTEXT     ( medicine )  )
       ( NEAR-SYNONYMS
            ( ARE   ( ( TERM_IS       ( sarTan ) )
                    ( KEYWORD_IS      ( cancer ) )
                    ( RELATIONSHIP  ( particular_case ) )
            )
            ( ( TERM_IS      ( neo-plasTi ) )
              ( KEYWORD_IS   ( neoplastic ) )
              ( RELATIONSHIP ( expresses_quality ) )
       )    )
       ( FREQUENCY_SCORE_AMONG_ACCEPTATIONS    ( 1 )  ) )
```

For the other acceptation, 'growth', the value 2 was given for frequency score among acceptations; this is a subjective estimate, and the context is Israeli Hebrew. For the acceptation 'tumour', associated word-formation patterns were listed under RULES. Names for particular kinds of tumour are formed in Israeli Hebrew as compounds (literally, 'tumour in the X'). Moreover, apart from present-day standard medical terminology (let alone medical terminology in medieval Hebrew), there was an attempt made by Even-Odem and Rotem in 1975 to introduce mcuh medical terminology [18], that however did not get institutional approvel, and therefore did not gain acceptance and did not enter use. Even-Odem and Rotem's medical dictionary (and in particular, Rotem's introduction) proposed to translate the international derivational suffix *-oma* for names for tumours,[15] with a Hebrew suffix *ómet* (phonemically: /-omt/). This was applied throughout their dictionary, using Hebrew stems.

In the example from ONOMATURGE, the acceptation-identifier is the value of an attribute, `KEYWORD`, found as nested under `MEANING`, that is nested inside the acceptation-chunk. As we want to access the acceptation identified by `tumor` under `KEYWORD`, we include `tumor` after `ARE`, in the query: by consulting the metarepresentation, NAVIGATION will learn it should access `KEYWORD` (in the

[15] For example, in English, *epithelioma* denotes an epithelial tumor, *endothelioma* denotes an endothelial tumor, and so forth.

proper location), to check the identity of acceptation-chunks, and thus select the acceptation-chunk in which we are interested. Inside that chunk, we want to have the facet of the attribute `RELATED_TO` selected, and thereinto, we specify the selection of the attribute `RULES`. The subframe whose top attribute is `RELATED_TO` clusters such linguistic information that would not fit under the attrubute `MEANING`, in the acceptation-chunk. `RULES` lists word-formation morphemes possibly employed to convey the meaning of the acceptation.

The suffix-chunk identifier is the value of the attribute `IS`, nested inside the suffix-chunk. We specify, in the path, that we are interested in the chunk describing the suffix identified by `omt` (i.e., the phonemic transcription of the suffix proposed by Even-Odem and Rotem; in ONOMATURGE, phonemic transcription was used throughout). Inside the suffix-chunk, we are interested in the value of the attribute `RELEVANCE_ORDINAL` — this being a numeric estimate of the frequency of use of the suffix `omt` in forming modern Hebrew terms indicating specific tumours, with respect to other suffixes or different kinds of formative morphemes that in Hebrew are able to convey the same meaning, as being identified by the keyword `tumor` (but it must be said that the frame under consideration was developed for the sake of the example: actually, that particular suffix did not manage to get acceptance in Israeli Hebrew medical terminology).

8.2 Queries with Variables

In this subsection, we continue with the specification for the format of queries, the way it was defined for NAVIGATION in [72, Subsec. 7.2.2.3]. According to the syntax of FRANZ LISP, whenever a list is preceded by a quote sign, then it is interpreted as consisting of constants only. Without the quote before the parenthesis, the first token found inside the parenthesis would be assumed to be the name of a function. On the other hand, the syntax of FRANZ LISP allows to insert variables inside a list of constants. Then, the parenthesis should be preceded by a backquote sign, instead of the quote, and the variables inside the parenthesis should be preceded by a comma. (Whenever a quote can be used before a parenthesis, a backquote, too, can be used. If no comma is included, then the effect is the same.) This syntax was incorporated in the specification of the syntax of queries as allowed by NAVIGATION, according to the definition in [72, Subsec. 7.2.2.3]. For example,

```
(navigate_and_retrieve  `( ,object  MORPHOLOGICAL_CATEGORY ) )
```

would retrieve the value of the attribute `MORPHOLOGICAL_PROPERTY`, as found at the top level of the frame-instance whose name is the value of the variable `object` which is assumed to have been assigned a value before.

By contrast, if we want a variable inside a path as stated in a query, to qualify all of the instantiated range at the current range of nesting, then the syntax of NAVIGATION queries, as originally defined, envisages another feature: *variable-names* that are not preceded by a comma, but, instead, by an underscore. Each string beginning by an underscore — if used where a *chunk-identifying value*

would fit (like `omt` in the example given earlier in this subsection) qualifies a fan of values, whose type is determined by the position in the path. In his doctoral project, Nissan had not implemented yet the specific handling of variable-names beginning by an underscore, but this could be easily done in LISP, because in LISP any data can be evaluated, and interpreted as a variable or as a function. In the event, once El-Sana implemented an operational version of NAVIGATION, queries conformed to the syntax of PROLOG instead. See the code of NAVIGATION in Appendix C, and the metarepresentation of how attributes in ONOMATURGE are nested, in Appendix D.)

For example, in the query

```
(navigate_and_retrieve
  '(gid~ul ASSOCIATIONS_OF_IDEAS ARE _Any_value FOR_WHOM) )
```

the variable `_Any_value` qualifies the values `Goedel` and `giddily` — supposing that you bothered to include such values in the frame of `gid~ul` in a subframe whose top attribute is `ASSOCIATIONS_OF_IDEAS`. By lexicographers' practice, there is no reason in the world to list something like this subframe in a lexical entry, but we are making up an example for the sake of illustrating the syntax:

```
( ASSOCIATION_OF_IDEAS
   ( ARE ( ( IS   ( Goedel )
           ( FOR_WHOM ( ( FOR ( computation-theorists ) )
                        ( LIKELIHOOD   ( 0.05 - 0.3 )  )
                      )
                      ( ( FOR   ( mathematicians
                                  computer-scientists
                        )       )
                        ( LIKELIHOOD   ( 0 - 0.1 )  )
                      )
                      ( ( FOR   ( laypersons ) )
                        ( LIKELIHOOD   ( 0 ) )
             )  )          )
             (  ( IS    ( giddily ) )   .....
) )     )
```

The query

```
(navigate_and_retrieve
     '(gid~ul  ACCEPTATIONS  ARE  _Any_acceptation
               RELATED_TO    LIKELY_CONTEXT       ) )
```

qualifies the value of `LIKELY_CONTEXT` inside both acceptations (e.g., the value is `medicine` when `LIKELY_CONTEXT` is under `RELATED_TO` inside the acceptation-chunk of `tumor`), and should return the list of lists

```
(  ( medicine )   ( quantifiable_thing  life ) )
```

where the order among the inner lists carries no meaning. Another kind of variable to be introduced, is a variable representing any attribute that is found as nested directly inside the current level of nesting. Such variables are identified by a plus symbol prefixated. For example,

```
(navigate_and_retrieve
       '(gid~ul  ACCEPTATIONS  ARE  tumor
                 RELATED_TO  RULES  +Any_rule ) )
```

should retrieve all of the single-rule chunks, found either under `SUFFIXES`, or under `COMPOUND-FORMATION_PATTERNS`, which are the two attributes matching `+Any_rule` in the example we have been considering in this subsection. Thanks to the built-in syntax of Franz Lisp, according to the original definition of Navigation users would be able to include a sub-path as being the value of a variable included inside a path stated in a query. For example, if the variable `subpath1` was previously assigned, as value, the list

```
'(ARE  _Any_acceptation  RELATED_TO)
```

then the query

```
(navigate_and_retrieve
  `(gid~ul  ACCEPTATIONS  ,@subpath1  LIKELY_CONTEXT ) )
```

would be interpreted just as:

```
(navigate_and_retrieve
    '(gid~ul  ACCEPTATIONS  ARE  _Any_acceptation
              RELATED_TO    LIKELY_CONTEXT       ) )
```

According to that specification of the syntax, `,@` (not just a comma) would have had to be prefixated before the variable-name (i.e., `subpath1`), if only a comma was prefixated, then the value of `subpath1` would have been inserted inside the path as being an internal list:

```
'(gid~ul  ACCEPTATIONS
          ( ARE  _Any_acceptation RELATED_TO )
          LIKELY_CONTEXT
 )
```

8.3 Motivation of the Metarepresentation from the Viewpoint of Queries: Consulting Keys, and Allowing Incomplete Paths

One could ask, why should a metarepresentation be used at all, as retrieval could just select the specified nested attribute, in the current facet.[16] One reason is

[16] The present subsection is based on [72, Subsec. 7.2.2.4].

ascertaining the identity of slotless facets, by accessing their identification key in the proper location, as indicated by the metarepresentation. This could not be done just by sticking to the actual nesting instantiation as found in frame-instances, if application-dependent information on the position of keys of given kinds of slotless facets is not provided to the retrieval programs. Such information is incorporated in modules of SIMPLE NAVIGATION as composed according to the application. NAVIGATION, instead, is an application-independent program, and draws information about the application from an interface, constituted by the metarepresentation.

Another reason for using a metarepresentation, when Nissan defined it for the lexical frames of ONOMATURGE, concerns extensions he envisaged for NAVIGATION. This motivation consists in the aim of freeing users from having to know exactly the path of nesting reaching the attribute whose value they need, or reaching the nonterminal where the subtree they need is rooted. We wanted to allow paths, as stated in queries, to include just some necessary elements ("landmarks"), along the nesting path as found in the schema, or in the frame-instances. This is an extension of NAVIGATION that eventually was implemented by El-Sana.

By using a metarepresentation, freedom degrees are provided for stating a simple query without specifying the whole path: it would be sufficient to state just the final facet-identifier in the path, and some intermediate facet-identifiers included as being "landmarks". Such landmarks are necessary only when ambiguity is involved because of attributes with the same name but with different "ancestry" (i.e., with a different sequence of nesting).

Just few of facets — those whose schema is repeated in a sequence nested at the same level — have facet-identifiers, in our approach. Such facet-identifiers can be found either at the top level of the repeatable subframe, or even at a deeper level, according to definitions in the metarepresentation. Another approach, that was described by Setrag Khoshafian, Patrick Valduriez, and George Copeland [36], introduces system-generated identifiers termed *surrogates,* independent of any physical address, and being the value of a special attribute, `SURROGATE`, for each and every tuple object in a nested structure. (See Subsec. 10.2.)

9 The Early Version of RAFFAELLO that Maps Nested Relations to INGRES Flat Relations

9.1 An Overall View

In 1985, a program was developed by two of Nissan's students under his supervision, that on the one hand, maps Nissan's LISP nested relations onto flat relations under INGRES and, on the other hand, reconstructs the LISP frame instance according to relations stored in INGRES, and then answers queries.[17]

[17] This section is based on [72, Subsec. 7.2.5]. The implementors of the INGRES-oriented version of RAFFAELLO were Gai Gecht and Avishay Silberschatz, who had taken the *Expert Systems* course taught by Nissan. INGRES is a relational database management

The INGRES-oriented version of retrieval with RAFFAELLO was developed and tried as being an alternative to frames as built in files and processed by the all-in-LISP versions of the retrieval component of RAFFAELLO. The INGRES-oriented component was meant to probe certain opinions on the implementation of nested relations. In nested relations, each *compound tuple* may be sconsidered to an instance of a frame-instance *à la* RAFFAELLO (in the latter, however, the nesting hierarchy of attributes is more flexible).

The RAFFAELLO/INGRES interface proved to be slow, due to cumulation. This was ascribed especially to the elaborate mapping as supported specifically by INGRES, and, besides, to the path of retrieval, as shown in Fig. 2.

One virtual step of retrieval may require, say, forty actual retrieval steps in INGRES, if the query requires access into a deep level of nesting inside the simulated frame-instance. Also storage actions are slow, see Fig. 3.

From this viewpoint, our experience corroborated the claims motivating the efforts to build DBMSs based on a nested-relation algebra, without any stage of mapping into flat relations. The relative slowness of the INGRES-oriented version of retrieval with RAFFAELLO was taken to be indicative of the advantages one could expect to accrue from direct manipulation of nested relations with RAFFAELLO. Some approaches to nested relations in the early 1980s used to accept nested relations at the user front-end, and internally transform nested relations into flat relations.

Office forms, input as nested relations, are transformed into flat relations inside APAD, as described by Kitagawa & al. in 1981 [38].

Because of the cost of the transformation, an alternative idea was more appealing: a more efficient solution would resort to such database management systems that are specifically devised for processing nested relations. Projects in this direction were under way during the 1980s in various places. As a matter of fact, prototypes were being built by researchers (in industry, IBM had been

---

system (DBMS) available under the UNIX operating system: see [17, 115]. The overall project we are describing was given by Nissan the name RAFFAELLO, because of the same semantic motivation by which the name POSTGRES was given to a successor of INGRES. But RAFFAELLO was obtained by pseudo-algebraic reasoning. When Nissan began the ONOMATURGE project, initially (in 1984) it was considered whether to implement the database of that expert system in INGRES, for all of there going to be nested relations. What happened instead was that Nissan carried out the all-in-LISP implementation of the SIMPLE NAVIGATION version of RAFFAELLO. The DBMS of UNIX was called INGRES after the painter Ingres, who influenced the Lyon School, itself related to the Pre-Raffaellites. For the sake of simplicity, Ingres was assimilated to a Pre-Raffaellite. Then, from

$$\text{AFTER} + \text{INGRES} = \text{AFTER} + \text{PRE} + \text{RAFFAELLO}$$

we obtain: AFTER + INGRES = RAFFAELLO. This is how the name RAFFAELLO was devised. Afterwards, Nissan learned that Stonebraker and Rowe, the developers of INGRES presented [100] in 1986 the POSTGRES database management system, intended to be the successor of INGRES. The name POSTGRES was derived from *post + INGRES*.

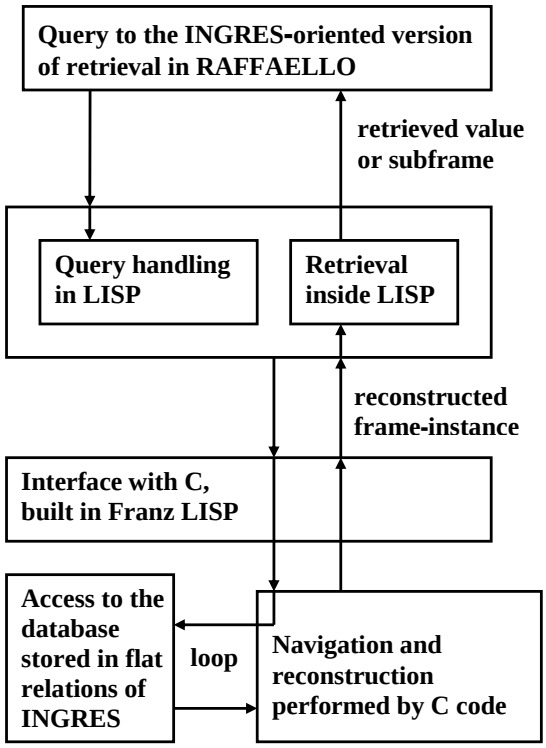


Fig. 2. Retrieval with the INGRES-oriented version of RAFFAELLO

.

developing a tool), but at the time Nissan started to develop the ONOMATURGE project, such nested-relation based database management systems were not commercially available as yet.

Nested relations research was published, e.g., in [1, 85]. Such research was carried out in Darmstadt (West Germany) and at the IBM Research Center in Heidelberg, as well as by groups of researchers in Paris, in Austin (Texas), in Tokyo, and elsewhere. In personal communication to Nissan (1985), Prof. Hans-Jörg Schek remarked: "It would be important to support AI data structures by DB data structures. The more similar these structures are, the more effective is the support which expert systems layers can get from a DB kernel".

9.2 Query Handling in the INGRES-Oriented Version of RAFFAELLO

Retrieval with the NAVIGATION version of RAFFAELLO is *syntax-directed,* as it consults the nesting-schema in the metarepresentation. The INGRES-oriented *instance-directed:* attributes nested directly inside the same property (that is,

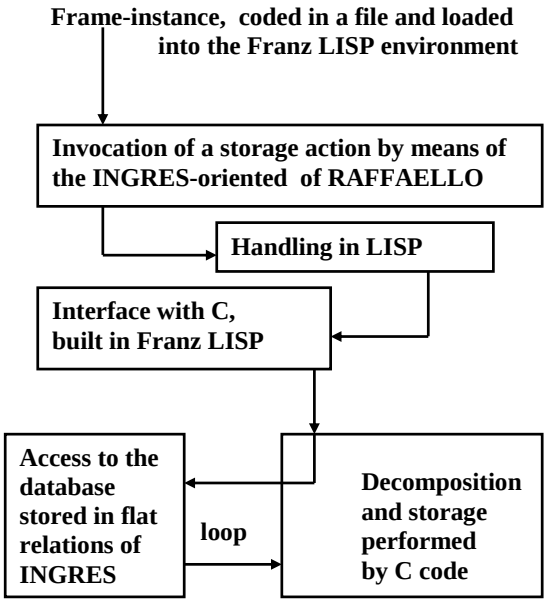


Fig. 3. Storage with the INGRES-oriented version of RAFFAELLO

.

childsets) as found in the frame-instance under consideration are stored as mapped onto a set of *tuples* (rows) scattered in INGRES *flat relations* (arrays); in order to retrieve, the frame-instance is reconstructed in LISP from the INGRES relations, and the relevant data are extracted to satisfy the query.

In the all-in-LISP versions of retrieval in RAFFAELLO, frames are loaded into LISP directly from files, without accessing any different logical representation, which the INGRES-oriented version used to do instead, by accessing a transformed representation stored under the INGRES database management system.

In Sec. 8, we have considered the format of queries as per the specification of NAVIGATION, and saw that they consist of as invoking the function `navigate_and_retrieve` with its arguments. The same format was adopted by the INGRES-oriented version of retrieval. The implementation of the function `navigate_and_retrieve` in the *Lisp* and PROLOG versions of NAVIGATION accesses one tree-level at every recursive step; each step invokes several function-levels, each processing one of the labels of the CUPROS metarepresentation language, with which the productions of the nesting-schema are interspersed. Some frame facets are not headed by any attribute-name: such properties are *slotless facets*, also called *slotless chunks*. They are identified by the value of a suitable "child" or "grandchild", being the identifying key of the slotless chunk among its similarly structured siblings. Access to the proper instance of repeated slotless facet requires consulting (by backtracking) instances of a certain attribute meant for identification: either a directly nested property (a "child"),

or an attribute nested even more deeply (a "grandchild"), devised for identifying the slotless chunk. The *filler* of a property (i.e., of a *facet*) is whatever follows the attribute-name inside the parenthesis, if the property is slotted, or the parenthesis itself, if the property is slotless.

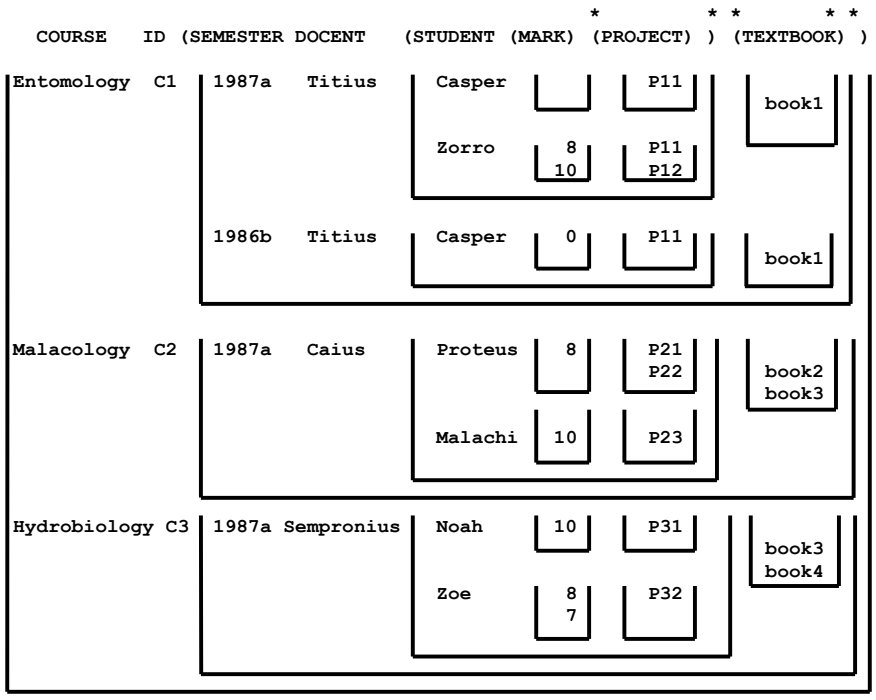


Fig. 4. Hierarchical schema of a frame-instance (ignoring actual attribute-names).

Refer to Fig. 4. The query `(navigate_and_retrieve '(<path>))` where let `<path>` be

```
(<A>_slot <AB>_slot <ABAAA>_value <ABAB>_slot)
```

retrieves the filler of `<ABAB>_facet` after consulting `<ABAAA>` in order to identify `<ABA>` as opposed to `<ABB>` or `<ABC>`. The INGRES-oriented version of retrieval, in sc Raffaello, transforms `<path>` into a path such that `<ABAAA>_value` is explicitly preceded by a flag signalling that grandfather-identification is needed. Then, accessing INGRES, search is performed by backtracking.

The RAFFAELLO/INGRES interface exploits the option, which FRANZ LISP offers [40], to access functions external to LISP that are themselves in the C language. C itself [35] allows a UNIX Shell command to be executed, by assigning it as argument to the function `system`. Then, if the Shell command invokes INGRES, we access INGRES from inside FRANZ LISP; see Fig. 2 and Fig. 3. Heuristics

need not to know whether frame-instances referenced are available in LISP or in INGRES. If a frame-instance needed is not in any of the files loaded into LISP, then the INGRES-oriented version of retrieval tries to retrieve it from INGRES.

9.3 Details of the Representation Under INGRES

The *childset cardinality* (or *ChiSCard*) of a property (i.e., facet) whose slot (i.e., attribute) is `FACET` is the number of facets actually found in the childset of `FACET` in the frame-instance considered. Thus, *ChiSCard*(`FACET`) is variable, and may differ with respect to the corresponding facet in other instances, or in the same instance once modified. We defined the INGRES relations REL1, REL2, REL3, and so forth, for *ChiSCard* being 1, 2, 3, and so forth, until an upper limit that is practically immaterial. Let *width*($\texttt{REL}_i$) be the number of *domains* (i.e., columns) of $\texttt{REL}_i$ defined for *ChiSCard* $= i$. Now,

$$width(\texttt{REL}_i) = 1 + 3i$$

The first element is a numeric key: *TID*, that is, the *tuple identifier* inside the relation. Then, each child has a *triplet* of elements associated. If the property (a vertex, in the nesting-tree of the frame-instance) is *nonterminal*, then the elements are: a numeric code *CODE* of the *facet-token*, and two pointers.

A *facet-token* is any token that represents a facet in a CUPROS metarepresentation: any attribute (even terminal, that has no left-hand side in the CUPROS production system), or any *slotless-facet identifier*, that is, any identifier that can appear as a left-hand side in productions of the metarepresentation, hence, not only attributes, but even identifiers of properties that are not headed by an attribute-name (such as the nonterminal in the metarepresentation, corresponding to the root of the subtree of a single acceptation, in ONOMATURGE).

The facet-token may be either a mnemonic string (possibly long, or admitting synonyms), or — if the facet is slotless — a left-hand side in the metarepresentation but not an attribute-name slot in the facet as found in the frame-instance at the object-level of representation. The correspondence of facet-tokens with *CODE* numbers is stated in a suitable file.

Bear in mind that RAFFAELLO allows *slot-homonymy* (that is to say, *attribute-homonymy*). That is to say, a slot, i.e., an attribute-name, may mean different things as with different *ancestry*, i.e., as nested (in frames at the object-level representation) inside such facets that are described by different productions in the metarepresentation. Homonymous slots have the same *CODE* (as the INGRES-oriented version is instance-driven, we were able to afford that).

As for *pointers* (the 2nd and 3rd elements of triplets), one of them points to the $\texttt{REL}_i$ relation according to the *ChiSCard* of the vertex corresponding to the triplet, while the other pointer points to the relevant tuple there.

Instead, if the vertex is a *terminal*, then the *CODE* element is followed by a pointer to the relation *LEAVES*, and by a pointer to the tuple, there, that stores the value of that terminal property.

Let us consider the frame-instance of Fig. 4. The root (and the name) is <A>. In our database in INGRES, the relation ROOT has a tuple listing <A> and pointers to a particular tuple of 10 elements in REL3 (as <A> has 3 children):

```
{  <A>_TID     ,
   <AA>_CODE   ,   "REL2"    ,   <AA>_TID   ,
   <AB>_CODE   ,   "REL3"    ,   <AB>_TID   ,
   <AC>_CODE   ,   "LEAVES"  ,   <AC>_TID     }
```

In the case of slotless facets, as the subtree-root is not an identifying slot, the search process checks, say, a certain grandchild to be distinguished. (Actually, repeated instances of the same facet in the same childset may either be slotless, or have all the same slot that by itself does not provide identification.)

Tuples consist of the tuple key (*TID*) and of triplets, whose 1st elements are *CODEs* of facet-tokens. An alternative solution could eliminate that element, and reduce triplets to just pairs, at the cost of having another element in the tuple: a pointer to a tuple in a $\texttt{CHILDREN}_i$ relation with i columns. Tuples there would be ordered sets of *CODEs* of facet-tokens, for every childset met with $ChiSCard = i$. The identity of pairs in $\texttt{REL}_i$ tuples would depend on the order found in the tuple of $\texttt{CHILDREN}_i$ that is being pointed to.

10 The CUPROS Metarepresentation, vs. Data-Dictionaries

10.1 A Separate Nesting-Schema, vs. an All-Encompassing Data-Dictionary

The CUPROS metarepresentation is useful as a knowledge-analysis tool, a specification, and a reference during the design, implementation and maintenance phases. Why should a metarepresentation of legal nesting among attributes, be better than an all-encompassing schema, including type definitions of the value-sets of each attribute?[18] In RAFFAELLO, we separate the data-dictionary from the nesting-schema, by reckoning that developing first just a skeleton, in terms of nesting, subserves the knowledge-analysis better, without fettering it with type definitions that could follow in calmer moments, or even directly after a session of skeleton-development.

We have investigated also a metarepresentation involving data-types, separately from the nesting-schema. In 1985, *Expert Systems* students, Yehudit Shakhi and 'Irit Dahan, in coding a data-type schema for a certain application, in the framework of *local* metarepresentation subtrees, that can be occasionally found inside (or pointed to from inside) a frame-instance, and that are meant to be applied to a portion of that instance, and possibly to a class of inheriting instances. This feature however was not integrated into the control of NAVIGATION. For the time being, it was felt, it was quite adequate to have just

[18] This section is based on [72, Subsec. 7.4].

one data-structure storing a metarepresentation, and this would be processed by NAVIGATION by default; optionally, the name of the metarepresentation structure can be provided explicitly. Switching from a global metarepresentation to a local metarepresentation is envisaged by the `g:` label in the syntax of the CUPROS metarepresentation language, but this was not handled yet by the implementations of NAVIGATION. We are going to see that on an example in Subsec. 11.5 below.

10.2 Indexing for Large Frames

Let us consider possible ways of access to frame instances, of the kind dealt with by RAFFAELLO. The simplest way (the one that was implemented) is finding the name of the frame-instance, and then accessing it: knowledge coded procedurally in the control component embodies the indication of the particular attribute whose value it needs. Another way, that is allowed by NAVIGATION, is accessing a particular attribute, once the name of the frame-instance is provided in a list together with the path from the root of the frame to the attribute considered.

The name of frame-instances can be found in other frames, or is drawn from the input of the expert system using RAFFAELLO. While we didn't implement any inheritance manager in RAFFAELLO or RAFFAELLO (but did so in FIDEL GASTRO), [19]

such a component has to draw upon chains of pointers from given properties in a given frame-instances, to other frame-instances. The way RAFFAELLO was actually developed, if no list of frame-instances to be accessed is known *a priori*, then we have to perform brute-force search. In ONOMATURGE, such situations do not occur, but for the sake of generality, this problem should be taken into consideration. If we have no way of knowing in what subset of frame-instances, certain values are likely to be found inside the instances (even though we know what attribute to look for), then we can gain insight into the contents of the frame-instance only by entering it from the root of the property-tree, and by performing a blind search.

To make search "smarter", we could develop an approach for accessing frame-instances according to their contents. One idea is to have an *index* or a *concordance* of frame-instances automatically generated and updated. *Indices* are well-known not only from textual document processing, but from from database organisation as well.

[20] By the time Nissan and his students were developing RAFFAELLO, the issue of indexing for nested relational structures has already been investigated, in the

[19] FIDEL GASTRO is the expert system for gastronomy and planning at institutional kitchens we described in Subsec. 1.3 above.

[20] Principles of *secondary indexing* — as well as *indexing* in general — in traditional database approaches current in the 1980s were discussed, e.g., by Cardenas [10, Sec. 2], Ullman [102, Secs. 2.7 to 2.9], Hatzopoulos and Kollias [31], and Schkonik [92]. Cf. Ghosh [27, Secs. 3.4, 5.1] and, in Wiederhold [110], Subsecs. 4-2, 4-3, but in particular his Subsec. 4-4, that deals with indexing for *tree-structured files*.

literature. Khoshafian et al. [36] discussed in 1986 an approach to complex objects (nested structures) that identifies *every* tuple object by means of a unique identifier, termed a *surrogate* (or a *level-id surrogate*). With RAFFAELLO, identifiers of nested facets (i.e., *subframes*) are necessary only for sequences of repeated subframe-instances with the same schema; our approach is more flexible because of other reasons, as well: it enables flexibility concerning the location, inside the subframe, of the key, i.e., subframe-identifying attribute. Khoshafian et al. [36] described an approach to indexing: the *Fully Inverted Hierarchical Storage Model* (*FIHSM*).

Consider the following code:

```
C1: {[ SURROGATE:     sur_1      ,
       COURSE:        Entomology ,
       BY_SEMESTER: { {[ SURROGATE:    sur_2  ,
                         SEMESTER_IS   1987a  ,
                         DOCENT:       Titius ,
                         STUDENTS:  {  {[ SURROGATE: sur_3 ,
                                          IS:       Casper ,
                                          MARK:       nil ,
                                          PROJECT:    P11
                                       ]}  ,
                                       {[ SURROGATE: sur_4 ,
                                          IS:        Zorro ,
                                          MARK: { 8 , 10 } ,
                                          PROJECT:
                                             { P11 , P12 }
                                       ]}  ,
                                    } ,
                         TEXTBOOK:     book1
                      ]} ,
                      {[ SURROGATE:    sur_5  ,
                         SEMESTER_IS   1986b  ,
                         DOCENT:       Titius ,
                         STUDENTS:  {  {[ SURROGATE: sur_6 ,
                                          IS:       Casper ,
                                          MARK:     0      ,
                                          PROJECT:  P11
                                       ]}  ,
                                    } ,
                         TEXTBOOK:     book1
                      ]}
                    }
   ]}
```

This is an example of an adaptation of that approach to express our frames equivalently. It is the equivalent, with surrogates as described by Khoshafian

and colleagues, of the frame instance of an academic course shown in the following code:

```
(assign_frame   C1
   (  (COURSE       Entomology)
      (BY_SEMESTER
        (  (SEMESTER_IS   1987a)
           (DOCENT        Titius)
           (STUDENTS
              (  (IS     Casper)
                 (PROJECT   P11) )
              (  (IS      Zorro)
                 (MARK      8 10)
                 (PROJECT P11 P12)))
           (TEXTBOOK     book1) )
        (  (SEMESTER_IS   1986b)
           (DOCENT       Titius)
           (STUDENTS

              (  (IS     Casper)
                 (MARK      0  )
                 (PROJECT   P11)))
           (TEXTBOOK     book1    ]
```

that is itself a coding in frame-form of the nested database relation shown in Fig. 4, and whose metarepresentation in CuProS is shown in the following code:

```
(setq  CuProS_of_course-database
   '( (root  ( COURSE BY_SEMESTER ) )
      (BY_SEMESTER
          ( n: semestral_chunk) )
             ; after "n:", a repeatable subtree-schema.

      (semestral_chunk   ; the attribute
          ( i: SEMESTER_IS   ; after "i:"
            DOCENT           ; identifies
            STUDENTS         ; the nameless
            TEXTBOOK ) )     ;       father.

      (STUDENTS (n: student_individual))

      (student_individual
          ( i:  IS
            MARK
            PROJECT  )  )
```

Judging from the nested relation in Fig. 4, it would appear to be the case that Khoshafian et al. [36] in 1986 were considering such complex objects, that

just one complex object or atomic value can appear as being the value of an attribute, but other approaches (including RAFFAELLO) allow sequences of values (possibly, even sequences of complex objects with the same schema) to appear as the instance of the same attribute. `{ }` denotes set objects, whereas `[ ]` denotes tuple objects (i.e., sets of attribute/value elements). The example is stated not according to the same representation method as the one proposed by Khoshafian et al. [36], but instead by means of an adaptation: sequence-valued attributes are included.

In the following code, we show a collection of binary relations, meant to be kept for indexing purposes, and that biunivocally correspond to the complex object shown in the previous codes in this subsection, as well as in Fig. 4:

```
C1_1:   {[ SURROGATE:     sur_1             ,
           COURSE:        Entomology         ]}
C1_2:   {[ SURROGATE:     sur_1             ,
           BY_SEMESTER:   { sur_2 , sur_5 }  ]}
C1_3:   {[ SURROGATE:     sur_2             ,
           SEMESTER_IS:   1987a              ]}
C1_4:   {[ SURROGATE:     sur_2             ,
           DOCENT:        Titius             ]}
C1_5:   {[ SURROGATE:     sur_2             ,
           STUDENTS:      { sur_3 , sur_4 }  ]}
C1_6:   {[ SURROGATE:     sur_3             ,
           IS:            Casper             ]}
C1_7:   {[ SURROGATE:     sur_3             ,
           MARK:          nil                ]}
C1_8:   {[ SURROGATE:     sur_3             ,
           PROJECT:       P11                ]}
C1_9:   {[ SURROGATE:     sur_4             ,
           IS:            Zorro              ]}
C1_10:  {[ SURROGATE:     sur_4             ,
           MARK:          { 8 , 10 }         ]}
C1_11:  {[ SURROGATE:     sur_4             ,
           PROJECT:       { P11 , P12 }      ]}
C1_12:  {[ SURROGATE:     sur_2             ,
           TEXTBOOK:      book1              ]}
C1_13:  {[ SURROGATE:     sur_5             ,
           SEMESTER_IS:   1986b              ]}
C1_14:  {[ SURROGATE:     sur_5             ,
           DOCENT:        Titius             ]}
C1_15:  {[ SURROGATE:     sur_5             ,
           STUDENTS:      sur_6              ]}
C1_16:  {[ SURROGATE:     sur_6             ,
           IS:            Casper             ]}
C1_17:  {[ SURROGATE:     sur_6             ,
           MARK:          0                  ]}
```

```
C1_18:  {[ SURROGATE:     sur_6          ,
           PROJECT:       P11             ]}
C1_19:  {[ SURROGATE:     sur_5          ,
           TEXTBOOK:      book1           ]}
```

This is a collection of binary relations equivalent to the complex object of Fig. 4. Whereas the latter is meant to be kept as being the primary copy, these binary relations, instead, are meant to be kept for indexing purposes. This approach is adapted from Khoshafian et al. [36], but we allow the attribute (other than the surrogate) in these binary relations to be set-valued, in order to cope with the primary copy, a complex object that admits as values even sequences of complex objects with the same schema. This extension requires that surrogates be easily identified as being such, at least whenever they appear as values of the second attribute (the one after `SURROGATE`) in the binary relation. Besides, relational operators should suitably account for the extension.

Let us go back to the subject of indexing frames of the kind from which RAFFAELLO carries out retrieval. One consideration about full indexing of the hierarchical structure of frame-instances is that it could prove too costly to keep whole paths from the root of the tree (frame-instance) until the leaf, to locate each occurrence: then, just a certain subtree could be pointed to, leaving it to the retrieval component, to search the subtree for the value requested. That is, such a choice for the index would ignore the internal structure of the subtree. This constitutes a trade off between memory efficiency and time efficiency requirements. Besides, once a frame-instance is loaded, pointers are valid if referring to nodes of its hierarchical structure, while the contents *files* in the database, even though the latter store frame-instances, could possibly be referred to by means of other kinds of coordinates (e.g., with a *concordance*, as if frames were text; this would allow to refer to *user comments* in the files of frames, without having to introduce `COMMENT` attributes in the frames, that would use up memory when loaded, or reconstruction-time to delete them on loading).

The metarepresentation in RAFFAELLO, the way it was developed, is a metarepresentation of attributes, and, therefore, a *metastructure of the type*, not of the *values*. In order to access the proper instances of frames, or at the very least, to access promising frame-instances, we would need indications on values, too.

However, *frame-indices* have their own limitations: they would point just to *disaggregated* values, i.e., to an index of values as found in terminal properties in the tree of attributes. This is not enough to enable a synthetic grasp of what is represented in subframes containing a given set of values. Owing to the hierarchical structure of frames, secondary indices as known from databases or concordances as known from full-text information retrieval may obviate to problems with search in trees, but, if bare values do not refer to clear contexts, entries in the index or the concordance would point to too many occurrences with respect to what queries intend, and thus concordances would just reduce — not eliminate — search in trees: sets of coordinates for a given entry in the concordance (a value in the frame) could be seen as a forest of hierarchical “skeletons”

drawn out of the frame-base, and search should check there whether the value occurrence is relevant or not.

Perhaps, semantically coherent portions of frame-instances — likened to chapters in a book — could be *labelled* with indicators of the topic, for insertion in the index. This is something familiar, by now, from Web technology. Webpages have lists of keywords, in order to attract search engines, and this is (apart from the misuse of keywords at webpages) like keyword-lists in scientific papers.

11 Syntactic Elements of CuProS, the Metarepresentation Language of Raffaello

11.1 Basic Terminology

Raffaello provides the possibility of representing nested relations inside the property-lists of Lisp atoms: the top-level attributes in the frame are properties in the property-list. Nested properties however obey a syntax defined in the metarepresentation language of Raffaello, and not belonging to Lisp. In fact, basically the syntax is independent of Lisp.

By *slot*, we are going to refer to any attribute in a tree of properties. By *facet*, we refer to any attribute together with its value. The value, which we term a *filler*, is either a terminal value or value-list (if the facet is a leaf in the tree of properties), or — if the facet is a nonterminal vertex in the tree of properties — a subtree (or even a list of subtrees). In practice it has never happened, in our frames, that the filler include a terminal value and also further levels of nested facets, as being brothers. We term a subtree of properties, a *subframe*.

Attribute-schemata are described — at a *meta-level* of representation (*metarepresentation*) — by means of a production system, where *productions* state legal genealogies of slots, and are augmented with *labels* that express particular conventions for conveying particular features (e.g., slotless repeatable subframes, as identified by the value of some child or grand-child facet defined in the schema itself).

The metarepresentation, according to our approach, consists of a data-structure storing a *production system*. Each *production* is a parenthesis including a *left-hand side* and a *right-hand side*. This set of rules constitutes a grammar, specifying valid structures of nested relations]for the given application.

Let us refer to the *left-hand side* of productions by the acronym *LHS*, and to the *right-hand side* of productions by the acronym *RHS*. The RHS lists attributes constituting the legal childset of the attribute that is the LHS. We sometimes refer to a *nesting level* by the name *generation* (i.e., a generation in a tree).

Each LHS is either a *facet-identifier*, or a parenthesis including several facet-identifiers. Facet identifiers are either *attribute-names* (and then, they do not include lower-case letters), or *implicit nonterminals* Implicit nonterminals are lower-case names that, in the metarepresentation, represent what in frame-instances are *slotless facets*. In Onomaturge, acceptation-chunks and rule-chunks (where rules are lexical derivation patterns) are examples of slotless

facets, that is, they are each a nonterminal nested parenthesis that does not begin by an explicit attribute-name.

Right-hand sides are either *structured*, or *simple* lists of facet-identifiers. Structured right-hand sides are subdivided by means of *labels* belonging to the reserved lexicon of the CuProS language.

The higher level of subdivision can be according to "ancestry", that is, to the nesting-context: the legal schemata of nesting inside the facet represented by the left-hand side are different according to the different ancestry possible for that left-hand side. Thus, the right-hand side is subdivided into parts constituted by an ancestry label (`f:` or `a:`), by a list specifying ancestry, and by a specification of legal sets of facet-identifiers that can be nested.

Such a specification can be decomposed further, into mutually exclusive portions (preceded by `x:`), which are simple lists of facet-identifiers that possibly are nested directly under the attribute (or slotless facet) in the left-hand side.

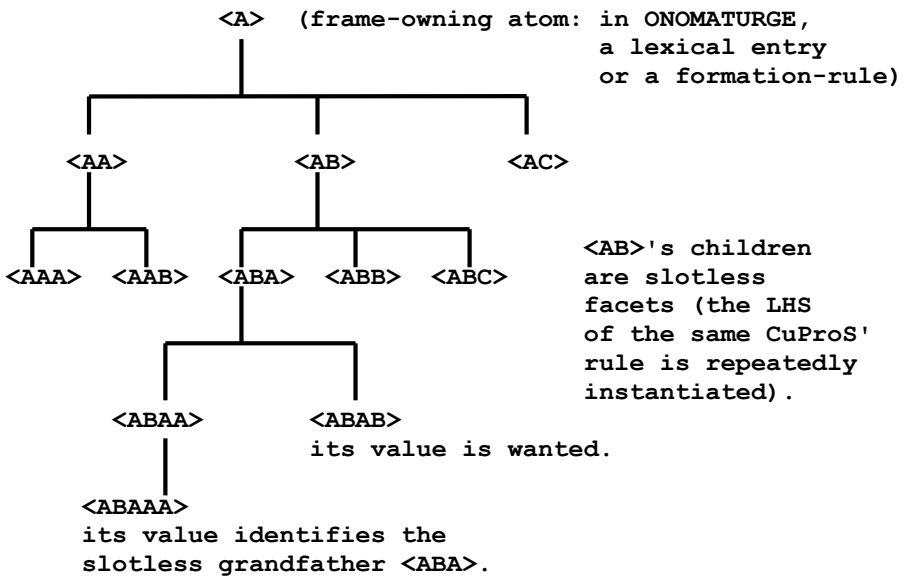


Fig. 5. Processing input facet-identifiers.

In such "simple" lists of facet-identifiers, some facet-identifiers can be preceded by a label describing some feature of the facet-identifier (as opposed to labels delimiting a set of facet-identifiers), such "one-follower labels" include `n:` as well as `c:` and `i:` (related to slotless facets), `g:` (related to local metarepresentations to be switched to), and so forth.

11.2 Parsing the Labels in the Metarepresentation

In Fig. 1, we have seen the general outline of the control flow in NAVIGATION. Fig. 5 illustrates actions taken by NAVIGATION in order to access the proper productions in the metarepresentation, whenever slotless facets (symbolised by F) are involved. The *key* of the single slotless facet is accessed inside an attribute that is nested (directly, or at a deeper level) inside the slotless facet itself. Fig. 6 and Fig. 7 show how processing switches between right-hand sides and left-hand sides in productions of the metarepresentation, that is, how productions are *chained*.

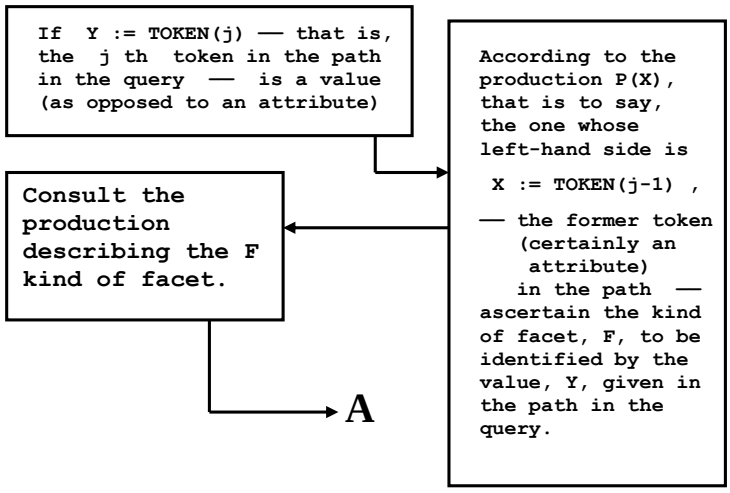


Fig. 6. The way left-hand sides of productions inside the metarepresentation are handled when the latter is consulted, in order to chain productions

Fig. 8 and Fig. 9 show how labels found inside (possibly structured) right-hand sides guide the processing. That way, whenever the same attribute-name (e.g., `ARE`) is found in different nesting contexts (*ancestry contexts*) inside frames, then according to its production in the metarepresentation, the proper childset of nested facets (attributes, or slotless facets) is selected.

At an inner level inside right-hand sides, possibly `x:` labels (for "exclusive or") delimit mutually exclusive sets of facet-identifiers. Labels with different priorities are processed in different points in the schema, as indicated in Fig. 8 and Fig. 9. See further, at the end of Subsec. 11.4, the two parts in Fig. 10 and Fig. 11. Boxes with a grey southeast corner were not implemented in the 1988 version in LISP of NAVIGATION.

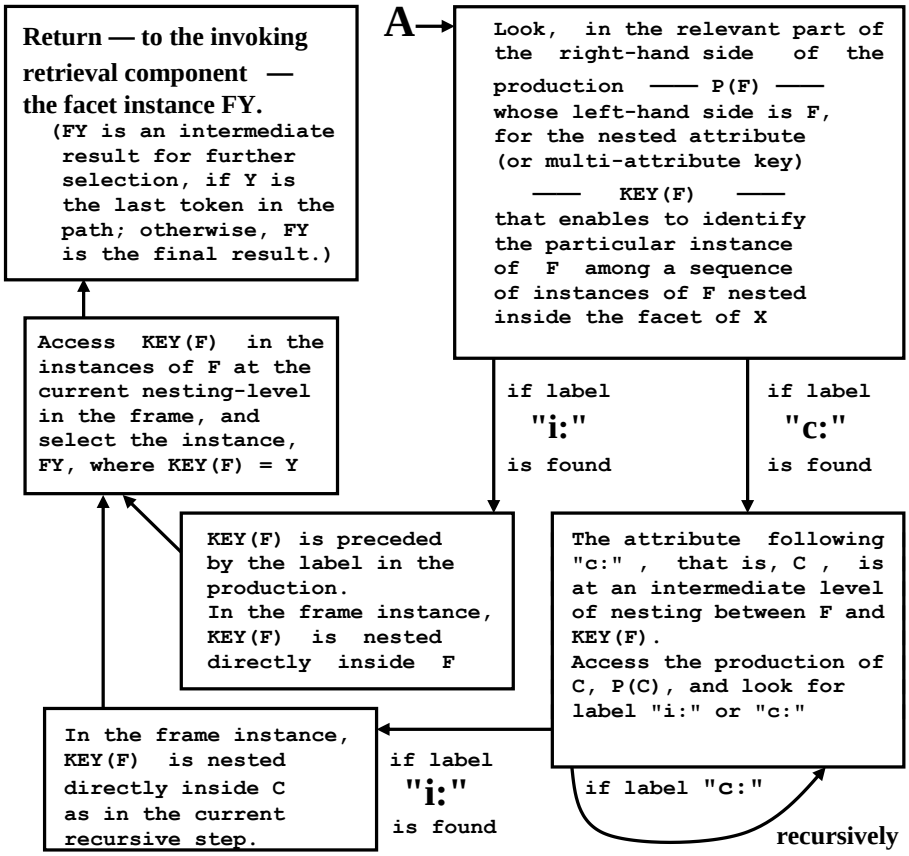


Fig. 7. The way left-hand sides of productions inside the metarepresentation are handled when the latter is consulted, in order to chain productions

11.3 A Sample Metarepresentation. Part I

In Subsec. 8.2, we have considered how to formulate queries containing variables, in the path leading to what is sought. Such queries were exemplified on a rather simple lexical frame, a simplification of the kind found in the database of the Onomaturge expert system for word-formation. The following code shows a metarepresentation of the nesting structure for such frames. This will enable us to get a taste of what the labels in a CuProS metarepresentation mean, before we turn to explaining the syntax more systematically.

Consider, first of all, that `setq` is the assignment operator in Lisp. Considering the attribute name `MORPHOLOGICAL_CATEGORY` bear in mind that `MORPHO_CAT` is a synonym for that name. Such synonyms of attribute names are declared in a separate data structure, not the CuProS ruleset we are considering.

```
(setq  metarepresentation_1
```

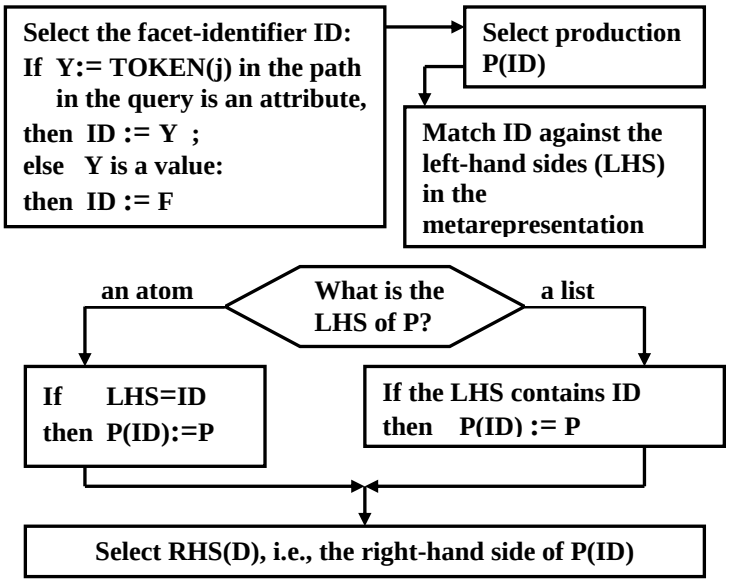


Fig. 8. Manipulation by RAFFAELLO as triggered by syntactic features detected inside the metarepresentation, starting with labels whose priority is highest. These boxes were already implemented in the LISP version of NAVIGATION as developed by Nissan by 1988, except — in Fig 10 and 11 — boxes with a grey bottom right corner.

```
'(
   (pointer_atom        ( POINTER ) )

   (atom                ( INFLECTION
                          MORPHOLOGICAL_CATEGORY
  ACCEPTATIONS
  ASSOCIATION_OF_IDEAS
  FRAME_GENERATED_ON
  CODED_BY              ) )

   (MORPHOLOGICAL_CATEGORY   ( x: terminal_list
                               x: See_ACCEPTATION ) )
```

Bear in mind that the label `x:` stands for "exclusive or". Also, bear in mind that it is possible to add comments to a CUPROS metarepresentation: this is done in what remains of a line after a semicolon, because this is how in LISP one adds comments.

Pay attention to the syntax of the attribute `ARE` if found nested inside `ACCEPTATIONS`. `ARE` is a *polysemous* attribute-name, that in different contexts refers to different slots. That is to say, it may happen to find `ARE` nested in some slot, i.e., under some attribute, different from `ACCEPTATIONS`. Parsing dis-

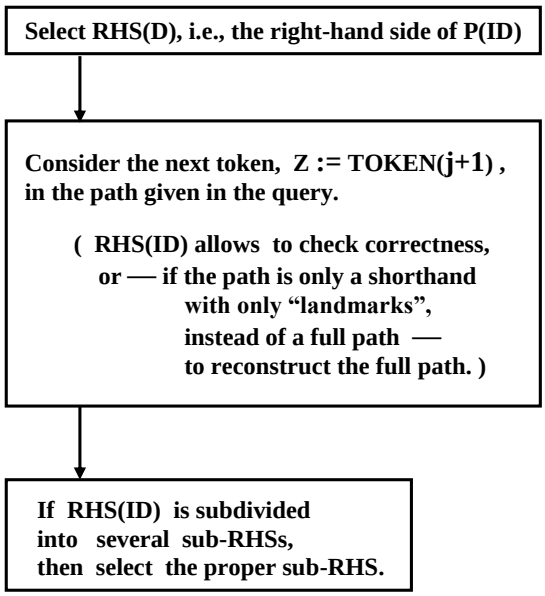


Fig. 9. Manipulation by RAFFAELLO as triggered by syntactic features detected inside the metarepresentation.

ambiguates polysemous slots according to their ancestry in the property-tree. The label `f:` is followed by a parenthesis stating ancestry (in our present example, the ancestor `ACCEPTATIONS`), and then by a separate childset of attributes. When found nested under `ACCEPATIONS`, the attrubute name `ARE` is followed by a sequence of facets with no attribute-name at the their beginning, and whose schema is the one stated in the production whose left-hand side is the string `single_acceptation` — but then also consider that it may happen to find `ARE` nested under the attribute `ASSOCIATION_OF_IDEAS` (in which case, `ARE` introduces a list of at least one "association chunks"), or then `ARE` may be found nested under either `SYNONYMS` or `NEAR-SYNONYMS` and this in turn involve two alternative syntactic features. The first option is captured by `n: syno_chunk` and this means that one finds under `ARE` a sequence of at least one "synonym chunk". That is to say, a sequence only of chunks each describing a single (near-)synonym, and sharing the nesting-schema, so an attribute-name would be superfluous, and this is why the "synonym chunk" is slotless, i.e., it is not preceded by an attribute name. The alternative option is captured by the label `x:` being followed by both `n: SINGLE_SYNO` and `n: SYNO_CLUSTER`. The code of the metarepresentation rules for `ACCEPTATIONS` and for `ARE` is as follows:

```
(ACCEPTATIONS        ( ARE ) )


(ARE   ( f: ( ACCEPTATIONS )
```

```
            n: single_acceptation

         f: ( SYNONYMS  NEAR-SYNONYMS )
            x:   n: syno_chunk
            x:   n: SINGLE_SYNO
                 n: SYNO_CLUSTER

         f: ( ASSOCIATION_OF_IDEAS )
            n: association_chunk
)     )
```

The reason why under the label `x:` one finds two attributes, each preceded by the label `n:` is that a sequence where two kinds of facets may occur, so we must name the proper attribute.

11.4 A Sample Metarepresentation. Part II

Let us turn to the metarepresentation production that describes the valid structure of an "association chunk". There are alternatives, so we use the high-priority label `x:` followed by one of the options. Therefore, we find indeed an option whose syntax is captured by `x: i: IS` and this means that one finds in the slotless association chunk just the key for a siungle attribute, and this key value is preceded by the attribute `IS`. Otherwise, we have the option captured by the syntax `x: is: TERM` and this means that at most two more attributes may appear in the key, and that this is signalled by finding (instead of `IS`) the attribute name `TERM`, along with one or two more attributes. Which ones? This is captured by the syntax `j: KEYWORD` and this means: a keyword disambiaguating the meaning of the term. But the syntax `j: SCRIPT` tells us that we may also find an identification of the script of the keyword. We may have chosed instead to identify the language, or the spelling of a language, instead of the script. Envisagfing having several script was premature in 1988, but is now quite a possibility, with the spread of Unicode and of XML, that may accommodate various scripts thanks to Unicode.[21]

[21] At a later time, for a while, Nissan was member by invitation of a Unicode definition committee in England, with the role of making proposals concerning the Hebrew script, and for that purpose, he prepared specifications that included typographical needs from various contexts of use of the script, from scholarly transcriptions, to lexicographic practice as being instantiated in various Israeli dictionaries, to the practice of typesetters of devotional literature. For example, an early Israeli dictionary of Amharic had used a reversed *qamats* Hebrew diacritical mark in order to render what romanisation expressed by means of a reversed `e` and Nissan included that. Philological works from Israel that need to account for Syriac data, transliterate them into the Hebrew script letter by letter, but as Syriac scriptorial practice is to express a plural word by writing the word in the singular and adding two dots side by side over the end of the word, also Israeli transliterations from Syriac do likewise, and Nissan included that diacritical mark. Medieval Judaeo-Arabic, and the

usual scripts of the various modern Jewish vernaculars (e.g., modern Judaeo-Arabic, Judaeo-Spanish, and Judaeo- whatever, apart from Yiddish that was dealt with in Unicode separately) have their own needs in terms of diacritical marks.

Besides, the history of diacritical marks for the Hebrew scripts includes not only the Tiberian system currently in use, but also the Babylonian system, still in use by some philologists, and that survived in the Yemeni Jewish community. There also existed the simpler Palestinian System. The Babylonian and Palestinian systems added marks above letters, in between, whereas the Tiberian system mostly adds marks under each letter. Moreover, the history of the Tiberian system itself gives rise to Unicode needs: the late linguist Shlomo Morag pointed out, as can be seen in his collected papers, that historically some grammarians also envisaged the semivowel *ḥaṭaf-ḥiriq*, just as there is a diacritical mark for the semivowels *ḥaṭaf-qamats*, *ḥaṭaf-pataḥ*, and *ḥaṭaf-segol*. Nissan pointed out that much to the committee. The popular Megiddo English-Hebrew dictionary [46] introduced special diacritical marks for transliterating English into the Hebrew script. Devotional literature has specific needs, e.g., with the last letter of the Tetragrammaton containing in smaller type the word indicating how that divine name needs to be pronounced. Whereas this is traditional in typesetting, there is a modern practice (promoted by one publisher of prayer books, Moreshet) of distinguishing between the mark for a *qamats* when it stands for /a/ and the same mark when is stands for /o/ (in the latter case, it is called a "small *qamats*", but is usually printed the same; the innovation was in making the "small *qamats*" look different, by making a vertical line longer). Other *desiderata* include the bent-neck and the beheaded versions of the letter *lamed* (traditional in Hebrew typesetting, historically out of the need to reduce spacing between lines, but the practice still exists in prayer books), the ligature of the letters *aleph* and *lamed* (a requirement for the devout), and the variant of the letter *vav* cut in the middle (a famous *hapax* from the Pentateuchal weekly reading of *Phineas*).

All of this was pointed out to the committee. Moreover, there are the Masoretic marks, indicating prosody in the Hebrew Bible. Whereas the committee was initially eager to listen, eventually nothing was taken on board, with no explanations given, and no further contact from the committee. A professor was in the committee, who is known for his publicly expressed dislike for the Hebrew script. His minimalistic preferences prevailed. This is apparently why the Hebrew script is not well subserved by Unicode, with some glaring and dismal inadequacies. This did not need to be the case. By contrast, the Arabic and Arabic/Persian script is subserved beautifully (evidently because the respective committee did an excellent job), with diacritical signs subserving niche communities of users, and some typographical signs traditional from devotional literature (a cultural need that is as legitimate as any other). Extensions of the Arabic script from Asia are included, but one would also like to see such extensions from Africa (where modified Arabic letters appear in at least one textual genre: magic), as shown in his papers from 1967 and 1968 that also described autonomously developed West African scripts [13].

Dalby enumerated so-called *sub-Arabic* scripts with the function of magico-cryptic alphabets. There are several of them: in the Hodh region of the south-western Sahara, eleven secret alphabets were recorded. ([57, 56] and [13, p. 172, fn. 45]). Apart from the *al-Yāsīnī* alphabet, other sub-Arabic scripts include the *al-Qalfatīrī, al-Falaluīsī, al-Ṭabī'ī* and *al-'Ajamī* alphabets, these being alphabets recorded by Monteil [57], the *'Ibrāniyya* ("Hebrew") alphabet recorded by Monod [56], the "amulette" alphabet of Mauchamp [53, p. 208], and Ibn Waḥshiya's collection of 84 alphabets

Moreover, whereas by now we are acquainted with the `x:` label for "exclusive or", we are going to see an example of the `y:` label as well. The `y:`-preceded segment lists optional content that may appear in any of the `x:`-preceded segments (that, instead, are mutually exclusive).

```
(association_chunk   ( x:  i: IS
                       x:  i2: TERM
                           j:  KEYWORD
                           j:  SCRIPT
                       y:  FOR_WHOM
)                    )
```

Under the attribute `FOR_WHOM` one may find one or more slotless chunks. Inside each such chunk, it is expected to find the attributes `FOR` and `LIKELIHOOD` (the latter expresses a numerical range, that may be, e.g., hyperfuzzy). Pay attention to the syntax `i_inc: FOR` that conveys the prescription that `FOR` contains a variant of a key, allowing inclusion: for example, the values may be `mathematicians` or `computer-scientists` or both. It is valid to list either or both of them, if under `FOR` we find both of them listed.

```
(FOR_WHOM        ( n: for-whom_chunk ) )

(for-whom chunk  ( i_inc:  FOR
                   LIKELIHOOD
)                )
```

In the following, we make use of the `c:` label. In the example, it precedes the attribute name `MEANING` and this means that inside the slotless chunk of a single lexical acceptation, one may find `MEANING` nested as intermediate: it is inside the facet to be identified by the key, but the key itself is nested deeper.

```
(single_acceptation  ( c: MEANING
                       CONNOTATION
                       RELATED_TO
```

(the latter is controversial). Ancient Epigraphical South Arabian characters occur in an early 19th-century navigational guide from Kuwait, and were apparently used for magical nautical calculations. Also note special characters used in some Arabic books of magic ([pp. 150–167] [113] and [15, pp. 158–159], cited in [13, pp. 173–174]), as well as Arabic cryptography, e.g., the secret writing ascribed to the secretary of the Sultan Aḥmad al-Manṣūr of Morocco (1578–1603) ([11], cited in [13, p. 172, fn. 45]). "One of the scripts from the Hodh which is particularly rich in non-Arabic characters is the *al-Yāsīnī* alphabet, named either after its inventor (the mage Yāsīn?) or after the two 'mystic' letters which open the *Yāsīn* sura of the Koran (sura 36)" [13, p. 172]. (Nissan is thankful to Dr. Dorit Ofri of Geneva for making him aware in 1995 of David Dalby's papers [12, 13] and supplying him with photocopies, while she was preparing her Bern thesis about the Vai culture of Liberia, which has a peculiar alphabet of modern origination.)

```
                          FREQUENCY_SCORE_AMONG_ACCEPTATIONS
)                       )


(MEANING                ( i: KEYWORD
                          TEXTUAL_DEFINITION
                          DENOTATIONAL_FEATURES   )  )

(CONNOTATION        ( POS/NEG
                      CONNOTATIONAL_FEATURES   )  )

(POS/NEG            ( x:  terminal_singleton
                      x:  AT_FIRST
                          IN_CONTEXT          )   )
```

We are going to encounter the `a:` label. It is a high-priority label (like `f:`), and its syntax only differs from that of `f:` in that what follows `a:` needs to be enclosed by a parenthesis. Both those labels are about ancestry. The following means that the attribute `IN_CONTEXT` may be found nested inside the `POS/NEG` facet inside the `CONNOTATION` facet, or then alternatively one may find `IN_CONTEXT` nested directly under the attribute `BLA_BLA_BLA`.

```
(IN_CONTEXT         ( a:  ( POS/NEG CONNOTATION )
                          IMPRESSION_IS
                          IF_CONTEXT_IS
                      f:  ( BLA_BLA_BLA )
                          n: bla_bla_bla_chunk
)                   )
```

Here is the production for the `RELATED_TO` facet inside the lexical frames of ONOMATURGE. In a separate structure, the attribute name `RULES` is declared to be synonymous with `FORMATION_RULES` (as those rules are lisnguistic rules for word-formation. In Semitic languages, *free-place formulae* are a typical kind of derivational pattern).

```
  (RELATED_TO           ( RULES
                          PECULIAR_FACETS
                          LIKELY_CONTEXT
                          SYNONYMS
                          NEAR-SYNONYMS    )  )

  (RULES                ( SUFFIXES
                          PREFIXES
                          FREE-PLACE_FORMULAE
                          REDUPLICATIVE_PATTERNS
                          COMPOUND-FORMATION_PATTERNS ))
```

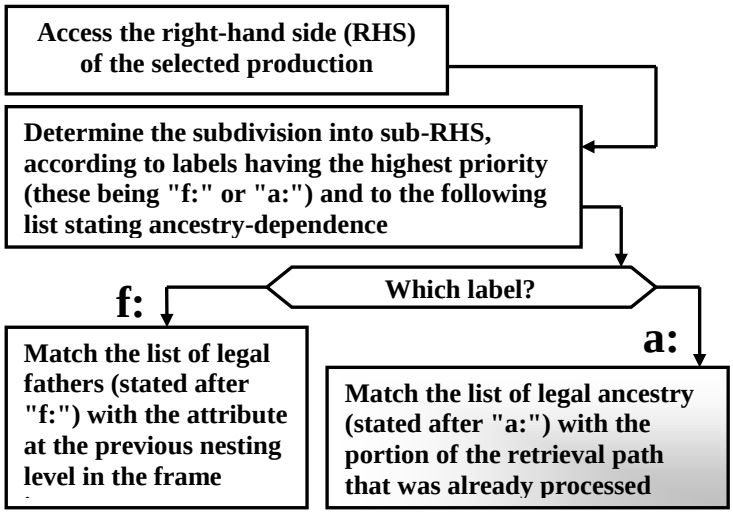


Fig. 10. Manipulation by RAFFAELLO as triggered by syntactic features being the `f:` label or the `a:` label.

Fig. 10 shows the control flow of interpretationvwhich enables to handle appropriately the `f:` label and the `a:` label. Fig. 11 shows the control flow of interpretation which enables to handle appropriately the `x:` label, the `n:` label, and the `g:` label. Boxes with a grey southeast corner were not implemented in the 1988 version in LISP of NAVIGATION.

11.5 A Sample Metarepresentation. Part III

The following production stands out, because its left-hand side is multiple: it is a list of five attribute names. This is equivalent to having five separate productions with the same right-hand part, and *atomic* left-hand sides.

```
(  ( SUFFIXES
     PREFIXES
     FREE-PLACE_FORMULAE
     REDUPLICATIVE_PATTERNS
     COMPOUND-FORMATION_PATTERNS  )
   ( n: formation-rule_chunk )  )
```

We may have listed these five rules instead:

```
(SUFFIXES
          ( n: formation-rule_chunk )  )

(PREFIXES
          ( n: formation-rule_chunk )  )
```

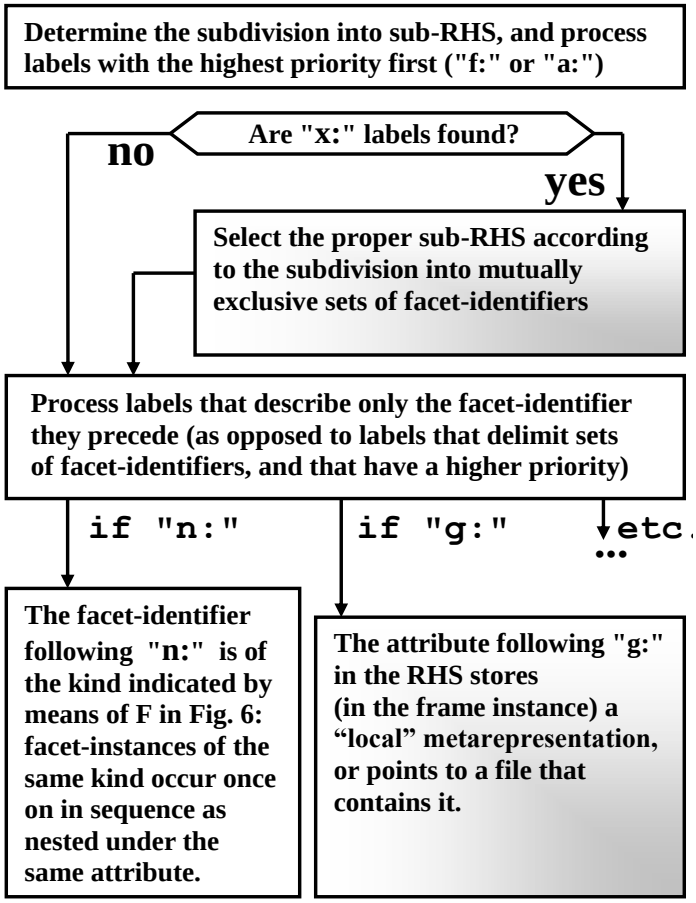


Fig. 11. Manipulation as triggered by syntactic features being the `x:` label, or the `n:` label, or the `g:` label.

```
(FREE-PLACE_FORMULAE
          ( n: formation-rule_chunk )  )

(REDUPLICATIVE_PATTERNS
          ( n: formation-rule_chunk )  )

(COMPOUND-FORMATION_PATTERNS
          ( n: formation-rule_chunk )  )
```

The following rule prescribes the valid structure for a slotless "formation-rule chunk". The `i2:` label along with the `j:` label prescribe that the key has at least one attribute (it is `IS`), and at most further two attributes, which is useful in case rule-names are ambiguous.

```
(formation-rule_chunk      ( i2: IS
     RELEVANCE_ORDINAL
     j: INFLECTION_PARADIGM
     j: MORPHO_CAT_IS       ) )

(INFLECTION_PARADIGM  ( FORM_IDENTIFICATION
INFLECTED_FORM_IS     ) )
```

We allow the global metarepresentation to be supplemented with some local metarepresentation, either stored directly in a part of the lexical frame, or pointed to from that same facet. This is enabled by using the `g:` label, as already mentioned at the end of Subsec. 10.1 above. The following rule states that under the attribute `PECULIAR_FACETS` one should expect to find a local metarepresentation stored or pointed to from inside the facet whose slot is `LOCAL_METAREPRESENTATION`. By the reserved word `etc` in the context of this kind of rule, we means: further attributes, as described in the local metarepresentation.

```
(PECULIAR_FACETS     ( g: LOCAL_METAREPRESENTATION
                       etc
)                    )
```

We have seen earlier the rule for `ARE`. But here in the following we have rules that state that the structure of `SYNONYMS` and of `NEAR-SYNONYMS` is just that we should expect to find the `ARE` facet nested inside them, and to find nothing else (i.e., no sibling facet of `ARE`) nested inside them. Morevoer, we give the rule prescribing the valid structure of the slotless "synonym chunk". In each such chunk, we may find the key, and moreover the non-key facet whose attribute name is `RELATIONSHIP`. The value of the key is found in the facet whose attribute is `TERM_IS` but as this may not be sufficient for unambiguous identification, the `i1:` label stipulates that the key may be augmented by at most one other key facet, this being (as the `j:` label informs us) the facet whose attribute is `KEYWORD_IS`.

```
(SYNONYMS            ( ARE ) )

(NEAR-SYNONYMS        ( ARE ) )

(syno_chunk       ( i1: TERM_IS
                    j:  KEYWORD_IS
                    RELATIONSHIP    )  )
]
```

The symbol `]` is standard syntax in Franz Lisp for closing all open parentheses.[22] This is why this symbol is found at the end of CuProS metarepresen-

[22] Wilensky's [111] *LISPcraft* is a book about Franz Lisp, whereas his [112] *Common LISPcraft* is a book about the Common Lisp dialect of the Lisp programming language. Franz Lisp was also described by Larus [40].

tation, as well as at the end of the lexical frames of ONOMATURGE, unless one wants to close down all levels of parenthesis one by one.

11.6 Slot-Synonymy, and Contextual Disambiguation of Slot-Polysemy

Our metarepresentation syntax allows *slot-synonymy.* That is to say, you may refer to the same attribute by different names; for example, `MORPHO_CAT` is (by convention in the ONOMATURGE expert systems) a legal synonym of `MORPHOLOGICAL_CATEGORY`. Besides, the metarepresentation syntax allows *slot-polysemy.* That is to say, the same attribute-name may refer to slots that in different contexts, may father different subframe-schemata (not just variants of RHS meant to describe the same kind of knowledge).

For example, ARE is a dummy slot, that in the frames of ONOMATURGE may be the child of `ACCEPTATIONS`, or of `SYNONYMS`. Therefore, `ARE` is a *polysemous* attribute-name, that in different contexts refers to different slots. NAVIGATION (i.e., the component of RAFFAELLO that is guided by the metarepresentation), according to its its fuller specification, disambiguates polysemous slots according to their ancestry in the property-tree. In such cases, the metarepresentation syntax subdivides the RHS of the production whose LHS is S, into parts headed by the label `f:` or `a:` followed by a parenthesis stating ancestry, and then by a separate childset of attributes (an `f:`-*sub-RHS* body, or an `a:`-*sub-RHS* body).

The labels `f:` and `a:` differ according to the conventions of the ancestry-list following the label, and preceding the body. In the case of the `f:` label, the parenthesis is a list of slots possibly fathering the LHS in the case where the legal childset of the LHS is as stated in the body. Instead, in the case of the `a:` label, the parenthesis is a list of lists of slots: each innermost list states a possible father, its own father as well, and so forth. That is to say: `a:` is useful whenever ambiguity involves more than a single generation.

Another case (far more frequent) is possible: instances — S1, S2, ... — of the same slot S can happen to be children of different slots (say, P, Q, and R), and yet S1, S2, etc. refer to the same meaning of S, and their legal childset is as stated, in the production of S, for the same meaning of S. The metarepresentation syntax expresses this by having S listed in different productions with different LHSs, while in the production whose LHS is S, no `f:` or `a:` label is included (that is: S is *monosemous,* unambiguous), or otherwise, P, Q, and R belong to the same ancestry-list, so the same `f:`-sub-RHS or `a:`-sub-RHS is referred to by S1, S2, and so forth.

Metarepresentation labels are in lower case, and usually end by a colon. The syntax of the label `a:` is just like the syntax of `f:` except that `a:` requires one more level of parenthesis, in the *ancestry-statement.*

`f:` `(` <*slot*> `)` is equivalent to: `a:` `( (` <*slot*> `) )`

`f:` `(` *A* *B* `)` is equivalent to: `a:` `( (` *A* `) (` *B* `) )`

By contrast, neither

```
a:  ( ( X  Y ) )
```

nor

```
a:  ( ( X  Y )  ( Z ) )
```

can be expressed in terms of the `f:` option.

`a:` is suited when stating just the father would be ambiguous. Then, the ancestry is stated in a list, beginning by the father, followed by the grandfather, and so forth, until no more ambiguity remains.

11.7 More Conventions

The kind of *sub-RHS* as distinguished according to ancestry, is not the only kind of RHS subdivision allowed, even though it ranks as having the highest precedence among *RHS-subdividing labels.* A slot may father different attribute-schemata in the same context, that is, as having the same ancestry itself: our metarepresentation syntax envisages RHSs that are possibly subdivided into several alternative sub-RHSs, even nested.

At the same level of subdivision of an RHS, both `f:`-sub-RHSs and `a:`-sub-RHSs are allowed to appear. At inner level, the sub-RHS body may include further subdivisions as stated by means of a lower-priority RHS-subdividing label, such as `x:` (See in Subsec. 13.1).

RHSs list possible children of the LHS: not all of the slots listed *must* appear in actual instances. This is, after all, intuitive for one used to formal grammars. It stands to reason that flexibility in nesting is subserved by a spectrum of alternatives being envisaged.

According to our conventions about the syntax of frames (our nested relations), slots (attributes) are in upper case, and possibly include underscores or digits as well. Values in frame instances — not in the metarepresentation — are either numbers, or special-symbol strings, or strings which contain lower-case letters and possibly also digits, special symbols, or upper-case letters. Values are not listed in the kind of metarepresentation we are describing.

In the metarepresentation, *slots* are only a particular case of *facet-identifier.* A facet-identifier is *nonterminal,* iff beside appearing inside some RHS or RHSs, it is also the LHS of some production (a *different* production, as we do not admit cyclical productions).

11.8 Slotless-Facet Identifiers, and Repeatable Facet-Schemata

There may be, in a metarepresentation, such nonterminal facet-identifiers that are in lower-case. This convention is needed, in order to represent, in the metarepresentation, such *slotless facets* as occasionally appear inside frames. Why do frames include facets with no expressly named slot?

Slotless facets are *repeatable* under the same father (that is: the sub-RHS body that is nested deepest, in the RHS subdivision into alternative sub-RHSs), and if that father does not have any other kind of child facet-identifiers, then

putting a slot would not bear identifying information. However, not every repeatable facet is slotless: slots may be put there even if unnecessary, and moreover, if a repeatable child is not a singleton child-set in the considered RHS, then it should have a slot. That slot would not distinguish facet instances with the same slot, but it would distinguish them from children with any other slot.

`n:` precedes a facet-identifier that, as instantiated in the same facet whose slot is the LHS, can be repeated there several times. This is a frequent case in the metarepresentation. Most often, *repeatable-facet identifiers* (that is, facet-identifiers preceded by `n:`) in the metarepresentation are in lower-case, that is, they identify a *slotless* facet. Bear in mind that `n:` is not an RHS-subdividing label.

In fact, an *RHS-subdividing label* does not head a body. A *sub-RHS body* includes one or several facet-identifiers, and ends as soon as another RHS-subdividing label with the same precedence is met. (For example, an `f:`-sub-RHS body or an `a:`-sub-RHS body ends as soon as the successive *ancestry-disambiguating label* (that is, `f:` or `a:`) is met.

By contrast, `n:` (like several other labels) does not head a sub-RHS body, but it qualifies only the facet-identifier being its immediate successor. We say, then, that `n:` is a *one-successor label.*

11.9 Ancestor-Identifying Descendants of Repeatable Facets: Attributes Storing Identifiers of Subframe-Instances

`i:` is one more one-successor label. In order to enable the identification of the proper instance of a certain repeatable facet — when a path is given as input to the `navigate_and_retrieve` function of NAVIGATION, or when a path is the value in a `PATH` or `POINTER` facet — each repeatable-facet identifier has a *father-identifying child* (or an *ancestor-identifying descendant* in general, if the `c:` label is also used: see below).

A *father-identifying nonterminal* always corresponds to a *slotted* facet (as opposed to a *slotless* facet). In the path given as argument to the function `navigate_and_retrieve` the filler of the *father-identifying facet* (or *ancestor-identifying facet*) fits in the place of the identified father (or ancestor) — if the latter is a slotless facet — or follows its slot, in the path (if the father is an instance of a slotted repeatable facet).

Every *ancestor-identifying descendant* is preceded by `i:` in the RHS where it appears. Let us consider the case where A1 is a *repeatable-facet identifier,* and A2 is its *ancestor-identifying descendant.* Then, A1 is preceded by `n:` in the RHS(s) where it appears. Besides, if A2 is a direct child of A1 (that is to say: in frames, A2 is nested immediately under A1), then, in the production whose LHS is A1, the RHS lists A2 as preceded by the label `i:`

However, it may also happen that the *ancestor-identifier* A2 is not an immediate child of the ancestor A1 it identifies. In the lexical frames of Nissan's ONOMATURGE expert system, this happens with the single acceptations, that are identified by the value of the facet `KEYWORD.` The latter is a child of the

facet `MEANING,` that in turn, is a child of the facet storing the single acceptation. Then, A2 is the facet `KEYWORD,` whereas A1 is the slotless-facet identifier `single_acceptation` under `ARE` — which itself is under `ACCEPTATIONS.`

The metarepresentation syntax provides the one-successor label `c:` that in the production whose LHS is `single_acceptation` precedes the attribute `MEANING` as listed in the RHS. Then — in the production whose LHS is `MEANING` — the RHS lists the attribute `KEYWORD` as being preceded by the label `i:`

Visiting the metarepresentation, the *inter-production path* between *single_acceptation* (as being preceded by `n:`) and the RHS where `KEYWORD` is listed (as being preceded by `i:`), it deserves notice that the *signal* represented by the label `c:` occurs only once: `KEYWORD` is a *grandfather-identifying grandchild.* Ancestry more remote than that could correspond to a longer inter-production path, along which `c:` would appear several times.

11.10 Multi-Attribute or Multi-Facet Identification-Keys

A rather infrequently occurring label is `i`*<digit>* `:` of which an example is `i2:` The label `i`*<digit>* `:` — also a one-successor label for ancestor-identification — states that the slot stated as following it in an RHS, is a component in an *identification-key,* but that it is not necessarily a *singleton key:* at most *<digit>* further attributes could appear in the key.

Let us explain that by means of an example, drawn from the ONOMATURGE expert system for word-formation. In ONOMATURGE, most word-formation rules are templates, where an exploded stem fits. This is typical of the morphology of Semitic languages. The lexical root is a sequence of radical consonants, that generally speaking need not be consecutive, and they fit inside empty "plugs" interspersed inside the template, that apart from the plugs also includes substrings of vowels and possibly formative consonants that are not part of the lexical root.

This corresponds, e.g., to suffixes in English: the suffix *-able* means 'that can be done' whereas the equivalent derivational template in Hebrew is /Paˆ iL/, where P, ˆ, L are metasymbols representing the consonants of a "triliteral" root consisting of three radical consonants. This way, by inserting the letters of the root $\sqrt{\text{lmd}}$, (that denotes 'learn') in the derivational template /Paˆ iL/, one obtains the adjective /lamid/, i.e., 'learnable'. Once you obtain a word from the word-formation template, that word is in its lexematised, i.e., uninflected form, that is to say, this is the form the way you could insert it as a headword of some entry in the dictionary. Now, let us consider inflected forms, out of the inflected paradigm of the considered word-formation pattern from which the word derived. For example, /Paˆ iL/ is the uninflected pattern, whereas its plural form is /Pˆ iLim/, as instantiated in the word /lmidim/ for the sense 'which are learnable'.

Sometimes, the uninflected identifier of a word-formation pattern happens to "homonymously" refer to different patterns (possibly with the same meaning) that are inflected according to (at least partially) different inflection paradigms.[23]

[23] We explain this in some detail in the Appendix A to the present article. In this footnote, suffice it to concisely explain that this is the case of the uninflected pattern-

One such derivational pattern is /PaˆoL/. One variant of it has the plural /PaˆoLim/, and another variant has the plural /PaˆoLot/.

Then, let us think about any facet (for example, in the `RULES` subtree, in the subframe of an acceptation inside a lexical frame) that has to list /Paˆ oL/, as being associated with only one of the two inflection paradigms. The *single_rule* slotless facet would then contain an identification-key, as well as non-key attributes. For unambiguous pattern-names, it would be enough to state:

```
( (IS  <pattern-name> )
  (RELATED_INFO  <dependent information> )  )
```

where the nested facet `(IS <pattern-name>)` is the *father-identifying facet.* In the metarepresentation, the corresponding productions are:

```
(PATTERNS    ( n:   pattern_with_information ) )
(pattern_with_information   ( i:   IS
                              RELATED_INFO ) )
```

That way, `IS` is a *singleton key*, that is, a key including a single attribute. Instead, in the case of the `/Paˆ oL/` derivational pattern of Hebrew, the inner facet `(IS Paˆ iL)` is not enough for identification, inside the *single_rule* slotless facet. Correspondingly, in the metarepresentation, `IS` alone is not enough, in the key. Therefore, the productions resort to the label indicating the poisibility of an extended key being used:

```
(PATTERNS    ( n:    pattern_with_information ) )
(pattern_with_information  ( i2:    IS
                               RELATED_INFO
                          j:  INFLECTION_PARADIGM
                             j:  MORPHO_CAT_IS
 )                            )
```

By this syntax, we mean that the key may include *at most* $_{<}$*digit*$_{>}$ +1 (here: 3) facets, that in our example are: `IS`, `INFLECTION_PARADIGM`, and `MORPHO_CAT_IS`.

---

identifier /Paˆ oL/, of which one variant shortens the first syllable's *a* in the inflected forms, and another variant retains it instead. The latter is not as productive in the Hebrew lexicon, i.e., its derivatives are rarer, and moreover, in Israeli Hebrew no new derivatives are formed by using it any more.

It was typically productive in the Roman era, because of the influence of Aramaic, and by applying it, masculine nominalised participles are generated. Moreover, both the *a*-shortening /Paˆ oL/ and the *a*-retaining /Paˆ oL/ have two subvariants, according to whether the ending of the plural is *-im* or *-in*, or is *-ot* instead.

Of the *a*-retaining /Paˆ oL/, there exists in Hebrew a pattern whose plural form is /PaˆoLot/, but there also exists in Hebrew the *a*-retaining pattern /Paˆ oL/ whose plural form is /PaôLim/, and which is seldom instantiated, and most examples of which belong to Tannaic Hebrew (also known as Tannaitic Hebrew or Middle Hebrew, from the Roman age), and are often obsolete in contemporary Hebrew.

Instances could be those included in the following code, being an example of different single-rule subtrees. They are legal inside the subtree rooted in RULES, inside an acceptation of different lexical lexical entries. The pattern-name /Pa^oL/ is ambiguous (because it is associated with two different inflection paradigms, whose salient feature is the form of the the plural). Therefore, a multi-attribute key is stated.

```
( (IS  Pa^iL)  (RELATED_INFO  <dependent information> ) )

( (IS  Pa^oL)
  (INFLECTION_PARADIGM  ( (FORM_IS Pa^oLot)
                         (FORM_CLASSIFICATION plural)
  )                     )
  (RELATED_INFO  <dependent information> )
)

( (IS  Pa^oL)
  (INFLECTION_PARADIGM  ( (FORM_IS Pa^oLim)
                         (FORM_CLASSIFICATION plural)
  )                     )
  (RELATED_INFO  <dependent information> )
```

We have used, in the previous code, the `j:` label, a one-successor label that precedes each legal *key-completion* attribute. In the code given earlier, we used the slot INFLECTION_PARADIGM (which is the slot of a nonterminal facet), in order to obtain a 2-facet key (including the attributes inside the INFLECTION_PARADIGM nested facet belonging to the key).

Conceptually, the same pair { IS, INFLECTION_PARADIGM } could even not be enough, as words of some different morphologic category may be generated out of the same pattern, even once its paradigm has been stated. For example, we could have masculine vs. feminine nouns, or we could have adjectives vs. adverbs, and so forth. Then, a 3-facet key should be used.

As a matter of fact,

- { IS } keys are frequent inside RULES subtrees;
- { IS, INFLECTION_PARADIGM } keys are the most likely exception;
- { IS, MORPHO_CAT_IS } keys are expected to be rarer;
- { IS, INFLECTION_PARADIGM, MORPHO_CAT_IS } keys are unlikely, but not ruled out by the metarepresentation syntax of ONOMATURGE.

Besides, bear in mind that there is no need of distinguishing between AND and OR, in this representation of the frame-schema AND/OR tree; however, optionally, one could state *mutually exclusive* parts of the same RHS (or sub-RHS), by means of the `x:` label. (See in Subsec. 13.1).

`i`<*digit*>`p`<*digit*>`:` is another kind of label. Just as `c:` has been introduced in order to allow having the `i:` facet in a level of facets nested more deeply

than just the children of the facet of the `n:`-preceded slot considered, the option `i`<*digit*> `p`<*digit*> : enables having a facet — whose slot is preceded in the RHS by `j:` (key-completing) — .R in a level nested more deeply than just that of the `i`<*digit*> : facet.

When the slot preceded by the `j:` label is a "nephew" in the tree, rather than a brother, of the slot preceded by `i`<*digit*> :, then let `i`<*digit*> `p`<*digit*> : precede what should have been the `i`<*digit*> :-preceded slot in the proper RHS.

The digit between `i` and `p` may be zero, or a greater natural number, and has just the semantics of the digit of the `i`<*digit*> : label. That is to say, that digit indicates that that particular number of `j:`-preceded slots are present in the same RHS (if undivided), or in the same innermost sub-RHS.

As for the digit between `p` and the colon, it indicates that that particular number of `p:`-preceded slots are present in the same RHS or innermost sub-RHS.

Now, let us consider the `p:` label. When a key-completing facet is at the same tree-level of the key necessary component — this being the facet whose slot is preceded by `i`<*digit*> : or `i`<*digit*> `p`<*digit*> : — then its slot is preceded by `j:` in the proper RHS or sub-RHS.

Instead, when a key completing facet is at the level nested immediately more deeply (i.e., is a "nephew" of the key necessary component), then the father of that key-completing nephew is a brother of the key necessary component, and is listed in the same RHS or sub-RHS, and is preceded by the `p:` label.

As for the "nephew" itself, the label `q:` should precede that slot, in the RHS or sub-RHS where it is listed.

11.11 Facets Containing Local Navigation-Guides

The syntax of CuProS is as flexible as to allow a subtree of a frame instance to contain a metarepresentation specifying how to navigate locally. That is to say, there is the option of the metarepresentation and the object-level representation (the latter being the frame instance) commingling together. This is not a situation we would normally envisage or recommend (while recognising that in some future application, a local metarepresentation may be convenient). In fact, in the implementation of Navigation we did not include also the parsing of such a local metarepresentation by switching to it as found inside the frame instance being visited; it would be straightforward however to allow the parser such switching.

`g:` should precede slots of *navigation-guide facets* mentioned in the frame instance itself, and describing the access to instance-specific schemata of subframes.

While frame-representation in frames is the *object-level* of representation, the production-system stating legal schemata of frames is the *meta-level* representation, or *metarepresentation.*

The `g:` option enables the object-level representation to include subframes described by *local metarepresentations:* such a description is stored in — or pointed to from — facets whose slot is preceded by the label `g:` in the *global*

metarepresentation (or, more generally, in the metarepresentation that is immediately more general, if several levels of locality of metarepresentations are present).

12 Sample Rules from the Metarepresentation of the Lexical Frames of ONOMATURGE

12.1 The Top Few Levels in Lexical Frames

Thus far, we have explained various labels that may occur inside a CUPROS metarepresentation. Further options as expressed by means of labels in a CUPROS metarepresentation will be explained by referring to the sample metarepresentation code shown in the following. These are productions from the metarepresentation of the lexical entries of the ONOMATURGE expert system.

```
(atom                ( INFLECTION
                       MORPHO_CAT
                       x:    ACCEPTATIONS
                       x: LEXEMES
        . . . . . )  )
```

The root of the lexical frame is represented by *atom*. A lexical entry or a rule may have more than one meaning (i.e. acceptation). This is why there is a subtree rooted in `ACCEPTATIONS`. Sometimes acceptations are clustered in lexemes of the same entry. If `LEXEMES` is under the root, then `ACCEPTATIONS` is not there at the same nesting level. Such being the case, `ACCEPTATIONS` may be found instead as being a child of the slotless nonterminal *single_lexeme* — which is itself a child of `LEXEMES`.

```
(MORPHO_CAT          ( x: terminal_list
      x: ARE
                         y: s: single_lexeme
                            s: single_acceptation ) )
```

Under `MORPHO_CAT`, a possible example of *terminal_list* is

```
(noun masculine plural)
```

A very important part of a lexical frame is the part where different meanings are distinguished and described:

```
(ACCEPTATIONS        ( ARE ) )
```

Acceptations are distinct meanings (especially such word senses that are related to each other) of a lexical entry, or even of a word-formation rule (this is so, because some given derivational pattern may be productive of agent nouns, for example).

```
(LEXEMES              ( ARE ) )
```

When the etymology is different, or when meanings are unrelated, then dictionaries would rather list the lexical entry under distinct paragraphs, and these are called *lexemes.* Each lexeme may include even several acceptations.

12.2 The Metarepresentation Rules for Facets of Individual Lexemes or Acceptations

Consider the structure of the subtree of an individual lexeme:

```
(single_lexeme   ( i: LEXEME_KEYWORD
                   INFLECTION
                   MORPHO_CAT
                   ACCEPTATIONS ) )
```

Inside the facet of `LEXEME_KEYWORD`, in the lexical frames of the ONOMATURGE expert system one expects to find an English-like, unambiguous identification string. By "English-like", we mean that the string looks like an English word or compound, but this string is conventional, so one should not mistake this for an actual description in English of what the given lexeme means.

Concerning the facets `INFLEXION` and `MORPHO_CAT`, facets with those slots may also be found directly under the root of the lexical entry, that is to say, directly in the metarepresentation production rule whose left-hand side is the string `atom` (the labels `s:` and `z:` are relevant for this).

```
(single_acceptation  ( c: MEANING
                       CONNOTATIONS
                       RELATED_TO
                       AURO
                       . . . . .  ) )
```

In the latter CUPROS production, the slot `MEANING` is the root of a subtree that may include any out of different kinds of artificial intelligence representations for expressing the meaning of the same semantic concept. Actually the ONOMATURGE expert system did not require accessing the `MEANING` facet, because the manipulation of the English-like descriptors of acceptations or lexemes was enough for getting reasonable results.

The `AURO` facet was used by the control component of the sc Onomaturge expert system in order to rank the candidate neologisms that it generates in output. `AURO` is a fuzzy mark of a subjective estimate of frequency vs. rarity. For both the stem employed in a coinage trial, and the rule if owning a frame, the main scores retrieved and employed are `AURO` and `AURC` as well as `NORMATIVITY` (if found). `AURO` stands for *Actual Use Relevance Ordinal,* whereas `AURC` stands for *Actual Use Relevance Cardinal.*

`AURO` describes context-independent subjectively rated frequency of the acceptation with respect to the other acceptations of the same lexical entry.

AURO = 1 means that the considered acceptation has the subjectively rated highest frequency among those associated with the acceptations of the considered frame. Note however that scores are neither necessarily consecutive numbers, nor unique. Two acceptations may happen to have, both, the same AURO score; if no smaller AURO value is found in any AURO facet of any other acceptation in the same frame, then we are facing the worst case of ambiguity (provided that context is ignored, and this of course quite a big methodological assumption, and possibly a flaw of the design; the proof however is in the pudding, and the results of ONOMATURGE looked reasonable).

AURC describes context-independent subjectively rated frequency of the acceptation with respect to the whole lexicon. That is, AURC states how rare the term is, as used in a certain acceptation, in the contemporary use of the considered natural language. AURC values around 250 are associated with frequent acceptations of frequent terms. AURC values around 800 indicate that the term (as in the acceptation considered) is considerably rare. We must stress that numeric information that we provide on diffusion, is *not* statistically based on the frequency of occurrences in document corpora. Our frequency degrees are subjectively estimated diffusion degrees, where scaling is also conventional and "naive", with no claim of precision in terms of frequency, or even of emulating actual values of frequency: Nissan, while developing ONOMATURGE, took the liberty to exaggerate the diffusion or the rarity of terms, acceptations, and rules. By contrast, rigorous investigation into *term frequency* has been carried out in several places, concerning various languages or corpora.

12.3 Facets for Semantic, Morphological, and Historical Information

Now, let us turn to the metarepresentation of the RELATED_TO facet. Inside it, the RULES facet lists such word-formation rules that have a meaning coincident with the considered meaning of the lexical-entry atom. In the ONOMATURGE expert system, derivational patterns were implemented either in LISP, on in the Unix Shell language. For example, the Hebrew derivational pattern Pa^~aL was implemented as template-rule called Pa^~aL and meaning 'any professional'. Therefore it could be expected to be listed in the RULES facet inside the lexical frame of a word meaning'a professional'.

Under the facet PECULIAR_FACETS it was envisaged that one could include a possible local guide, that is to say, a "private" metarepresentation to be stored inside the lexical frame itself, instead of separately.

```
(RELATED_TO      ( RULES
                   g: PECULIAR_FACETS
                   CONTAINED_IN_THE_IDIOMS
                   SYNONYMS
                   NEAR_SYNONYMS
                   CONTRARIES
                   MOP
```

```
                . . . . . ) )
```

Actually `MOP` facets were not included inside the lexical frames in the ONOMATURGE expert system, but the very possibility of inclusion was envisaged. Roger Schank's and Michael Dyer's *Memory Organization Packets,* or *MOPs* [16] enable to represent an abstraction of goal-driven scripts, corresponding to an abstract contractual relation (such as a shop, without the need to stick to a particular kind of shop), and with several characters being associated with various roles, each with its own chain of goals and plans. This enables interpretation by default, when something is not stated explicitly in the story analysed by Dyer's BORIS automated story-understanding program. About story understanding, see, e.g., [74, 75] in Vol. 1 of the present book.

```
(ARE          ( f: ( SYNONYMS  NEAR_SYNONYMS )
                   n: componential_difference_doubleton
                 f: ( ACCEPTATIONS )
                   n: single_acceptation
                 f: ( LEXEMES )
                   n: single_lexeme
                 f: ( MORPHO_CAT )
                   n: HISTORY
                   n: MORPHO_WITH_WEIGHT
)             )

(componential_difference_doubleton
                           ( i: IS
                       DIFFERENCES ) )
```

The latter production rule is suitable for a semantic componential analysis. For example, under the `IS` facet, one may find `(IS sofa)` in the frame of the term that denotes the general lexical concept 'chair' (for sitting). Eventually, componential analysis was not used by the control component of the ONOMATURGE expert system, but some of the preparatory work [63, 64] considered the use of componential analysis indeed.

```
(MORPHO_WITH_WEIGHT    ( i: MORPHO_CAT_IS
           RELEVANCE_IS ) )
```

Sometimes, the morphological gender of a noun, or — in general — the morphological category of a word or of a template, differs in different historical strata of the given language.[24] Besides, when a noun has been, e.g., masculine

[24] This was historically the case of the Hebrew noun *śadé,* i.e., 'field'. It is masculine in Biblical and Modern Hebrew, but feminine in Tannaic Hebrew. It was also the case of *lašón,* the name for 'tongue' in Hebrew. In Biblical Hebrew it is feminine, but in Tannaic (or Tannaitic) Hebrew it is sometimes masculine, and sometimes feminine.

and feminine in the same stratum, one ought to state what was the relative frequency in use, e.g., that feminine was far more frequent than masculine. This is shown in the following metarepresentation code.

Take notice of how we position the CONTEXT facet: this way, the same MORPHO_CAT_IS may refer to several contexts, with different relevance.

```
      (HISTORY  (  x:   i1: MORPHO_CAT_IS
                        RELEVANCE_IS
                        j:  STRATUM
   2x:  i:  MORPHO_CAT_IS
n:  CONTEXT
   3x:  i:  MORPHO_CAT_IS
d:  DEFINE_RELATION
rel: MORPHO_CAT_IS
rel: CONTEXT
   x:   i:  MORPHO_CAT_IS
    RELEVANCE_IS
    ORDINAL
   x:   i:  MORPHO_CAT_IS
    RELEVANCE_IS
    CAME_AFTER
   y:   s: single_lexeme
        s: single_acceptation
      )          )
```

The CAME_AFTER facet states some historical sequence. The argument of CAME_AFTER is the value of another MORPHO_CAT_IS, or some pointer to any atom or frame-vertex. As for *single_lexeme* and *single_acceptation* — consider that sometimes, the morphological category (e.g. masculine noun or feminine noun) depends upon the particular meaning of the word or of the template-rule. Then, a MORPHO_CAT facet (the root of a subtree) should be looked at, under the particular acceptation or lexeme.

```
      (CAME_AFTER          ( n: MORPHO_CAT_IS
                             n: POINTER       ) )
```

As we are not sure that HISTORY will remain unique, in the metadata schema, we give its proper ancestry. ARE is ambiguous. Therefore, its father is given, too:

```
      (CONTEXT        ( a: ( (HISTORY 2x ARE MORPHO_CAT) )
                        n:  stratum_information
```

By contrast, *šémeš* — the name for 'sun' in Biblical Hebrew — is feminine but on occasion masculine (but it is always feminine in Israeli Hebrew), whereas in Tannaic Hebrew the term itself was replaced with the feminine noun *ḥammá* (literally, 'hot one'), just as *yaréaḥ,* the masculine noun by which Biblical Hebrew refers to the moon (and that is also the term in use in Israeli Hebrew), was replaced in Tannaic Hebrew with the feminine noun *levaná* (literally, 'white one').

```
)                )

(stratum_information ( i: STRATUM
                  RELEVANCE_IS ) )

(DEFINE_RELATION  ( a: ( (HISTORY 3x ARE MORPHO_CAT) )
                    n: relation
)                 )
```

12.4 Considerations About Facets in the Lexical Frames

In the introduction to Menaḥem Moreshet's *A Lexicon of the New Verbs in Tannaic Hebrew* [58, Sec. III.4, p. 75 ff], one finds this example of disappearance or shrinking use of Biblical Hebrew verbs from given lexical roots in the Tannaic stratum of Hebrew, and the emergence of a new verb that took over for denoting the same sense 'to be in charge (of)':

$$\underset{\downarrow}{\sqrt{\text{šrt}}}\ ,\quad \underset{\downarrow}{\sqrt{\text{khn}}}\ \text{—}\ \underset{\uparrow}{\sqrt{\text{šmš}}}$$

(shrank) disappeared appeared, took over

In the frame of the root $\sqrt{\text{šmš}}$, or of the verb /šimmeš/, derived from it, and that means 'to be in charge (of)', it makes sense to include this information about the history of those verbs:

```
(EVOLUTION
  ( (FROM_STRATUM   ( Biblical_Hebrew )  )
    (TO_STRATUM     ( Tannaic_Hebrew )  )
    (PHENOMENA
               (  (ROOT  ( $rt ) )
                  (DID   ( shrink ) )   )
               (  (ROOT  ( khn ) )
                  (DID   ( disappear ) )   )
               (  (ROOT  ( $m$ ) )
                  (DID   ( appear  take_over ) ) )
) ) )
```

This is a kind of phenomenon to which we are going to devote some attention in Appendix B. In the present subsection, instead, we are going to make a few more general considerations. Let us consider a particular kind of facet from the lexical frames of ONOMATURGE: `HISTORY` under `MORPHO_CAT`. Its purpose is to store knowledge about the evolution of how a certain lexical entry used to belong to *morphological categories*, throughout history as documented in different *historical strata* of the language. [25]

[25] In the present volume, the articles [77, 78] are concerned with historical linguistics, and in part the approach to representation is akin to lexical nested relations as discussed here.

Such knowledge is not resorted to by the control of sc Onomaturge, but in general, knowledge-bases in lexicography may involve also such information; for example:

- for such lexical entries that used to be both *masculine* and *feminine* in the same stratum or in different strata;
- or for lexical entries that used to be *singularia tantum* (i.e., only in the singular) — or, instead, *pluralia tantum:* only in the plural — during a certain historical period, while admitting both singular and plural forms in some other historical periods;
- Sets of semantically related lexical entries happen to share particular morphological features, in given languages: even only the gender.[26]
- More in general, morphological evolution for large classes of terms happen to concern change of morphological gender, especially the terms evolve from a language (or stratum) admitting the neuter, to a language with only masculine and feminine.

Let us elaborate about the latter. A quantitatively relevant, heterogeneous class of Latin neuter nouns are masculine or feminine in Romance languages that have no neuter gender. This phenomenon is related to the partition of nouns belonging to the five Latin nominal declensions, among new paradigms in Romance languages or dialects; cf., e.g., §§350–355 and 383–385 in [87]. A similar phenomenon, perhaps simpler as it involves only the morphological gender, occurred with Yiddish terms in the neuter, that became either masculine or feminine in the Lithuanian dialect of Yiddish, that, because of the influence of the Lithuanian language, had no neuter gender.

Morphological phenomena sometimes involve the gender of *phytonyms* (that is, names of plants), in a given language. This is the case of Italian terms indicating fruit-bearing trees: they often share the treatment of the gender, in opposition to the gender of names of fruits. In Romance languages, names of trees (and fruits) correspond nouns belonging to declensions (and genders) as

[26] Of course, gender alone is far less significant, semantically, than a formation-pattern with a small set of acceptations. Nevertheless, in given contexts, gender still gives useful indications. For example, Italian *pera,* in the feminine, denotes 'pear' (the fruit), whereas *pero,* masculine, denotes 'pear-tree' (and etymologically corresponds a neuter Latin noun). In most cases of fruits that belong, as concepts, to traditional culture (thus, excluding some exotic fruits), gender distinguishes fruits from trees, in Italian.

Gender is somewhat relevant also for *zoonyms,* that is, names of animals. According to the historical period and to the dialect (as it happened in Italian) certain zoonyms had their masculine form used as being the *unmarked* form (that is: the general term, denoting both males and females), while, instead, the feminine form (derived morphologically from the entry in the masculine form) was capable of indicating both females and indistinctly males and females. The distinction between *marked* vs. *unmarked* terms in semantics and in morphology is standard knowledge in linguistics; see, e.g., [47, Vol. 1, Ch. 9.7]. The evolution of the gender of Italian*zoonyms* was discussed by Rohlfs [87, §381, cf. §353].

in Latin. The morphology of phytonyms in Italian was discussed by Rohlfs [87, §382]. Let us consider one more example involving phytonyms and gender. In Latin, the suffix *-ago* was resorted to in order to form phytonyms, and was preserved in Italian in this role, having become *-àggine* in some feminine Italian phytonyms [88, §1058]. In some Italian dialects, the suffix developed a masculine variant (see [88, §1058]), perhaps because of the final *-o* of the third Latin nominal declension, that was attracted into the set of masculine nouns of the second declension (ending in *-us* in the nominative, but in *-o* in the ablative and dative); cf. [87, §§352, 353]. Nevertheless, in Sicilian one finds *piràinu* (see [88, §1058]), i.e., 'pear-tree' (applying the suffix, as opposed to Italian *pero* and to Latin), the masculine suffix and gender were adopted, probably by attraction into the class of tree-names, that use to be in the masculine, in Italian and in its dialects.

The morphology of Hebrew (and Aramaic), too, is concerned, as phytonyms are involved. A *phytonym-formation pattern* in Talmudic Aramaic is the rare `/Pa^PLina/`, instantiated in /parpḥina/ (from the root $\sqrt{\text{prḥ}}$, associated with the meanings 'flower' and 'to flower'), and in /ḥarḥbina/ (which has become *ḥarḥăviná* in Israeli Hebrew), from the root $\sqrt{\text{ḥrb}}$, associated with the meaning 'sword' (of the Hebrew noun ḥerb/ *ḥérev*), and also with the meanings 'dry' and 'destroyed' of the Hebrew adjective ḥareb/ *ḥarév*. In Hebrew, the few instances of the derivational pattern `/Pa^PLina/` are in feminine, by interpreting the final *-a* as indicating the gender, whereas in Aramaic instead the final *-a* originally was the suffixed definite article. In Aramaic, the final *-a* originally was the suffixed definite article, but this role, that is still found in Biblical Aramaic, was lost in later strata, that kept the ending in nouns as found in any syntactic position. A likely conjecture is that the article suffix derived from /ha/, for the demonstrative 'this', that on the other hand also became the Hebrew prefix /ha-/, with the role of definite article and interrogative particle. Cf. the evolution of the Latin demonstrative *ille,* i.e., 'that', which evolved into the Italian prefixed definite articles, as opposed to an areal feature from the Balkans, including, e.g., the Rumanian suffix *-ul* for the determinative article: Italian *il lupo,* Rumanian *lupul,* English *the wolf.*

In a footnote in Alinei and Nissan [3], it was suggested that the immigration of agriculturalists speaking a version of Northwest semitic in which the demonstrative became the article suffix (like in Aramaic) may be what gave raise to the Balkan areal feature (found in Bulgarian, macedonian, Rumanian, and Albanian) by which the determinative article is a suffix. This conjecture pinpointed the suggestion made by Alinei in [2, pp. 215–216] that it was an "unknown language" from the Fertile Crescent, at the time of the introduction of agriculture into the Balkans, that introduced into the Balkanic *Sprachbund* the suffixation of the demonstrative article.

12.5 The Effects of Omission from a Noun Phrase

If one term, or a class of terms, evolved from a noun phrase that included the term considered and an accompanying term, by omitting of the latter, then

gender (or number) agreement in the original noun phrase often determines the gender (and number) of the resulting term.

Omissions may imply assumptions, and assumptions often evolve with technology. Even tramway lines, as opposed to bus or trackless-trolley lines, happen to involve conventions on gender: in Milan, in order to reach the university where he was studying, Nissan used to take *la 60* (a bus route), and then *il 23* (a tram route), instead of taking *la 90* (the route of a trackless trolley). While numbers (or letters) are the name of lines of the urban public transports, the name of tramway lines is in the masculine, by implying *tram,* which is masculine and indicates the vehicle (i.e., the tramcar) or the route, as opposed to the rather affected *tranvia*, that is feminine and indicates the route, and as opposed to *linea autofilotranviaria* that is in the feminine, and indicate any bus (*autobus*), trackless trolley (from *filobus*), or tramway route. By contrast, bus routes are in the feminine, by implying *linea (automobilistica)*, as opposed to *autobus*, which is in the masculine, and indicates the vehicle. A trackless trolley (a trolleybus) is termed a *filobus*, which is morphologically masculine, but its routes are referred to in the feminine, by implying *linea*.

12.6 Socio-Cultural Change, and Morphological Gender

Socio-cultural change is sometimes at the root of morphological gender change, in the way professionals are usually termed. As an example of culture-bound morphological change of gender, consider the names for such professions that, because of varying socio-cultural conditions, used to be typically carried out by men, while at present they are typically carried out by women, so the unmarked form used to follow the prevailing situation (for example: Italian *segretario* and Hebrew /mazkir/ denote 'secretary' and, as they are in the masculine form, at present use to indicate especially a leader in the parties or trade-unions, as well as political advisers, while today, the unmarked term for a secretary at commercial firms, or in low ranks of the administration, is in the feminine: Italian *segretaria*, and Hebrew /mazkira/, that properly, indicate a 'lady secretary'. (In Israeli Hebrew, you record a phone message on the "automated /mazkira/", thus implying that had there been a human being to answer your call, a lady secretary would have taken it). In English, the referent of *typist* is usually assumed by default to be a woman, but in the early 20th century, the entry in the feminine, lady typists were still a novelty. The lexical concept 'lady typist' therefore underwent a transition from being the marked subordinate concept of 'typist' (because a typist was assumed to be a man), to being the unmarked subordinate concept (when one would rather expect a woman to fill the job). In some historical cultures, therefore,'secretary' was assumed to be male, and this was reflected in the default morphological gender. On the other hand, the *secretary* of a political organisation is still masculine by default (also morphologically if the given language has a morphological gender, or at any rate in the conceptual map), even in such cultures where *secretary* — a subaltern office worker — is feminine by default.

12.7 More Concerning Morphological Gender

As one can see, there are good reasons — at least, for certain purposes — for dealing with morphological features, e.g., with morphological gender, from the historical viewpoint, when one is representing lexicographical information. In the next section, we are going to discuss the syntax of the metarepresentation portion of the example. Incidentally, not all of the semantic phenomena we listed are suited by the metarepresentation productions in the metarepresentation excerpt whose syntax we are going to discuss, but our previous discussion provides a motivation for defining facet-subschemata on the history of morphological gender for given lexical entries.

Concerning morphological gender in various languages including Hebrew, Alan D. Corré (in a posting dated 4 May 1990, which appeared in the e-list *Humanist Discussion Group,* Vol. 4, No. 5, circulated on 7 May 1990) made the following remarks, which we reproduce by kind permission given by Corré to Nissan on 8 May 1990:

> Clearly there is some connection between sex and gender, but in most languages it is not clear cut. English pays little attention to sex/gender. It differentiates chiefly in the third person singular pronouns. Occasionally inanimate things are called "she" but this is often rather deliberate. (Consider the news item that went something like this: "Her Grace christened the ship by breaking a bottle of champagne over her bows, after which she gently slid into the water.") Tamil, a Dravidian language quite unrelated to English behaves in much the same way. It has three third person pronouns like English, but there is no adjectival concord by gender. The third person verb does have separate endings though. On the other hand, German, a language closely related to English, has complex gender distinctions that often relate to the shape of the word rather than the sex of the object referred to. Thus the word for "girl" is neuter, and referred to as "it" because the word has a diminutive ending which selects the neuter gender. The Romance languages lack a neuter gender, or largely so, and squeeze all objects into the masculine/feminine dichotomy. This is often determined by the shape of the word. Thus the word for "sentry" in French is feminine, and referred to as "she" although women rarely fulfill this role. In Latin some words of typically feminine shape can however be masculine, *poeta,* for example. One looks in vain for logic in natural languages which constantly change, ironing out some irregularities while creating others. We may ask why the distinction exists at all. Professor Rabin of the Hebrew University told me of an individual of his acquaintance who had unilaterally decided that gender distinctions in Hebrew were unnecessary in the modern world, and refused to use any feminines, referring even to his wife as "he". (I mean the English "he"; the word "he" in Hebrew happens to mean "she"). I imagine that most people would find this quite difficult. It is worth pointing out that Semitic languages are particular to distinguish gender in the "you forms, even where some other distinctions are

obliterated in modern dialects. Accordingly one finds that the recipes on the side of food packages in Hebrew ("take one tablespoon... add water... stir") are invariably in the feminine, while the instructions for operating a hacksaw will be in the masculine. It's easy to see how this fosters sex roles, and probably this is part of the key to the whole issue. Natural languages have many subtle markers which put varying degrees of space between interlocutors. In a recent showing of *People's Court* Judge Wapner chastised a defendant who addressed him as "man". The individual replied: "Sorry, judge." I also once had occasion to calm a colleague who was incensed at a student who had used the expression "... and all that shit..." in an exam paper. I pointed out that the student was probably unaware that such a locution may be OK in the local bar, but is not to be used in written English, and he simply should be advised accordingly. These expressions give cues as to the relationship between speakers, and sometimes misfire. French has a familiar *tu* and a formal *vous* and even verbs to indicate the usage (*tutoyer* — 'to call someone *tu*'). Whether one uses one or the other can sometimes be a matter of difficulty. It's interesting to note that in local Tunisian French, *tu* is routinely used, presumably because the colonists didn't see fit to address the locals ever by the polite form, which is itself a comment on social attitudes. Gender differences are probably tied up with these subtle ways that we differentiate you/me boy/girl lower/higher and so on. Such things can be exasperating or fascinating, and that will probably determine whether you enjoy studying foreign languages, or avoid them like the plague.

13 More Syntax for CuProS Metarepresentations

13.1 Mutual Exclusion of Sub-RHSs

Let us explain the supplementary syntax we introduced in the sample of metaprepresentation given in the previous section. We are going to do so by referring to that example itself. In the production of `HISTORY` in the metarepresentation, we find the `x:` label, which subdivides the RHS into *mutually exclusive sub-RHS bodies*. One more sub-RHS may be present at the same level, as preceded by the label `y:` This is in order to indicate a portion that can be present with facet-structure instantiations according to any of the sub-RHSs that in the metarepresentation, are preceded by `x:` inside the same RHS of the same production.

In the simplest case, the `x:` label can be exploited to impose an *exclusive or* between attributes as found in instances. More generally, `x:` is an RHS-subdividing label, and is used to state *mutually exclusive* sets of attributes. `x:` has a lower priority with respect to `f:` and `a:` Therefore, `x:` can be nested inside sub-RHS subdivided according to ancestry, while the opposite cannot occur. For example, the following production lists `x:`-sub-RHSs inside an `f:`-sub-RHS:

```
(ARE    (f:  ( <father> )
```

```
                x:  ....
                    ....
                x:  ....
                    ....
            a:  ( <ancestry_list> )
                x:  ....
                    ....
                x:  ....
                    ....
    )       )
```

The use of the `y:` label is meant to spare code. A `y:`-sub-RHS may appear after `x:`-sub-RHSs, and list facet-identifiers that *can* be (and are meant to be) different from those of the `x:`-sub-RHSs of the same level, even though the content of the `y:`-sub-RHS body is not necessarily disjoint with respect to the set of facet-identifiers as appearing in the preceding `x:`-sub-RHS.

Bear in mind that not all of the slots (or, generally, facet-identifiers) listed in a RHS or sub-RHS, should necessarily appear in given frame instances.

The rarely used label `yx:` precedes mutually exclusive `y:`-sub-RHSs.

13.2 Pointers to an Exceptional Position in the Hierarchy

In the production whose LHS is `HISTORY` and that was introduced in Subsec. 13.2, the `y:`-sub-RHS includes two slotless-facet identifiers. These are `single_lexeme` and `single_acceptation` They are each preceded by the label `s:` That means that in certain instances, `HISTORY` should not be put under the global `MORPHO_CAT` facet of the lexical entry as a whole, but instead in the facet of a single lexeme or of a single acceptation. This occurs when the morphological category depends on the specific meaning in which the lexical entry is being used.

More generally, `s:` is occasionally useful because sometimes, the localization of a nonterminal slot (together with the subtree stemming out of it) is unsuitable, in some particular instance. It may happen that elsewhere in the hierarchy, a certain position be suitable instead for that nonterminal. Then, `s:` means that the following slot (an `s:`-preceded) slot is a "suitable father, that is to say, the slot that should be the father of the slot being the LHS of the considered production. The *pointer* is *implicit* in `s:` (unlike `z:`).

`z:` also indicates an alternative location in the hierarchy. However, the pointer — with the `z:` option — should be an explicit facet. That liberalises the way of pointing, and moreover allows to point to a slot being polysemous (cf. `f:` and `a:`), and thus ambiguous.

Suitable formats for `z:`-preceded slots are as follow:

```
    (LOOK_UNDER    <suitable_father> )
```

is suitable when the slot `<suitable_father>` can be identified unambiguously. Else, a `PATH` or `POINTER` facet would be a suitable *filler* of the `LOOK_UNDER` facet:

```
(LOOK_UNDER    (PATH    <path> )  )
```

as following `z:` in the production, inside the metarepresentation. This way, `z:` is a one-successor label, but it is followed by a facet, instead of a slot or a slotless-facet identifier.

`<path>` is a full path from the frame-root until `<suitable_father>`. By `<path>` a list is meant, whose first element is the name of the entry owning the frame, and whose last element is <*suitable_father*> One could develop alternative formats to point to <*suitable_father*> For example, by devising a one-successor label — a variant of `s:` — to be followed (like `z:`) by a facet, but with that facet containing an *ancestry-statement* (cf. `a:`), instead of a *full path* as required by `z:`

13.3 Ancestry-Statements That Constrain the Schema of an Attribute to the Context of a Given Sub-RHS

Labels may have the form <*digit*> <*label*> `:` For example, the label <*digit*> `x:` is derived from `x:` Digits are added before literal labels of the RHS-subdividing kind, in order to provide identification for given sub-RHS, in complicated cases when ancestry-statements should specify a particular context as prescribed by a given sub-RHS of a production whose LHS is the preceding slot (or facet-identifier in general).

An example of this is an ancestry-statement inside the RHS of the production of the slot `CONTEXT` as shown in the following:

```
(CAME_AFTER    ( n: MORPHO_CAT_IS
               n: POINTER       ) )

(CONTEXT       ( a: ( (HISTORY 2x ARE MORPHO_CAT) )
                 n:  stratum_information
)               )
```

Concerning `( (HISTORY 2x ARE MORPHO_CAT) )` the reason for stating that path in the tree is that as we are unsure whether `HISTORY` will remain unique, we give its proper ancestry. We could have just given the immediate ancestor, which is `ARE` Nevertheless, `ARE` is ambiguous, as it may occur at different places in the tree. Therefore, its father is given, too.

```
(stratum_information ( i: STRATUM
                     RELEVANCE_IS ) )

(DEFINE_RELATION  ( a: ( (HISTORY 3x ARE MORPHO_CAT) )
                     n: relation
)                     )
```

Let us consider, in particular, <*digit*> `x:` labels. If the ancestry-statement is

```
( .......  <slot>   <digit>x  ....... )
```

then the `x:`-sub-RHS (inside the RHS of that production whose LHS is `<slot>`) whose initial `x:` is replaced by ⟨*digit*⟩`x:` (with the digit as suitable), should be selected. That is to say: if the ancestry statement is, for example,

```
( SLOT1   3x  SLOT2 )
```

then the proper `x:`-sub-RHS in the production of `SLOT1` is the one beginning by `3x:` ⟨*digit*⟩ is not necessarily the ordinal number of the sub-RHS (in the subdivision at the same level). In fact, ⟨*digit*⟩ is used as being an indicator — like line labels in FORTRAN — with no numeric meaning (if not a mnemonic role) attached to it. On the other hand, the very fact that a `x:`-sub-RHS has been labelled by ⟨*digit*⟩`x:` should not be taken to imply that all of the `x:`-sub-RHSs of the same production should be labelled by digits.

Moreover, more than a single `x:`-sub-RHS of the same production can be pointed to at once, by formulating the ancestry-statement as follows:

```
( .....  SLOT_i+1  <digit>x  <digit>x  SLOT_i  ..... )
```

where:

- the two digits are different,
- the order between the ⟨*digit*⟩`x:` labels is indifferent,
- $i+1$ is the index of a "younger generation of slots (that is: of facets nested deeper).

That definition is extended to such sub-RHSs that are not `x:`-sub-RHSs. Then, since it is possible to nest sub-RHSs of different kinds, a hierarchy of nested levels could be stated in the ancestry-statement, inside parentheses:

```
( ...
  SLOT_i+1
             ( <digit><outer_label>
               <inner_label>
             )
  SLOT_i
  ...
)
```

is an example of a legal way of stating an ancestry-statement.

⟨*digit*⟩⟨*label*⟩⟨*digit*⟩ is a generalisation of `label><digit>:` For example, ⟨*digit*⟩`x`⟨*digit*⟩ is to `x`⟨*digit*⟩ (see below), as ⟨*digit*⟩`x` is to `x:`

`x`⟨*digit*⟩ is a generalisation of `x:` It enables us to state an *inclusive or* of different *exclusive-or fans* inside the same RHS (or inside the same sub-RHS, at the same level). For example:

```
x1:   SLOT1
      SLOT2
```

```
x1:   SLOT2
      SLOT3
      SLOT4
      SLOT5
x2:   SLOT6
      SLOT7
x2:   SLOT8
x2:   SLOT9
```

means that all of the sets

```
{ SLOT1 , SLOT2 , SLOT6 , SLOT7 }  ,
{ SLOT1 , SLOT6 }  ,
{ SLOT2 , SLOT3 , SLOT5 , SLOT6 , SLOT7 }  ,
{ SLOT2 , SLOT4 , SLOT7 }  ,
{ SLOT3 , SLOT5 , SLOT8 }  ,
{ SLOT4 , SLOT5 , SLOT9 }  ,
```

and so forth, are slots of facets that can be legally co-present as nested immediately inside that facet whose facet-identifier is the LHS of the considered production in the metarepresentation.

13.4 Cartesian Spaces of Attributes

Whenever a hierarchical organisation of facets would be cumbersome, you can resort to the label `rel:` in order to define an n-to-n shallow relation in the space of several attributes. Let us refer again to the metarepresentation:

```
(atom         ( INFLECTION
                MORPHO_CAT
             x: ACCEPTATIONS
           x: LEXEMES
           . . . . . . . ) )
(MORPHO_CAT  ( x: terminal_list
         x: ARE
               y: s: single_lexeme
                  s: single_acceptation ) )
(ACCEPTATIONS        ( ARE ) )
(LEXEMES             ( ARE ) )
(single_lexeme       ( i: LEXEME_KEYWORD
                       INFLECTION
                       MORPHO_CAT
                       ACCEPTATIONS ) )
(single_acceptation   ( c: MEANING
                        CONNOTATIONS
                        RELATED_TO
                        AURO
```

```
                          . . . . . ) )
(RELATED_TO           ( RULES
                        g: PECULIAR_FACETS
                          ;   A possible local guide:
                          ;   a  metarepresentation.
                        CONTAINED_IN_THE_IDIOMS
                        SYNONYMS
                        NEAR_SYNONYMS
                        CONTRARIES
                        MOP  ;  a graph of goals \& plans.
                      . . . . . ) )
(ARE          ( f: ( SYNONYMS  NEAR_SYNONYMS )
                   n: componential_difference_doubleton
                 f: ( ACCEPTATIONS )
                   n: single_acceptation
                 f: ( LEXEMES )
                   n: single_lexeme
                 f: ( MORPHO_CAT )
                   n: HISTORY
                 n: MORPHO_WITH_WEIGHT
)       )
(componential_difference_doubleton
           ( i: IS
         DIFFERENCES ) )
(MORPHO_WITH_WEIGHT    ( i: MORPHO_CAT_IS
                       RELEVANCE_IS ) )
(HISTORY  ( x: i1: MORPHO_CAT_IS
                   RELEVANCE_IS
                j: STRATUM
        2x: i: MORPHO_CAT_IS
          n: CONTEXT
        3x: i: MORPHO_CAT_IS
          d: DEFINE_RELATION
          rel: MORPHO_CAT_IS
          rel: CONTEXT
        x:  i: MORPHO_CAT_IS
             RELEVANCE_IS
             ORDINAL
        x:  i: MORPHO_CAT_IS
             RELEVANCE_IS
             CAME_AFTER
        y:  s: single_lexeme
            s: single_acceptation
)         )
```

```
(CAME_AFTER          ( n: MORPHO_CAT_IS
                     n: POINTER       ) )
(CONTEXT             ( a: ( (HISTORY 2x ARE MORPHO_CAT) )
                          n:  stratum_information
)                    )
(stratum_information ( i: STRATUM
                   RELEVANCE_IS ) )
(DEFINE_RELATION   ( a: ( (HISTORY 3x ARE MORPHO_CAT) )
                     n: relation
)                  )
```

An extension of the first `x:`-sub-RHS in the RHS of the production of `HISTORY` is the `x:`-sub-RHS that begins by `2x:` That label allows the same `MORPHO_CAT_IS` facet to refer to several contexts, with different relevance degrees (that are quantified, say, numerically, possibly by means of fuzzy values).

A 1-to-*n* correspondence is allowed. Let us see that in an example from the history of derivation in Hebrew. In Hebrew, the word-formation pattern /Poˆ eL/ generates *verb participles* of the Hebrew basic active conjugation in all of the historical strata. The relevance in use of /Poˆ eL/, in all of those strata, is very high. Besides, such participles may happen to be nominalized. The same pattern, /Poˆ eL/, has also generated *departicipial nouns* in every historical stratum, but the relevance in each single stratum differ. On the other hand,[27]

in Tannaic Hebrew — that is, the historical stratum of Hebrew in which the Mishnah (the nucleus of the Talmud) was compiled ca. 200 C.E. — the purpose of generating *departicipial nouns* was satisfied by adopting the pattern /Paˆ oL/. In Aramaic, the etymologically equivalent pattern, /Paˆ oLa/, generated participles of the same verbal conjugation. From /Paˆ oL/, Hebrew departicipial nouns were generated, albeit rarely so; some of them are still in use. See Appendix A. There are two /Paˆ oL/ homonymous polytemplates (i.e., procedural code items) in ONOMATURGE: one whose plural form is /Paˆ oLot/, and another one, instantiated in some obsolete terms of post-Biblical formation, whose plural is /Paˆ oLim/. In relation to this example, the following exemplifies, from inside a frame, different single-rule subtrees. They are legal inside the subtree rooted in `RULES`, inside an acceptation of different lexical lexical entries. A multi-attribute key is stated, because of the existence of the same pattern as with two different forms of the plural.

```
 ( (IS  PaˆoL)
   (INFLECTION_PARADIGM  ( (FORM_IS PaˆoLot)
                          (FORM_CLASSIFICATION plural)
   )                    )
   (RELATED_INFO  <dependent information> )
 )
```

[27] See Bar-Asher [5]. The paragraphs referred to are 16–20 (pp. [13]–[20] in the paper itself; pp. 94–102 as first published in *Leshonenu;* pp. 135–142 as reprinted in 1979/80).

```
( (IS  Pa^oL)
  (INFLECTION_PARADIGM  ( (FORM_IS Pa^oLim)
                          (FORM_CLASSIFICATION plural)
  )               )
  (RELATED_INFO  <dependent information> )
)
```

The triplet

```
atom = /Pa^oL/
MORPHOLOGICAL_CATEGORY = (departicipial noun)
STRATUM = later_Prophets
```

may have a numerical score associated, which expresses the salience in documented use. In fact, even as their text provide evidence for the introduction of /Pa^oL/ into Hebrew just for generating departicipial nouns, some of the later Prophets kept using /Po^eL/ not just for the participle, but also for departicipial nouns. In a later stratum of Hebrew, Tannaic Hebrew, /Pa^oL/ remained in use (along with /Po^eL/) for generating departicipial nouns, so a particular value of salience ("relevance) in the use of that particular historical stratum should be stated.

Table 1. The adjacency matrix of the n-to-n relation of Fig. 12.

	a	**b**	**c**	**d**
1	1	1	0	0
2	0	1	1	0
3	0	0	0	1

In such a situation, it is suitable to represent a 1-to-n correspondence, by resorting to the the `CONTEXT` facet, as indicated in the sub-RHS of `HISTORY` that begins by `2x:` In the metarepresentation, this corresponds to this rule:

```
(CONTEXT   ( a: ( (HISTORY 2x ARE MORPHO_CAT) )
                n:  stratum_information
)          )
```

A further generalisation allows n-to-n correspondence. It is suitable when a hierachical organisation is not the most natural one.[28] For example, let us

[28] Six [96] formalised a framework for data structures, in terms of a general class of objects: "arbitrary directed rooted graphs in which the information associated with each node is either empty, elementary or an object in the general class. Since many data structures cannot be modelled by tree structures in a natural way, such a generalisation to graphs is desirable. Furthermore, the concept of hierarchy in which a node may serve for an arbitrary complex substructure supports a structured view of complex data objects."

consider the relation shown in Fig. 12. Then, its adjacency matrix will be an array of zeros and ones, as in Table 1.

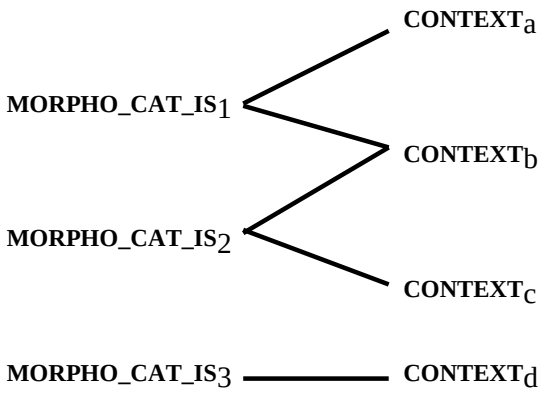


Fig. 12. An n-to-n relation defined in the space of two attributes.

`MORPHO_CAT_IS` facets state the morphological category. According to the schema allowed by the production of `CONTEXT` in the metarepresentation, `MORPHO_CAT_IS` facets are *repeatable*, as nested under `CAME_AFTER` — as per the following metarepresentation rule expressed as a CuProS production:

```
(CAME_AFTER    ( n: MORPHO_CAT_IS
                 n: POINTER       ) )
```

Then, at the same hierarchical level of the repeated `MORPHO_CAT_IS` facets, one could include a `DEFINE_RELATION` facet, and also repeated `CONTEXT` .R facets. `DEFINE_RELATION` itself is not a keyword of the CuProS language: this is the reason for having this slot preceded by the `d:` label, in the RHS or sub-RHS where it is stated.

See in the sub-RHS body preceded by `3x:` in the production having `HISTORY` as LHS, namely:

```
(HISTORY  ( x: i1: MORPHO_CAT_IS
                   RELEVANCE_IS
               j: STRATUM
        2x: i: MORPHO_CAT_IS
          n: CONTEXT
        3x: i: MORPHO_CAT_IS
          d: DEFINE_RELATION
          rel: MORPHO_CAT_IS
          rel: CONTEXT
        x:  i: MORPHO_CAT_IS
```

```
             RELEVANCE_IS
             ORDINAL
        x:   i: MORPHO_CAT_IS
             RELEVANCE_IS
             CAME_AFTER
        y:   s: single_lexeme
             s: single_acceptation
)         )
```

Let the label `d:` define relations represented by a binary matrix. The attributes `MORPHO_CAT_IS` and `CONTEXT`, involved in the relation, are stated to be brothers of `DEFINE_RELATION` in the same sub-RHS. Now, consider the production having `DEFINE_RELATION` as LHS:

```
  (DEFINE_RELATION  ( a: ( (HISTORY 3x ARE MORPHO_CAT) )
                      n: relation
  )                 )
```

In the `a:`-sub-RHS considered of the production whose LHS is the attribute `DEFINE_RELATION`, after the ancestry statement, only `relation` is stated to be repeatable (because of the `n:` label). This should help to avoid redundancy, as the attributes involved in the relation are stated as "uncles in the facet-hierarchy, each preceded by the `rel:` label.

Let us see a portion of a *frame-instance* (not of the the metarepresentation, this time). The following is a possible portion of a frame-instance; an n-to-n relation is represented:

```
(DEFINE_RELATION  ( ( MORPHO_CAT_IS CONTEXT )
                      ( 1 a )  ( 1 b )
                               ( 2 b )  ( 2 c )
                                                ( 3 d )
                  )
                      ( < another_relation> ..... )
                       . . . . . .
)
(MORPHO_CAT_IS  (ID 1)   <value or value_list
                            or frame_subtree> )
(MORPHO_CAT_IS  (ID 2)   <value or value_list
                            or frame_subtree> )
                          . . . . . .
(CONTEXT        (ID a)   <value or value_list
                            or frame_subtree> )
(CONTEXT        (ID b)   <value or value_list
                            or frame_subtree> )
                           . . . . . .
```

The sets {1,2,3} and {a,b,c,d} are just sets of labels of facets nested deeper. Then the label/value, or label/value_list, or even label/subtree association is

made in those facets that in the metarepresentation are preceded by the label `rel:` It would be possible to generalise that n-to-n correspondence in several ways. First of all, we have imposed a constraint, in the frame-schema, on the identity of the attributes involved in the relation. An extension could be introduced, allowing to define their identity instance-wise, and not rigidly in the schema. Moreover, by no means should a relation only involve two attributes: once the arity is liberalised, more coordinates could be stated for each case. This is allowed with the `d:` option itself. Moreover, such tables can be envisaged, that are not arrays of zeroes and ones. Some convention can be devised, for expressing the case value in the table-representation inside the frame hierarchy.

The `l:` label allows relations to be defined as being represented by labelled graphs, and then by matrices with whatever values. The syntax for the coordinate-list representing table-cases could be extended as follows:

```
( <coordinate>   ...   <coordinate>
  VAL      <value or value_list> )
```

Whenever a value-list is stated after VAL, a relation corresponding to a multi-labeled graph is represented.

14 Conclusions

In this article, we have presented the Raffaello project in data storage and retrieval, and the related CuProS syntax for metarepresentation of the schemata of deeply nested relations. The main application, in the 1980s and early 1990s, was in the domain of lexicography and word-formation. There were other applications as follows. The importance of the project has been meanwhile vindicated by the popularity that similar notions have gained with the widespread acceptance of XML.

Whereas the full syntax of CuProS was originally presented in an appendix to Ephraim Nissans doctoral thesis (in computational linguistics, its main supervisor was Yaacov Choueka, the jubilaris of the present volume), this is the very first time that the syntax of CuProS is published in full. It arguably still has features to recommend it, even in an era that has grown accustomed to XML. Even at the risk of this article partly resembling a manual, we feel it is important that we have gone here into the details, and that moreover, in the sections and appendices we have dealt to some extent with the application to lexicography and derivational morphology. This makes that kind of application more readily available for, hopefully, further future adoption of CuProS.

For retrieval from deeply nested, flexibly structured frame-instances as described by means of a metarepresentation coded in CuProS, the most developed tool, the one that fully met the specifications, was Navigation, the version of Raffaello that was implemented by Jihad El-Sana, whose won scholarly career next unfolded in the disciplinary context of image processing. By contrast, Nissans Onomaturge expert system (for Hebrew word-formation and the evaluation of psychosemantic transparency of lexical coinages generated by

ONOMATURGE itself) retrieved data from the frame-instances of lexical entries or derivational morphemes by resorting to SIMPLE NAVIGATION, a tool implemented by Nissan and tailored to the specific frame-structure adopted in ONOMATURGE.

Even though Nissan has also defined CUPROS, and had coded in CUPROS the structure of the frames of ONOMATURGE, it wasnt until El-Sanas implementation of NAVIGATION that CUPROS became a language actually being parsed by computer tools. This is rather like what happened with XML: that language existed, and was studied, well before implementations became available. That CUPROS and XML were developed in mutual independence of each other bears witness to the times having been mature for the appearance of their underlying idea. It was a more radical idea than the one advocated by the nested relations school within database research.

References

1. S., Abiteboul, Fischer, P.C., and Schek, H.-J. (eds.), *Nested Relations and Complex Objects in Databases* LNCS, vol. 361. Springer-Verlag, Berlin, 1989.
2. Alinei, M., *Origini delle lingue d'Europa,* Vol. 2: *Continuità dal Mesolitico all'età del Ferro nelle principali aree etnolinguistiche.* Il Mulino, Bologna, Italy, 2000.
3. Alinei, M., and Nissan, E., L'etimologia semitica dell'it. *pizza* e dei suoi corradicali est-europei, turchi, e semitici levantini. *Quaderni di Semantica: An International Journal of Semantics and Iconomastics,* 28(1), 2007, pp. 117–135.
4. Alterman, R., A Dictionary Based on Concept Coherence. *Artificial Intelligence,* 25(2), 1985, pp. 153186.
5. Bar-Asher, M., Rare Forms in the Tannaitic Language (in Hebrew). *Leshonenu,* 41, 1976/77 (5737 of the Hebrew calendar), pp. 83–102. Also in M. Bar-Asher (Ed.), *Collected Papers in Tannaitic Hebrew,* Vol. 2. The Hebrew University, Jerusalem, 1979/80 (=5740), pp. 123–142.
6. Barron, J., In a Futuristic House, Speak Clearly and Carry a Manual. *Daily Telegraph,* London, 28 October 2004, on p. 7 in *The New York Times* selection supplement.
7. Bertino, E., Catania, B., and Wong, L., Finitely Representable Nested Relations. *Information Processing Letters,* 70(4), pp. 165–173 (1999).
8. Bressan, S. (ed.), *Efficiency and Effectiveness of XML Tools and Techniques [EEXTT] and Data Integration over the Web: Revised Papers from the VLDB Workoshop, at the 28th Very Large Data Bases International Conference,* Hong Kong, China, 2002. Springer, Berlin & New York, 2003.
9. Bryan, M., *SGML and HTML Explained.* Addison Wesley Longman, Harlow, Essex, England, 1997, 2nd edn.
10. Cardenas, A.F., *Data Base Management Systems.* Allyn and Bacon, Boston, 1985, 2nd edn.
11. Colin, G.S. 1927. Note sur le système cryptographique du sultan Aḥmad al-Manṣūr. *Hespéris,* 7(2), pp. 221–228.
12. Dalby, D. 1967. A Survey of the Indigenous Scripts of Liberia and Sierra Leone: Vai, Mende, Loma, Kpelle and Bassa. *African Language Studies,* 8, initial map + pp. 1–51. School of Oriental and African Studies, University of London, distrib. Luzac, London, 1967.

13. Dalby, D. 1968. The indigenous scripts of West Africa and Surinam: Their Inspiration and Design. *African Language Studies,* 9, pp. 156–197. School of Oriental and African Studies, University of London, distrib. Luzac, London, 1968.
14. Dor, M., *Ha-ḥay bi-ymei ha-Miqra ha-Mishnah ve-ha-Talmud* (Hebrew: *The Fauna at the Times of the Bible, the Mishnah and the Talmud*). Grafor-Daftal Books, Tel-Aviv, 1997.
15. Doutté, E., *Magie et réligion dans l'Afrique du Nord.* Adolfe Jourdan, Algiers, 1909; J. Maisonneuve & P. Geuthner, Paris, 1984.
16. Dyer, M.G., *In-Depth Understanding: A Computer Model of Integrated Processing of Narrative Comprehension.* The MIT Press, Cambridge, MA, 1983.
17. Epstein, R., A Tutorial on INGRES. Memo ERL-M77-25, Electronics Research Laboratory, University of California, Berkeley, 1977.
18. Even-Odem, J., and Rotem, Y., *New Medical Dictionary.* (In Hebrew.) Rubin Mass Publ., Jerusalem, 1975.
19. Fischler, B.: Mass'ei ha shemot. [Hebrew: The Trajectories of Names: On the Development of the Terminology of Birdnames (1866–1972)]. *Lashon ve Ivrit: Language & Hebrew,* 4, July, pp. 6–35 (1990).
20. Freedman, D.N. and Rittersprach, A., 1967 The Use of Alef as a Vowel Letter in the Genesis Apocryphon. *Revue de Qumran,* 6 (1967), pp. 293–300.
21. Fuhr, N. (ed.), *Advances in XML Information Retrieval and Evaluation: 4th International Workshop of the Initiative for the Evaluation of XML Retrieval, INEX 2005,* Dagstuhl Castle, Germany, November 28–30, 2005. *Revised and Selected Papers.* Lecture Notes in Computer Science, vol. 3977. Berlin: Springer, 2006.
22. Garani, G., *A Temporal Database Model Using Nested Relations.* Ph.D. Dissertation, Computer Science and Information Systems Engineering, Birkbeck College, University of London, 2004.
23. Garani, G., Nest and Unnest Operators in Nested Relations. *Data Science Journal,* 7, pp. 57–64 (2008).
24. García-Page Sanchez, M., Sobre los procesos de deslexicalización en las expresiones fijas. *Español actual,* 52, pp. 59–79 (1989).
25. Genesereth, M.R., and Lenat, D.B., Meta-Description and Modifiability. Technical Report HPP-80-18, Heuristic Programming Project, Computer Science Dept., Stanford University, 1980.
26. Goldfarb, C.F., *The SGML Handbook,* ed. and introd. Y. Rubinsky. Clarendon Press, Oxford, 1990, repr. 2000.
27. Ghosh, S.P., *Data base Organization for Data Management.* Academic Press, New York, 1977.
28. Greiner, R., RLL-1: A Representation Language Language [sic]. Technical Report 80-0, Heuristic Programming Project, Computer Science Dept., Stanford University, 1980. An expanded version of the paper was published in the *Proceedings of the First National AAAI Conference,*
Stanford, 1980.
29. Grimes, J.E. 1984 Ḍenormalization and Cross-Referencing in Theoretical Lexicography. *Proceedings of the COLING'84 Conference,* pp. 38–41 (1984).
30. Grimes, J.E., de la Cruz Ávila, P., Carrillo Vicente, J., Díaz, F., Díaz, R, de la Rosa, A, and Rentería, T., *El Huichol: Apuntes sobre el léxico.* Department of Modern Languages and Linguistics, Cornell University, Ithaca, NY, 1981. (In Spanish, abstracted in English on pp. 283–291).
31. Hatzopoulos, M., and Kollias, J.G., A Dynamic Model for the Optimal Selection of Secondary Indexes. *Information Systems,* 8(3), pp. 159–164 (1983).

32. Holzner, S., *XML complete.* McGraw-Hill, New York, 1998.
33. Jastrow, M., *Dictionary of the Targumim, the Talmud Babli and Yerushalmi, and the Midrashic Literature.* [Definitions are in English]. 2 vols. G.P. Putnam's Sons, New York; Trübner, Leipzig, and Luzac, London (1886–1903). Reprints: Choreb, New York (2 vols. in 1, 1926, later reprinted by Chorev, Jerusalem); Shapiro [i.e., Shapiro & Vallentine], London (1926); Title Publ., New York (1943); New York: Pardes Publ., New York (2 vols., 1950), and with a new title, *Hebrew-Aramaic-English Dictionary...* (2 vols., 1969). Also (with the standard title), Judaica Press, New York, 1971 (2 vols. in 1); Jastrow Publishers, New York, 1903, repr. 1967 (2 vols. in 1); Shalom, Brooklyn, 1967 (2 vols.); Hendrickson Publ., Peabody, Massachusetts, 2005 (in 1 vol. of 1736 pp.). Also accessible on the Web;
Vol. 1: http://www.etana.org/sites/default/files/coretexts/14906.pdf
Vol. 2: http://www.etana.org/sites/default/files/coretexts/14499.pdf
Arranged for alphabetical access online at Tyndale House:
http://www.tyndalearchive.com/tabs/jastrow/
34. Kent, W., The Universal Relation Revisited. *ACM Transactions on Database Systems,* 8(4), pp. 644–648 (1983).
35. Kernighan, B.W., and Ritchie, D.M., *The C Programming Language.* Prentice-Hall, Englewood Cliffs, N.J., 1978.
36. Khoshafian, S., Valduriez, P., and Copeland, G., Parallel Query processing for Complex Objects. MCC Technical Report DB-397-86, Microelectronics and Computer Technology Corporation, Austin, Texas, November 1986.
37. Kimron, E. 1974/5 Middle Aleph as a Mater Lectionis in Hebrew and Aramaic Documents from Qumran, as Compared to Other Hebrew and Aramaic Sources. (In Hebrew.) *Lěšonénu,* 39. The Academy of the Hebrew Language, Jerusalem, 5735 = 1974/5, pp. 133–164. Also in: M. Bar-Asher (ed.), *Collected Papers in Tannaic Hebrew.* Vol. 2. The Hebrew University, Jerusalem, 5740 = 1979/80, pp. 335–348.
38. Kitagawa, H., Kunii, T.L., and Ishii, Y., Design and Implementation of a Form Management System, APAD, Using ADABAS/INQ DBMS. Report 81-18, Information Science Dept., University of Tokyo, 1981.
39. Kutscher, E.Y., Trivia (in Hebrew: Zutot). In: E.Y. Kutscher (ed.), *Archive of the New Dictionary of Rabbinical Literature,* Vol. 1. Bar-Ilan University, Ramat-Gan, Israel, 1972, pp. 95–105; English summary: pp. xxx–xxxii.
40. Larus, J.R. Parlez-Vous Franz? An Informal Introduction to Interfacing Foreign Functions to Franz LISP. Technical Report PAM-124, Center for Pure and Applied Mathematics, University of University of California, Berkeley (early 1980s).
41. Levene, M., *The Nested Universal Relation Database Model.* LNCS, vol. 595, Springer-Verlag, Berlin, 1992.
42. Levene, M., and Loizou, G., The Nested Relation Type Model: An Application of Domain Theory to Databases, *The Computer Journal,* 33(1), pp. 19–30 (1990).
43. Levene, M., and Loizou, G., Semantics of Null Extended Nested Relations. *ACM Transactions on Database Systems,* 18, 414–459 (1993).
44. Levene, M., and Loizou, G., The Nested Universal Relation Data Model. *Journal of Computer and System Sciences,* 49, 683–717 (1994).
45. Levene, D., and Rothenberg, B., *A Metallurgical Gemara: Metals in the Jewish Sources.* Metal in History series, vol. 4. Institute for Archaeo-Metallurgical Studies, Institute of Archaeology, University College London, London; distributed by Thames & Hudson, London (except in the USA and Canada), 2007.

46. Levenston E.A. and R. Sivan, The Megiddo Modern Dictionary. [English-Hebrew dictionary.] Megiddo, Tel-Aviv, 1966 (in one volume), repr. 1982 (in two vols., in other respects identical).
47. Lyons, J., *Semantics* (2 vols.). Cambridge University Press, Cambridge, England, 1977.
48. Lamping, J., and Rao, R., The Hyperbolic Browser: A Focus + Context Technique for Visualizing Large Hierarchies. *Journal of Visual Languages & Computing,* 7(1), 1996, pp. 33–55.
49. Larussa, M., Building a Translator for Nested-Relation into XML Document. Dissertation for a BsC degree in Computing Science (supervised by A. Al-Zobaidie and E. Nissan). University of Greenwich, London, 2001.
50. Light, R., *Presenting XML.* Sams.net, Indianapolis, Indiana, 1997.
51. Mackenzie, C., Falstaff's Monster. *AUMLA: Journal of the Australasian Universities Language and Literature Association* (New Zealand), 83, pp. 83–86 (May 1995).
52. Mark, L. and Roussopoulos, N., Metadata Management. *IEEE Computer* 19(12), pp. 26–36 (1986).
53. Mauchamp, E.: *La sorcellerie au Maroc. Oeuvre posthume.* Dorbon-ainé, Paris, n.d. [1911].
54. McGilton, H. and Morgan, R., *Introducing the UNIX System.* McGraw-Hill, New York, 1983.
55. Minh, T.T.H., *Approaches to XML Schema Matching,* Ph.D. Thesis, University of East Anglia, Norwich, England, 2007.
56. Monod, Th. 1938. Systèmes cryptographiques. In: Th. Monod (ed.), *Contributions à l'étude du Sahara occidental,* Fasc. 1: *Gravures, peintures et inscriptions rupestres.* Comité d'études Historiques et Scientifiques de l'Afrique Occidentale Française, Publications,Série A, no. 7. Librairie Larose, Paris, pp. 114–126.
57. Monteil, V. 1951. La cryptographie chez les maures. *Bulletin de l'I.F.A.N.,* 13(4), pp. 1257–1264.
58. Moreshet, M., *A Lexicon of the New Verbs in Tannaic Hebrew.* (In Hebrew.) Bar-Ilan University Press, Ramat-Gan, Israel, 1980.
59. Nakhimovsky, A., and Myers, T., *XML Programming: Web Applications and Web Services with JSP and ASP.* The Expert's Voice Series. APress, Berkeley, CA, 2002.
60. Nini, Y. (ed., introd.), Gamlieli, N.B. (trans.), *Al-Misawwadeh: Bēth-Dīn (Court) Records of the Ṣan'ānī Jewish Community in the Eighteenth Century.* Vol. 1: *Ms. Heb. 38° 3281/2, The Jewish and University Library, Jerusalem.* Publications of the Diapora Research Institute, Book 145. The Diaspora Research Institute, Tel Aviv University, Tel Aviv, 2001. (In Hebrew, with Judaeo-Arabic. English abstract.)
61. Nissan, E., *Proprietà formali nel progetto logico-concettuale di basi di dati.* (Italian: Formal Properties in the Logical and Conceptual Design of Databases.) Tesi di Laurea in Ingegneria Elettronica, Dipartimento di Elettronica, Politecnico di Milano, 1982. 2 vols., 400+200 p. Awarded the Burroughs Italiana Prize.
62. Nissan, E., The Info-Spatial Derivative: A New Formal Tool for Database Design. *Proceedings of the AICA'83 Conference,* Naples, Vol. 2, pp. 177–182 (1983).
63. Nissan, E., The Twelve Chairs and ONOMATURGE. Part I of: The Representation of Synonyms and of Related Terms, in the Frames of an Expert System for Word-Coinage. *Proceedings of the 5th International Workshop on Expert Systems & Their Applications,* Avignon, June 1985. Vol. 2 of 2, pp. 685–703, 1985.

64. Nissan, E., On Lions, Leopards and ONOMATURGE. Part II of: The Representation of Synonyms and of Related Terms, in the Frames of an Expert System for Word-Coinage. *Proceedings of the 5th International Workshop on Expert Systems & Their Applications,* Avignon, June 1985. Vol. 2 of 2, pp. 705–741, 1985.
65. Nissan, E., The Frame-Definition Language for Customizing the RAFFAELLO Structure-Editor in Host Expert Systems. In: Z. Ras and M. Zemankova (eds.), *Proceedings of the First International Symposium in Methodologies for Intelligent Systems (ISMIS'86),* Knoxville, Tennessee, 1986. ACM SIGART Press, New York, pp. 8–18, 1986.
66. Nissan, E., Data Analysis Using a Geometrical Representation of Predicate Calculus. *Information Sciences,* 41(3), pp. 187–258 (1987).
67. Nissan, E., The WINING AND DINING Project. Part II: An Expert System for Gastronomy and Terminal Food-Processing. *International Journal of Hospitality Management,* 6(4), pp. 207–215 (1987).
68. Nissan, E., ONOMATURGE: An Expert System for Word-Formation and Morpho-Semantic Clarity Evaluation. Part I: The Task in Perspective, and a Model of the Dynamics of the System. In: H. Czap and C. Galinski (eds.), *Terminology and Knowledge Engineering: Proceedings of the First International Conference,* Trier, Germany, 1987. Indeks Verlag, Frankfurt/M, pp. 167–176, 1987.
69. Nissan, E., ONOMATURGE: An Expert System for Word-Formation and Morpho-Semantic Clarity Evaluation. Part II: The Statics of the System. The Representation from the General Viewpoint of Knowledge-Bases for Terminology. In: H. Czap and C. Galinski (eds.), *Terminology and Knowledge Engineering: Proceedings of the First International Conference,* Trier, Germany, 1987. Indeks Verlag, Frankfurt/M, pp. 177–189, 1987.
70. Nissan, E., Nested-Relation Based Frames in RAFFAELLO. Meta-representation Structure & Semantics for Knowledge Engineering. In: Schek, H.-J. and Scholl, M. (eds.), *International Workshop on Theory and Applications of Nested Relations and Complex Objects,* Darmstadt, Germany, 6–8 April 1987. Handouts printed as a report of INRIA, Rocquencourt, France, pp. 95–99, 1987.
71. Nissan, E., Knowledge Acquisition and Metarepresentation: Attribute Autopoiesis. In: Z.W. Ras and M. Zemankowa (eds.), *Proc. 2nd International Symposium Methodologies for Intelligent Systems (ISMIS'87).* North-Holland, Amsterdam & New York, pp. 240–247, 1987.
72. Nissan, E., *ONOMATURGE: An Expert System in Word-Formation* (3 vols.). Ph.D. Dissertation in Computer Science, Ben-Gurion University of the Negev, Beer-Sheva, Israel, 1988.
73. Nissan, E., Deviation Models of Regulation: A Knowledge-Based Approach. *Informatica e Diritto,* year 18, 2nd Series, 1(1/2), pp. 181–212 (1992).
74. Nissan, E., Narratives, Formalism, Computational Tools, and Nonlinearity. In: N. Dershowitz and E. Nissan (eds.), *Language, Culture, Computation: Essays in Honour of Yaacov Choueka,* Vol. 1: *Theory, Techniques, and Applications to E-Science, Law, Narratives, Information Retrieval, and the Cultural Heritage.* Springer, Berlin.
75. Nissan, E., Tale Variants and Analysis in Episodic Formulae: Early Ottoman, Elizabethan, and "Solomonic" Versions In: N. Dershowitz and E. Nissan (eds.), *Language, Culture, Computation: Essays in Honour of Yaacov Choueka,* Vol. 1: *Theory, Techniques, and Applications to E-Science, Law, Narratives, Information Retrieval, and the Cultural Heritage.* Springer, Berlin.

76. Nissan, E., Terminological Database Modelling of Multilingual Lexis, Semantics, and Onomasiology by Using the CuProS Metarepresentation Language: An XML-Compatible XML-Precursor Enabling Flexible Nested-Relation Structures. In: N. Dershowitz and E. Nissan (eds.), *Language, Culture, Computation: Essays in Honour of Yaacov Choueka,* Vol. 2: *Tools for Text and Language, and the Cultural Dimension.* Springer, Berlin, this volume.
77. Nissan, E., Which Acceptation? Ontologies for Historical Linguistics In: N. Dershowitz and E. Nissan (eds.), *Language, Culture, Computation: Essays in Honour of Yaacov Choueka,* Vol. 2: *Tools for Text and Language, and the Cultural Dimension.* Springer, Berlin, this volume.
78. Nissan, E., Etymothesis, Fallacy, and Ontologies: An Illustration from Phytonymy. In: N. Dershowitz and E. Nissan (eds.), *Language, Culture, Computation: Essays in Honour of Yaacov Choueka,* Vol. 2: *Tools for Text and Language, and the Cultural Dimension.* Springer, Berlin, this volume.
79. Nissan, E., Knowledge Engineering for Word-Formation: Generating and Evaluating Candidate Neologisms. In: N. Dershowitz and E. Nissan (eds.), *Language, Culture, Computation: Essays in Honour of Yaacov Choueka,* Vol. 2: *Tools for Text and Language, and the Cultural Dimension.* Springer, Berlin, this volume.
80. Nissan, E., A Study of humorous Explanatory Tales (2 parts, 2011). In a book in preparation.
81. Nissan, E., and HaCohen-Kerner, Y., Information Retrieval in the Service of Generating Narrative Explanation: What We Want from GALLURA. In *Proceedings of KDIR 2011: International conference on Knowledge Discovery and Information Retrieval,* Paris, 2629 October 2011, pp. 487-492.
82. Nissan, E., Asaro, C., Dragoni, A.F., Farook, D.Y., and Shimony, S.E., A Quarter of Century in Artificial Intelligence and Law: Projects, Personal Trajectories, a Subjective Perspective In: N. Dershowitz and E. Nissan (eds.), *Language, Culture, Computation: Essays in Honour of Yaacov Choueka,* Vol. 1: *Theory, Techniques, and Applications to E-Science, Law, Narratives, Information Retrieval, and the Cultural Heritage.* Springer, Berlin.
83. Ornan, U., Hebrew Word Formation. In: M. Bar-Asher et al. (Eds.), *Volume in Honor of Z. Ben-Hayyim* (in Hebrew). Magnes Press, The Hebrew University, Jerusalem, Jewish year 5743 = 1982/3, pp. 13–42.
84. Osborne, W.R., A Linguistic Introduction to the Origins and Characteristics of Early Mishnaic Hebrew as it Relates to Biblical Hebrew. *Old Testament Essays,* new series, 24(1), 2011, pp. 159172.
85. Özsoyoğlu, M.Z., (ed.), *Nested Relations.* Special issue, *The IEEE Data Engineering Bulletin,* 11(3). IEEE, 1988.
86. Özsoyoğlu, Z.M., and Yuan, L.Y., A New Normal Form for Nested Relations. *ACM Transactions on Database Systems,* 12, pp. 111–136 (1987).
87. Rohlfs, G., *Grammatica storica della lingua italiana e dei suoi dialetti.* Vol. 2: *Morfologia.* Piccola Biblioteca Einaudi (PBE), Vol. 149, Einaudi, Torino, Italy, 1968. Revised edition, translated from:
Historische Grammatik der Italianischen Sprache und ihrer Mundarten. Vol. 2: *Formenlehre und Syntax.* A. Francke AG, Bern, Switzerland, 1949.
88. Rohlfs, G., *Grammatica storica della lingua italiana e dei suoi dialetti.* Vol. 3: *Sintassi e formazione delle parole.* Piccola biblioteca Einaudi (PBE), Vol. 150. Einaudi, Torino, Italy, 1969. Revised edition, translated from: *Historische Grammatik der Italianischen Sprache und ihrer Mundarten,* Vol. 3: *Syntax und Wortbildung.* A. Francke AG, Bern, Switzerland, 1954.

89. Rosen, B.K., Tree-Manipulating Systems and Church-Rosser Theorems. *Journal of the ACM,* 20(1): pp. 160–187 (1973).
90. Sáenz-Badillos, A., *A History of the Hebrew Language,* trans. J. Elwode. Cambridge University Press, New York & Cambirdge, U.K., 1993.
91. Sarkar, M., and Brown, M.H., Graphical Fisheye Views. *Communications of the ACM,* 37, 1994, pp. 73–84.
92. Schkolnick, M., The Optimal Selection of Secondary Indices for Files. *Information Systems,* 1, pp. 141–146 (1975).
93. Segal, M.H., *A Grammar of Mishnaic Hebrew.* Clarendon, Oxford, 1978 (Reprint of 1st edn., with corrections). Previously published in 1927, 1958 (repr. lithographically from corr. sheets of the 1st edn.), 1970, 1978, 1980.
94. Schwartz, S., *Were the Jews a Mediterranean Society? Reciprocity and Solidarity in Ancient Judaism.* Princeton University Press, Princeton, NJ.
95. Shaw, M.L.G., Bradshaw, J.M., Gaines, B.R., and Boose, J.H. Boose, Rapid Prototyping Techniques for Expert Systems. *Proceedings of the Fourth IEEE Conference on Artificial Intelligence Applications,* San Diego, California, March 14–18, 1988.
96. Six, H.-W., A Framework for Data Structures. In: G. Goos and J. Hartmanis (Eds.), *Graphtheoretic Concepts in Computer Science: Proceedings of the International Workshop WG80,* Bad Honnef, West Germany, (FRG), June 1980. Lecture Notes in Computer Science, Vol 100. Springer-Verlang, Berlin, 1980, pp. 250–267.
97. Smith, J.M., *SGML and Related Standards: Document Description and Processing Languages.* Ellis Horwood Series in Computers and Their Applications. Ellis Horwood, New York and London, 1992.
98. Spivak, J., *The SGML Primer.* CTI, Cambridge, Mass., and London, 1996.
99. *Gli Statuti Regionali.* Supplement to no. 1 (January 1972) of *Vita Italiana: Documenti e Informazioni.* Servizi delle Informazioni e della Proprietà Letteraria, Presidenza del Consiglio dei Ministri della Repubblica Italiana. Istituto Poligrafico dello Stato, Roma, 1972.
100. Stonebraker, M., and Rowe, L.A., The Design of POSTGRES. *Proceedings of the 1986 ACM SIGMOD Interantional Conference on Management of Data,* Washington, D.C., May 1986. *SIGMOD Record,* 15(2), June 1986, pp. 340–355.
101. Ta-Shma, I., Dates and Localities in the Life of Rabbi Zeraḥyah Ha-Levy of Lunel. (In Hebrew.) In: M.Z. Kaddari, N. Katzburg, and D. Sperber (eds.), *Bar-Ilan: Annual of Bar-Ilan University Studies in Judaica and the Humanities,* Vol. 12. Bar-Ilan University, Ramat-Gan, Israel, 1974, pp. 118–136. English summary: pp. XIII-XIV.
102. Ullman, J.D., *Principles of Database Systems.* Computer Science Press, Rockville, MA, 1982, 2nd edn.
103. Ullman, J.D., On Kent's "Consequences of Assuming a Universal Relation". *ACM Transactions on Database Systems,* 8(4), pp. 637–643 (1983).
104. van Gigch, J.P., *Applied General Systems Theory.* Harper & Row, London and New York, 2nd edn., 1978.
105. van Gigch, J.P., Modeling, Metamodeling, and Taxonomy of System Failures. *IEEE Transactions on Reliability,* R-35(2) (1986).
106. van Gigch, J.P., Beyond Modeling: Using Metamodeling for Design and Creativity. *Proceedings of the 1987 Annual Meeting of the International Society for General Systems Research,* Budapest (Hungary), June 1987.
107. van Gigch, J.P., The Metasystem Paradigm as an Inductive Methodology for Critical Thinking. *Proceedings of the 1987 Annual Meeting of the International Society for General Systems Research,* Budapest (Hungary), June 1987.

108. van Gigch, J.P., *Decision Making About Decision Making: Metamodels and Metasystems.* Abacus Press, Tunbridge Wells, Kent, England, 1987.
109. Weinberg, W., The History of Hebrew Plene Spelling. *Hebrew Union College Annual,* 46 (1975), pp. 457–487; 47 (1976), pp. 237–280; 48 (1977), pp. 291–333; 49 (1978), pp. 311–338; 50 (1979), pp. 289–317.
110. Wiederhold, G., *Database Design.* McGraw-Hill, New York, 1977.
111. Wilensky, R., *LISPcraft.* Norton, New York, 1984.
112. Wilensky, R., *Common LISPcraft.* Norton, New York, 1986.
113. Winkler, H.A., *Siegel und Charaktere in der Muhammedanischen Zauberei.* W. de Gruyter, Berlin & Leipzig, 1930.
114. Witt, K.-U., 1981 Finite Graph-Automata. In: Mühlbacher, J. (ed.), *Proceedings of the 7th Conference on Graphtheoretic Concepts in Computer Science (WG81),* June 15-17, 1981, Linz (Austria). Carl Hanser Verlag, Munich and Vienna, 1981, pp. 45–53.
115. Woodfill, J., Siegel, P., Ranstrom, J., Meyer, M., and Allman, E., INGRES Version 7 Reference Manual. Bell Laboratories, 1983.
116. *XML and Information Retrieval: Proceedings of the 2nd Workshop,* Tampere, Finland, August 2002. ACM Special Interest Group on Information Retrieval. ACM, New York, 2002.

Appendix A: Kinds of /Paˆ oL/ Among $C_1aC_2oC_3$ Hebrew Derivational Patterns

In footnote 23 to Subsec. 11.10, we referred to this Appendix. We are going to exemplify extensively from the lexicology of such terms that instantiate a given class of derivational patterns, and in order to make reference easier, we are going to number the paragraphs.

1. Sometimes, the uninflected identifier of a word-formation pattern in Hebrew (but this may be generally true also of derivational patterns in other Semitic languages as well) happens to "homonymously" refer to different patterns (possibly with the same meaning) that are inflected according to (at least partially) different inflection paradigms. This is the case of the Hebrew uninflected pattern-identifier /Paˆ oL/, of which one variant shortens the first syllable's *a* in the inflected forms, and another variant retains it instead. The latter is not as productive in the Hebrew lexicon, i.e., its derivatives are rarer, and moreover, in Israeli Hebrew no new derivatives are formed by using it any more. It was typically productive in the Roman era, because of the influence of Aramaic, and by applying it, masculine nominalised participles are generated. Moreover, both the *a*-shortening /Paˆ oL/ and the *a*-retaining /Paˆ oL/ have two subvariants, according to whether the ending of the plural is *-im* or *-in*, or is *-ot* instead.

2. Examples of *a*-shortening /Paˆ oL/ include /gadol/ (adj.) for 'big' (plural: /gdolim/); /šalom/ (noun m.) for 'peace' (plural, especially in the sense 'greetings': both /šlomim/ and /šlomot/); /qarob/ (pronounced [ka′rov], plural [kro′vim]) that as an adjective denotes 'close by', and as a noun denotes 'next of kin'.

3. In the *Babylonian Talmud,* at tractate *Siṭah,* 47b, one finds the plural compound *zeḥoḥei-hallév,* i.e., literally, 'those wanton of heart', thus 'boastful

ones', 'ones prepossessing'. As the so called constructed state (denoting 'X of') of the masculine plural of the /Paˆ oL/ term inside the compound is *zeḥoḥei-,* the masculine plural (in its absolute state) is not *zaḥoḥót,* but rather the morphologically normal *zeḥoḥím* /zḥoḥim/, with the less likely possibility that the form should be *zaḥoḥim,* as per a rare Tannaic Hebrew pattern found in the word *bazozím* for 'robbers'. (There also exists *haróg* for 'murderer', 'murderous robber', in the *Palestinian Talmud,* tractate *Qiddushín,* 66:2; the plural is *harogót.*) In the *Sifra* at *Aḥăré* 12:9, one finds the noun *nagód* for 'guide' (cf. Aramaic *nagóda*), and the plural is *nagodím.* One finds the noun *ṣaróf* 'smelter', 'goldsmith' in the *Palestinian Talmud,* at *Megillah* III, 74a, bottom (the standard Israeli Hebrew term is *ṣoréf*), and in the same sense, one finds *ṣaróv* (with a final /b/ [v] instead of /p/ [f]) in the *babylonian Talmud,* 82a, according to the version in the *'Arúkh* (a medieval glossary), the plural of *ṣaróv* being *ṣarovín.*

4. Distinguish between those four kinds of /Paˆ oL/ patterns, and the fifth pattern (typical of adjectives and names for colour), that in its inflected forms, is both *a*-shortening and last-radical redoubling (i.e., /LL/, but the convention in ONOMATURGE is to indicate redoubling by the tilde sign ˜ following the consonant it redoubles, i.e., here, /L˜/. As any transcription between slashes is phonemic, also the uninflected form reflects the consonant redoubling: /Paˆ oL˜/. Its plural form is /Pˆ uL˜im/. An example is *yaróq* for 'green', whose plural form is *yeruqqím.* This is also the pattern of the adjective *'arókh* for 'long', its plural being *'arukkím.*

5. There even is a sixth variant of /Paˆ oL/, such that the place of the stress does not move in the plural (because this is the word-form of a European modern loanword into Hebrew): this fifth pattern is instantiated in the noun *fagóṭ* for 'bassoon' (a musical instrument, in Italian *fagotto*), the plural form being fagóṭim. As in Israeli Hebrew, the most prestigious kind of pronunciation degeminates (i.e., double consonants are not audibly double), if follows that the word-form of *fagóṭ* is in practice the same as that of *gazóz* for 'fizzy drink', and the plural form is *gazózim* — but lexicographers prefer to consider the derivation pattern of *gazóz* to be /gazzoz/, by deriving it directly in Hebrew from the loanword [gaz] /gazz/ for 'gas' (the plural 'gasses' being denoted by /gazzim/).

6. In the Roman period, Hebrew used to have the noun /qaron/ for 'cart', (animal-driven) 'wagon' (it was a loanword from Latin through Greek), and the plural form was /qaronot/. But Israeli Hebrew has the noun /qaron/ especially for 'wagon' (of a train), and the plural is /qronot/. That is to say, the noun was switched from the *a*-retaining /Paˆ oL/ pattern, to the *a*-shortening /Paˆ oL/ pattern.

7. Of the *a*-retaining /Paˆ oL/, there exists in Hebrew a pattern whose plural form is /Paˆ oLot/, but there also exists in Hebrew the *a*-retaining pattern /Paˆ oL/ whose plural form is /Paˆ oLim/ or /Paˆ oLin/, and which is seldom instantiated, and most examples of which belong to Tannaic Hebrew (also known

as Tannaitic Hebrew or Middle Hebrew, from the Roman age), and are often obsolete in contemporary Hebrew.[29]

8. For example, in Tannaic Hebrew *laqóaḥ* means 'buyer', 'customer', the plural being *laqoḥót*. In Biblical and Modern Hebrew, the transitive verb *laqáḥ* means 'to take', but in Tannaic Hebrew its denotation was restricted to 'to buy', whereas the sense 'to take' was taken over by the transitive verb *naṭál*. The noun *laqóaḥ* is still in use in Israeli Hebrew, and has retained the sense it had in Tannaic Hebrew. So is the term for 'candlestick': *pamóṭ* (the plural is *pamoṭót*). In the *Tosefta* at tractate *Bava Metsi'a* 9:14, one finds *ḥafór* 'digger', and the plural is *ḥaforót*. In the *Mishnah* at tractate *Kelim* 26:5, one finds *saróq* for 'dealer in hatchelled wool of flax', 'hatcheller', the plural being *saroqót* (see *Mishnah* at *Kelim* 12:2) or *saroqín* (in the *Babylonian Talmud* at *Qiddushin* 82a according to the version in the medieval glossary *'Arukh*). Uncertainty about forms is more extensive for *šavóy* 'captor', 'enslaver', 'slaverer', plural *šavoyín* (see in the *Palestinian Talmud* at tractate *Giṭṭín* 45:4), whereas one would expect in Tannaic Hebrew *šabbay* (the plural being *šabba'im* or *šabba'in*) and in Aramaic *šabba'ah* or *šabboya* or *šavoya*, the Aramaic plural being *šabba'ei* or *šabboyei* 'captors', cf. *šabboyinhi* 'their captors' in Aramaic in the *Babylonian Talmud* at tractate *Ketubbot* 23a.

9. A term that is no longer in use, even though the concept it denotes is still current, is /qaroy/ (plural: /qaroyot/) — found in the *Palestinian Talmud,* at tractate *Megillah,* 75a — for 'congregant called to read in public a liturgical reading', such as a passage from the weekly portion of the Pentateuch (for this sense, the standard term at present is the compound *ha'olé lasséfer,* literally: 'he who goes up to [reading] the Book'), or one of the five *megillót* ("rolls" from the Hagiographa), but the present-day term is either a generic *haqqoré* (literally, 'the reader'), or *ha'olé laddukhán* (literally, 'he who goes up to the platform', but it may also refer to the lay officiant of prayer).

10. We have in Tannaic Hebrew *naṭoším* for 'fugitives whose estate is abandoned', *raṭoším* for 'absentees whose estate is abandoned and whose whereabouts are unknown, but who left of their own accord (and not fleeing under duress)', both these terms belonging to legal terminology. (In the late Roman empire, the case was frequent of landowners who left their estate, either because they had estates elsewhere, or because they wanted to escape the fiscal liabilities of being appointed a city councillor: they rather fled away, the tax burden being unbearable.)[30]

11. There also exist, in Tannaic Hebrew, *la'ozót* for 'ones who speak a foreign language and do not know Hebrew' (even though they are Jewish), and

[29] Typically, the masculine plural ending in Tannaic Hebrew is *-in* where Biblical Hebrew has *-im*. That some occurrences of terms peculiar of Tannaic Hebrew have been traded down with the *-im* ending rather than *-in* can typically be ascribed to scribal modification while copying the manuscripts (they often applyed forms with which they were familiar from Biblical Hebrew), or when the texts were prepared for printing.

[30] On the *bouleutai* (city councillors), seized lands, and squatters, see e.g. in Seth Schwartz's recent book [94], on p. 107 and 115–116 (especially the long fn. 15).

the latter term has remained in rabbinic discourse. The Tannaic Hebrew *paʿoṭót* for 'toddlers' (found in the *Mishnah* at tractate *Gittin,* 5:7) means 'toddler', and is still in use in Israeli Hebrew. But the variant *payoṭót* for 'toddlers' is obsolete; it is found in the *Palestinian Talmud* at tractate *Érubin,* 7, 24c, bottom. Tannaic Hebrew also had *masorót* for 'informers' ('traitors'), the singular being *masór*, but in Israeli Hebrew by *masorót* one understands 'traditions', being the plural of *masóret* ('tradition'). Both terms are associated with the Hebrew verb *masár* for 'to consign', just as in English both *traitor* and *tradition* (as well as *tradent,* for 'one handing down a tradition') are ultimately derived from the Latin verb for 'to consign'. In Hebrew from the 19th century, instead of *masór* for 'informer' (more precisely: 'one who consigns people unjustly to ones who would harm them') the regular Hebrew participle *mosér* would often rather be used (the plural being *mosrim*).

An inflected form of the Hebrew verb *masár* — as in the sense 'to denounce unjustly' — made its way into the 18th-century rabbinic court records (written in Judaeo-Arabic in Hebrew characters) of the city of Ṣanʿa in Yemen, that have been published by Nini [60]. In entry 897 on p. 312 in Nini's edition, recorded in the summer of the year 2076 of the Seleucid calendar, i.e., in 1765, one finds a tax collector, Yiḥya (this name is the Yemeni form of *Yaḥya*), who had been accused of embezzlement, and therefore ended up in prison for a while. Because of this, he had incurred costs that he recovered from the taxes he collected from the Jews of ʿAmrān, but part of them sued him in the rabbinic court, and the court ruled that he was in the wrong and must return the money he had recovered in that manner, and if anything, should Yiḥya have a complaint against any particular person for having denounced him unjustly (spelt in Hebrew characters as ‹ *'nh msrw*›), then Yiḥya should sue that person, and the court case would be between those two. It is quite likely that ‹ *'nh msrw*› is to be read *ánna msáru* (thus, with the verbal form being inflected as in the vernacular Arabic, the verb being a technical loanword from Hebrew fully integrated into the lexicon and grammar of Judaeo-Arabic). But it may also be that ‹ *'nh msrw*› is to be read *ánna mesaró* (thus, with the Hebrew verbal form appearing the way it is inflected according to the Hebrew grammar: the technical term would then have been inserted in its Hebrew form, as a *Fremdwort,* into the Judaeo-Arabic context).

12. In Israeli Hebrew, *pagóš* (whose plural form is *pagoším*) means 'bumper' (of a car), but in Tannaic Hebrew, that term means 'battering projectile', 'catapult stone' (in *Mishnah,* at *Kelim,* 16:8). In that sense, there also used to exist the variant *pagóz* (in Aramaic *pagóza*), whose plural form is *pagozín.* In Israeli Hebrew, that variant no longer exists, but in a modified form (*pagáz*) it means 'shell', i.e., 'projectile' (of artillery). By contrast, in Israeli Hebrew an ancient catapult stone is called *éven-balísṭra.*

13. We also have *laqoṭót* for 'poor persons who come into a field in order to collect the ears of grain that fell while being cut, and were left behind on purpose for the poor', 'grain-gleaner', and there is as well *namošót* for 'ones from amongst the weakest of mendicants', 'ones from amongst the slowest among the

poor who are allowed to collect the remainder of the crop in a field': the earliest interpretations were given in the *Talmud* itself, and vary between 'those who gather after the gatherers', and 'old people who walk with the help of a walking stick'. There also was a variant of *namošót* being *mašošót* and meaning the same (*Palestinian Talmud,* at tractate *Pe'ah,* VIII, beginning of 20d). Actually, Jastrow [33, p. 850, s.v. *mašoš*] defined it "groper, slow walker", and in the same talmudic passage, one also finds the idiom *memašméš uvá* — a couple of participles that literally means 'touching and coming' (i.e., 'feeling one's way and coming'), but idiomatically means 'coming gropingly', 'coming slowly', or 'coming nearer and nearer'.

14. In Modern Hebrew, the word *namošót* was sometimes used as a very negatively connotated word for 'epigons'. In Israeli Hebrew, there has been some metaphorical use of *namošót* (but in a substandard pronunciation: *n(e)mušót*) for 'wimps', most infamously when a politician who will remain unnamed stated that Israelis who live abroad are *nfólet šel namošót,* i.e., "scraps (*nfólet*) of *namošót*". The connotation was reversed, with respect to the historical context: he came out as being utterly inconsiderate (a constant with him), whereas the original cultural context was one of caring and sharing, even though with a streak of condescending pity. (The Israeli Hebrew noun *nfólet* for 'scrap', 'produce waste', is derived from the root for 'to fall', and therefore the semantic motivation is like that of the Italian name for the same concept, *cascàme,* and in the plural: *cascàmi* — literally, 'things fallen down'.)

15. Of *karóz* — for 'herald' (e.g., 'Temple herald' in Graeco-Roman age Jerusalem) or 'city crier' — the plural is *karozót.* That term is standard in Israeli Hebrew for 'city crier', even though in the last two centuries you would have been quite hard pressed to find any city crier around. Tannaic Hebrew *karóz* has an antecedent in Biblical Aramaic, /karoza/, in *Daniel,* 3:4. But there is possibly a relation to the Greek *κήρυξ*. Incidentally, Korazin (Chorazin) was a Roman-age town north of Capernaum, that itself was on the northern shore of the Sea of Galilee. (The spelling *Chorazin* reflects ancient phonetics: the Hebrew phoneme has currently the allophones [k] and [x] (i.e., *kh*), but the the Roman period, apparently even after a consonant or at the beginning of a word, one would hear *kh* (judging from the evidence of transcriptions into the Greek and the Roman alphabets).

16. Here is another example of the pattern /Paˆ oL/: in the homiletical collection *Genesis Rabbah,* 86, one finds the descriptor *géver qafóz* (literally, "a man [who is a] runner/jumper"), in the sense 'a runner', 'a quick man'. There is no instance of the plural of *qafóz* documented.

17. In the Hebrew Bible, the noun *taqóa'* (in the singular) for 'trumpet' or 'horn' — a musical wind instrument — " is found in *Ezekiel,* 7:14: "they blew the trumpet/horn", *taqe'ú battaqóa'*. The entry for that noun in the standard Even-Shoshan dictionary gives the plural as *taqo'ót,* thus making a choice about which variant of the /Paˆ oL/ variant to choose. The choice was in line with the typical choice that Tannaic Hebrew would make.

18. Consider the Modern Hebrew agent noun /garos/ 'grist maker' or 'grist dealer' (from the plural noun /grisim/ 'grist', for the product). The Even-Shoshan dictionary gives as correct forms of the plural both *garosím* and *garosót*. Note however that this Hebrew term may or may not be merely an adaptation of the Aramaic *garosa* (spelt *grwsh*) for 'grit-maker' or 'grist-dealer', as found more thasn once in the *Palestinian Talmud,* and of which such early rabbinic texts also give two forms of the plural in Hebrew: *garosím* and *garosót*. In the *Mishnah,* at tractate *Mo'ed Katan,* 2:5, one finds yet another agent noun, in Hebrew, *dašóš* for 'wheat-stamper' or 'groats-maker' (as grist was made by stamping upon the wheat). The term is found there in the plural form *dašošót*. In particular, the *Mishnah,* at tractate *Mo'ed Katan,* 2:5, mentions "the hunters and the *dašošót* [variant: *rašošót*] and the *garosót*". The alternation of the forms *dašošót* and *rašošót* is easily explained by the shape of the Hebrew letters for /d/ and /r/ being similar. The *dašošót* are those whose occupation was to pound or tread wheat into a pap, i.e., into grit (cf. the Aramaic noun *daysa* written with a final letter *aleph,* and the Israeli Hebrew noun *daysa* written with a final letter *he,* for 'pap', and cf. the Hebrew verb *daš* for 'to tread upon', 'to stamp upon', and the Arabic verb *dās* for 'to press', 'to squeeze'). The *garosót* instead used to pound wheat or beans into ground stuff (*grisím*) more coarse than a pap. Moreover, in the *Mishnah,* at tractate *Terumah,* 3:4, one finds (in the plural) the Hebrew agent noun *darokhót* for 'grape treaders' or 'olive treaders'. For both *dašóš* and *darókh* the etymology is quite transparently from verbs for 'to stamp upon'.

19. The standard Hebrew term for 'miller' is *ṭoḥén,* and its plural is *ṭoḥăním.* This term is in use in Modern and Israeli Hebrew, and has been in use for centuries. But in medieval Provence, Meiri[31] (writing about tractate *Pesaḥim,* p. 142 in the Jerusalem edn. of 1963/4) related that "I found that in Narbonne it happened that the *ṭaḥonót* (millers) were fixing (or: arranging) a sack full of wheat on the edge of a pit/well", and out of concern lest the what had become humid, the question rose whether that wheat could be permissibly used in order to prepare unleavened bread for Passover (quoted in [101, p. 125]). Clearly the plural *ṭaḥonót* for 'millers' instantiates a word-form available from Tannaic Hebrew. And yet, that word is not found in the extant corpus of texts in Tannaic hebrew from the Roman period. In the early (i.e., Roman-age) rabbinic literature, one does find, in the singular, the Aramaic form *ṭaḥona,* and the plural is *ṭaḥonayya* or *ṭaḥonin* (spelt *ṭḥwnyn*), and one even finds — in *Pesikta Rabbati,* 23–24 — *ṭaḥonim* (spelt *ṭḥwnym*), thus a Hebrew plural form. This suggests that the form *ṭaḥonót* was reconstituted in the Middle Ages, as both writers and readers mastered the word-formation of Tannaic Hebrew, and based on the Aramaic cognate were inspired to make up an adaptation to Hebrew, as the textual context required a Hebrew word.

20. The following example is from zoonymy (i.e., names for animal kinds). The noun *'aród* has two lexemes, and both of them denote a given animal kind. In the it Mishnah, at tractate *Kil'áyim,* 1:6, the noun *'aród* means 'onager' (a species of wild donkey). This noun is still in use in Israeli Hebrew, for the sense

[31] Rabbi Menachem ben Solomon Meiri was born in 1249 and died in 1315.

'onager', and its modern plural (e.g., in the Even-Shoshan Hebrew dictionary) is *'ărodím*. The *a* of the singular is shortened into what in traditional Hebrew grammar is considered to be a semivowel, even though the current pronunciation retains the *a*. Still, morphologically it is the *a*-shortening /Paˆ oL/ pattern that is applied to that noun. In Roman-age rabbinic texts, one finds two forms of the plural of *'aród* as meaning 'onager':

- one form of the plural that is spelt *'rwdwt* (found in the *Palestinian Talmud,* at tractate *Shekalim,* VIII, beginning of 51a), and may be either
  - *'ărodót* (thus, an instance of the *a*-shortening /Paˆ oL/ pattern),
  - or, more likely, *'arodót* (thus, an instance of the *a*-retaining /Paˆ oL/ pattern),
- and the plural form *'ărodi'ót* (in the *Babylonian Talmud,* at tractate *Menaḥot,* 103b).

21. In the early rabbinic literature, one also finds (only in the singular) the noun *'aród* (according to the usual reading of the spelling *'rwd*) denoting a kind of rather large reptile, thought to be poisonous,[32] and whose birth was fabled to be as a hybrid.[33] Note however that modern philological research has shown that apparently the original form of the noun when referring to a reptile was *'arvád*.[34] Therefore, in that sense the noun was not an instance of the pattern /Paˆ oL/.

22. It is important to realise that not all nouns or adjectives of the word-form $C_1aC_2oC_3$ (where C_i is a consonant) is necessarily an instance of one of the /Paˆ oL/ patterns. One needs to be especially careful when the first consonant is /m/ or /n/. Is that a radical (thus, /P/, the first radical), or rather a

[32] According to a tale, a sage was bitten by it. He didn't die, but the reptile died. So the sage carried the carcass of the reptile on his back, and at the house of learning told those present that as they could see, it is not the *'aród* that kills, but it is sin that kills. This tale appears in the *Babylonian Talmud*, at *Berakhot* 33a.

[33] So Rashi's commentary to the *Babylonian Talmud* at *Berakhot,* 33a: "'*Aród:* From the *naḥáš* (snake) and the *tsav* it comes [i.e., it is born], as they mate with each other and from both of them, what comes out is an *'aród*". This notion is turn is based on a passage in the *Babylonian Talmud,* at *Ḥullin,* 127a. Whereas in Israeli Hebrew, *tsav* means 'turtle' or 'tortoise', in Biblical and Tannaic Hebrew it apparently meant the same as *ḍabb* in Arabic, i.e., a large lizard: the spiny-tailed agama (*Uromastix*), but apparently also the other local large meat-yielding lizard, the desert monitor (*Varanus griseus*). Dor discussed this [14, pp. 163–165], as well as how the tortoise was referred to in those textual corpora [14, pp. 167–168].

[34] A synonym used to be *ḥăvarbar*. Dor [14, p. 163] claimed that the *'arvád* or *ḥăvarbar* could be identified with a large limbless lizard, the glass snake (*Ophiosaurus apodus,* which in Israeli Hebrew is called *qamṭán ha-ḥóreš*). When threatened, the glass snake squirts a green, smelly liquid. According to Dor, this may have been reason enough for this species to become the subject of folktales. Maybe the glass snake squirting green liquid was considered similar to poisonous snakes that release a poisonous liquid? At any rate, it makes sense that if this identification of the *'arvád* is correct, its bite was not lethal, albeit it was expected to be.

preformative that belongs in the pattern itself? In fact, the pattern of the passive participle, /niPˆaL/, becomes /naPoL/, when the middle radical (i.e., /ˆ/) is a mute *w*. Examples include the noun *mazón* 'food', whose plural /mzonot/ *mezonót* means 'foods' but also 'alimony'; and the adjectives *namókh* (phonemically /namok/) for 'low'; the medieval and modern *namóg* for 'melting away'; as well as /naloz/, whose plural /nlozim/ appears in the Bible (*Proverbs,* 2:15) in the sense 'wayward', but that in Israeli Hebrew is a rather formal term for 'contemptible'.

23. It is likely that the noun /mazor/, that in the Hebrew Bible means either 'bandage' (or 'compress') — which is the sense in *Jeremiah,* 30:13 (but also in the *Palestinian Talmud,* at tractate *Shabbat,* 5:1) — or 'ailment' (which is the sense in *Hoshea,* 5:13) was derived by applying the /maPoL/ pattern. In those sources, that noun is only found in the singular. The term is not in use in Israeli Hebrew, but the Even-Shoshan dictionary gives a plural form [mzorim]. Thus, this is an *a*-shortening form of $C_1aC_2oC_3$, as could be expected indeed of a pattern that is not /Paˆ oL/, even though both the /maPoL/ pattern and the various kinds of /Paˆ oL/ match the word-form $C_1aC_2oC_3$.

24. A clear example of the /maPoL/ pattern is the Biblical *Magog* (cf. Arabic *Majūj,* English *Magog,* Italian *Magoga*), all the more so as the context in Ch. 38 of *Ezekiel* states "Gog, from the land of Magog", and the *m* preformative often appears in the role of forming nouns for place. From the proper name *Gog* (which is apparently related to the Anatolian name preserved in the Classical sources as *Gyges* and in Akkadian as *Gugu*), Biblical Hebrew derived the root $\sqrt{\text{g(w)g}}$, whence the proper name for Gog's country, *Magog.* This is an instance of the /maPoL/ pattern, such that it happns to be the case that /P/ = /L/ = /g/.

25. We have already discussed *namošót* for 'ones from amongst the weakest of mendicants', 'ones from amongst the slowest among the poor who are allowed to collect the remainder of the crop in a field'; even that one may be (it was proposed by some) an instance of the /naPoL/ pattern (as being a particular case of the /niPˆaL/ passive participle pattern), but it is legitimate to analyse it as an instance of /Paˆ oL/ anyway (actually, such ambiguity of analysis is what historically caused the emergence of new lexical roots). As a matter of fact, Kutscher [39, Sec. 5 on pp. 98–99 (Hebrew) and p. xxxi (English)] claimed that the root of the word spelt *nmwšwt* is $\sqrt{\text{nmš}}$, and rejected Epstein's claim that the root is $\sqrt{\text{m(w)š}}$ instead (in which case, Kutscher argued, the *a* would be shortened, and the word would be *nemošot* instead of *namošot.* Kutscher was able to marshal as evidence an authoritative manuscript in which a diacritical sign for *a* appears indeed, as well as another manuscript in which the presence of an *a* is uncontrovertibly indicated by a mute *aleph* letter (thus, this being a *scriptio plena* instead of defective spelling, as the place of some vowels is indicated by the presence of letters (*matres lectionis*) with the role of being mute instead of consonantal: see [109] for the history of Hebrew plene spelling, and see [37, 20] for the emergence of *aleph* as a *mater lectionis* in the Qumran texts).

26. Also the derivational pattern /PaLon/ has the word-form $C_1aC_2oC_3$, where not all three consonants are radicals (only the first two are, the third

one being in this case "inorganic", i.e., a non-radical). An example is the noun *ratsón* (phonemically: /racon/) for 'will' (i.e., the action of wanting). It shortens the *a* in its inflected forms, e.g., (/raconi/>)/rconi/ *rtsoní* for 'my wish', and in derived terms such as the adjective /rconi/ 'voluntary' such as in *tnu'á rtsonít,* 'voluntary movement'. Also note the word (only singular) *latsón* (phonemically: /lacon/) for 'clowning' or 'pranking', for which the Even-Shoshan dictionary gives an inflected form with a shotrtened *a,* namely, *letsonó* ('his clowning', 'his pranking'). But take /garon/ 'throat'. It was apparently formed according to the derivational pattern /PaLon/, but then it was reanalysed as being an instance of the derivation pattern /Paˆ oL/, when /garon/ became itself the stem of a modern verb from which the form in use is the participle /mgoran/ *megorán* 'gutturalised'. That is to say, all three consonants of /garon/ were treated as though they were the radicals of the neologism.

27. In the *Mishnah,* at tractate *Kelim,* 11:3, names for products of metal working are enumerated, and these include the words spelt as *grwdwt* and *qṣwṣwt* which — unless they are to be pronounced *grudót* and *ketsutsót* — could rather be *garodót* and *katsotsót.* Nevertheless, instead of the spelling *grwdwt* (which is per force associated with the plural), in some manuscripts one finds the form spelt *grwdt,* thus a singular feminine term (*gródet*). In a book about metals and metal working in Jewish (especially talmudic) sources [45, Sec. 5.5, pp. 158–160], Dan Levene[35] and Beno Rothenberg explain *grwdwt* and *qṣwṣwt* as being names for different waste products of metal working, respectively 'filings and shavings', and 'cut bits of metal'.

28. The derivational pattern /Paˆ oL/, plural /Paˆ oLot/, appears to no longer be available for neologisation in present-day Hebrew. This does not mean it couldn't turn up in some given literary idiolect (i.e., in the peculiar language of some given author). In Nissan's own literary writings, there is an occurrence of *békher hannavonót* for 'young giraffe'. Literally, this compound means "*békher* ('young camel') of the *navonót* — plural of *navón*. The masculine noun *navón* is a neologism for 'giraffe' after the term *nabun*. The latter is an Africanism in Pliny the Elder's Latin treatise *Historia Naturalis*. Pliny's *nabun*, transcribed into the Hebrew script as <nbwn>, is then read as the extant Hebrew word spelled that way, i.e., /nabon/ *navón* which means 'intelligent (m. sing.). The plural of that adjective is *nevoním* 'intelligent' (m. pl.), vs. *nevonót* 'intelligent (f. pl.) being the plural of *nevoná* 'intelligent (f. sing.). Therefore, the neologised plural *navonót* 'giraffes'(which is found in a text written in the Hebrew script and including also the diacritical marks for vowels) enables differentiation, and because the vowel /a/ is maintained in the first syllable even though the stress is only on the syllable following the next, this word must be the plural of the pattern /Paˆ oL/, plural

[35] Because of their respective disciplines being far apart, it is for sure a rare thing to be able to cite, in the selfsame paper, both Mark Levene, the computer scientist from Birkbeck College of the University of London, and his younger brother Dan Levene of the University of Southampton, best known for his research about the magic spells found in incantation bowls from Mesopotamia in late antiquity. But here we are citing his book in archaeo-metallurgy.

/Paˆ oLot/. A playfully etymologising backup story for the neologism claims that being called *navón* is apt for the giraffe, because (this is true) it is the most intelligent ruminant. (Ruminants, however, are among the least intelligent among the Mammals.) In the same literary corpus, Nissan also neologised in Hebrew the name *aḥolót* for the axolotl (an Mexican amphybian), by emulating the phonetics of the Mexican Spanish and general standard Spanish name for the same animal, *ajolote,* and by adapting it into the Hebrew derivational pattern with which we are concerned. The plural ending in *-ot* as used for a word in the singular is found in Hebrew *běhemót* 'hippopotamus' (cf. *Job* 40:15),[36] and rarely in a man's name:

– *Maḥăzi'ót,* in *1 Chronicles* 25:4 and 25:30;

[36] The Hebrew word *běhemót* as being used not in the masculine singular, but in the feminine plural, means 'beasts' or 'domestic beasts'. Note morever the animal name spelled *'yšwt* or *'šwt* in the early rabbinic literature. It occurs in the *Mishnah* at *Mo'ed Katan* 1:4 and at *Kelim* 21:3. That spelling is read by some as *išut,* or then as *ešut* or *ašut* (or the Ashkenazi pronunciations *óšis, áyšis,* or *íšis*), but we cannot take it for granted that historically or in some historical receptions, there was here the ending *-ot* instead.

The *Babylonian Talmud* at *Mo'ed Katan* 6b states: "What is *'yšwt*? Rav Judah said: 'a creature that has no eyes'". This is now understood to have been the mole-rat, not to be confused for the mole. There exist no moles in nature in Israel. What does exist is the mole-rat, a blind fossorial rodent of the genus *Spalax,* now called *ḥóled* in Israeli Hebrew (by reapplication or by tentative identification of a biblical zoonym). The mole is outwardly similar, but is an insectivore, and does not belong to the rodents. The mole-rat has no eyes at all, whereas in the mole, tiny residual eyes can still be found, even though they cannot see.

In another early rabbinic text, in *Genesis Rabbah,* 51, the word *'yšwt* or *'šwt* denotes an animal "which sees not the light". The word as occurring there is used in order to explain *Psalms* 58:9 homiletically. In that verse from *Psalms,* in whose second part hemistich (i.e., second part out of two) the words *néfel éšet* occur, the sense of these is probably "a woman's miscarried fetus" and it is stated that it does not see light (because dead before being born). As the first hemistich of the same verse refers to a snail proceeding in its mucus, and as in *Psalms* semantic parallelism of the two hemistich of a verse is frequent, apparently for those two reasons, in *Genesis Rabbah* 51 it was assumed that also in the second hemistich an animal was being named.

The Hebrew zoonym *išút* is a homonym, or should we rather say, *išút* is polysemous in Hebrew. As Hebrew *iš* denotes 'man' and *ištá* denotes 'woman', the noun *išút,* formed with the abstraction suffix *-út,* means 'sexual life', such as in the compound *dinei išút* for 'norms of marital life' (from Jewish law).

Moreover, in the *Babylonian Talmud,* tractate *Ḥullin,* 62b, there is an occurrence of the Aramaic bird name *giruta,* written *g'rwt'*. In the early 20th century, the zoologist Israel Aharoni adapted it into the Hebrew neologism *girút,* which he applied to a species of waterfowl, the coot (*Fulica atra*). It is now called *agamiyyá* in Israeli Hebrew How those names for this particular bird species evolved in Hebrew is the subject of a table entry in a paper by Fischler [19, pp. 10-11, no. 1, and p. 10, note 1].

- *Měrayót,* in *Ezra* 7:3, *Nehemiah* 11:11, and *1 Chronicles* 5:32, 5:33, 6:37, and 12:15;
- *Měremót,* in *Ezra* 8:33 and 10:36, and in *Nehemiah* 3:4, 3:21, 10:6, and 12:3;
- *Měšillemót,* in *Nehemiah* 11:13 and *2 Chronicles* 28:12;
- *Šěmiramót,* in *1 Chronicles* 15:18, 15:20, 16:5, and *2 Chronicles* 17:8;
- *Lappidót,* in *Judges* 4:4 (if this is the name of Deborah's husband indeed, when she is described as "woman of Lappidot");
- *Yěrimót,* in *1 Chronicles* 7:7, 12:5, 24:30, 25:4, and 27:19, and in *2 Chronicles* 11:18 and 31:13;
- whereas *Yěri'ót* instead (found in *1 Chronicles* 2:18) is understood to have been the name of a woman.

29. The derivational pattern /PaˆoL/, plural /PaˆoLot/, also occurs in neologisation in Nissans literary writings in the plural *la'osót* for 'chewers, 'ones who chew. It is patterned after the Tannaic Hebrew word *la'ozót* for 'ones who speak a foreign language and do not know Hebrew' (even though they are Jewish), a term from rabbinic discourse for which see in §11 above. The neologism *la'osót* occurs in a playfully etymologising backup story, set in the aftermath of the Tower of Babel, and combines an explanation of why it is that a country is called *Laos* and that in Greek, λαός means 'people', 'population. That Hebrew-language story is discussed by Nissan and HaCohen-Kerner [81], as an example in the context of illustrating the control sequence in the designed multiagent architecture of the GALLURA project in automated generation of playful explanation. For the conceptual background of that project, see [80] in this set of volumes.

30. Let us conclude this appendix by quoting from an article by Osborne [84], whose subject is Early Mishnaic Hebrew, in relation to biblical Hebrew. We replace the words which appear in Osborne's original in the Hebrew script, with a transcription enclosed (according to the convention we applied throughout this paper) in single guillements. Moreover, we replace the exponents of Osborne's footnotes 53 and 54 with the respective citations of [93] and [90]. Bear in mind that MH stands for Mishnaic Hebrew, and BH stands for Biblical Hebrew:

> MH developed the qāṭôl patterns as a *nomina agentis.* In BH nouns of agency are often patterned after the participle form ‹qwṭl› . The following examples demonstrate the change observed in MH: ‹ṭḥwn› "miller", ‹srwq› "wool-comber, and ‹lqwḥ› "buyer" [93, 106]. Agency in MH is also exppressed by the suffix ‹-n› , as in ‹gzln› "robber" [90, 187].

Appendix B: Historical Strata, and Verbs That Took Over

In Subsec. 12.4, we gave an example based on the introduction to Menaḥem Moreshet's *A Lexicon of the New Verbs in Tannaic Hebrew* [58, Sec. III.4, p. 75 ff]. We are going to repeat it here in this Appendix B, and to give further

examples of how verbs belongiong to given lexical roots evolved from Biblical Hebrew to Tannaic Hebrew.

But first of all, note the following about transliteration in the following example, reflecting how the coding was done in the ONOMATURGE expert system. *Phonemic transcription* is between slashes, whereas *lexical roots* (Semitic consonantal stems) are between backslashes (but in the diagrapms as shown here, appear under the mathematical root symbol). Such roots of which there was more than one lexeme are distinguished by a numeric subscript. By `%` we transliterate the letter *aleph;* the phoneme `/%/` is a glottal stop (or zero). Inside roots, the letters `%, h, w, y` may be mute. By `$` we transcribe the phoneme /š/, which is like *sh* in the English adjective *sheer.* `/$/` is one of the two phonemes indicated by the Hebrew letter *shin,* which corresponds also to the phoneme that we write as `/S/`, and which phonetically is [s], the same as the phoneme `/s/` that corresponds to the letter *samekh.* By `H` we transliterated, in ONOMATURGE, the phoneme /ḥ/ (a voiceless pharyngeal fricative, but many Israelis pronounce it as [x], a voiceless uvular fricative). It is represented by the Hebrew letter *ḥet.* By `&` we indicate the letter and phoneme *'ayin,* which some people pronounce as a a voiced pharyngeal fricative, but others pronounce as a glottal plosive, or as zero. The phoneme expressed by the Hebrew letter *tsadi* is indicated by `/c/` and its modern phonetic value is [ts], even though historically (and in the liturgical Hebrew of Oriental Jews) it is [s̱], a velarised sibilant. By `/t/` the Hebrew letter *tav* is transcribed. By contrast, `/T/` trasncribes (in ONOMATURGE) the velarised phoneme /ṭ/, expressed by the Hebrew letter *ṭet.* We transliterate as `/q/` the phoneme expressed by the letter *qof,* that is [k] in Israeli Hebrew, but that historically (and in the traditional liturgical pronounciation of many Oriental Jews) is pronounced as [q], a voiceless uvular plosive, a consonant preserved in Modern Standard Arabic (while often not by Arabic vernaculars).

Now, let us turn to the exemplification of how verbs developed from Biblical Hebrew to Tannaic (or Tannaitic) Hebrew. According to the notation in Moreshet [58], provenience from the older, Biblical stratum (independently from its being characterised by disappearance, or decrease in use, or even increase in use), is indicated by a downward arrow, while the adoption of a stem, in Tannaitic Hebrew, to denote a certain meaning, is indicated by an upward arrow. Evolution modifies the roles of lexical entries (terms and roots) in the concerned semantic field: for example, Moreshet [58, p. 77] discussed the shrinking, from Biblical Hebrew to Tannaitic Hebrew, of the pool of verbs expressing the concept 'seeing'. Sevetral examples follow; all of them are drawn from Moreshet, and are so processed as to yield a representation in a portion of a nested relation:

A) Disappearance or shrinking use, in the Tannaic stratum of Hebrew, Biblical Hebrew verbs derived from given lexical roots, and the emergence of a new verb that took over for denoting the same sense 'to be in charge (of)':

$$\underset{\downarrow}{\sqrt{\text{šrt}}}\ ,\quad \underset{\downarrow}{\sqrt{\text{khn}}}\quad -\quad \underset{\uparrow}{\sqrt{\text{šmš}}}$$

(shrank) disappeared appeared, took over

In the frame of the root $\sqrt{\text{šmš}}$, or of the verb /šimmeš/, derived from it, and that means 'to be in charge (of)', it makes sense to include this information about the history of those verbs:

```
(EVOLUTION
  ( (FROM_STRATUM   ( Biblical_Hebrew )  )
    (TO_STRATUM     ( Tannaitic_Hebrew )  )
    (PHENOMENA
               (  (ROOT  ( $rt ) )
                  (DID   ( shrink ) )   )
               (  (ROOT  ( khn ) )
                  (DID   ( disappear ) )   )
               (  (ROOT  ( $m$ ) )
                  (DID   ( appear  take_over ) ) )
) ) )
```

B) Evolution from Biblical Hebrew to Tannaic Hebrew of verbs belonging to given lexical roots, and that denote 'to ask for'; one root shrank, another one disappeared, yet another one remained in use roughly to the same extent, and a new verb from a new root made its appearance, all of these conveying 'to ask for':

$$\underset{\text{(shrank)}}{\sqrt{\text{drš}}} \;,\; \underset{\text{disappeared}}{\sqrt{\text{nšh}_1}} \;,\; \sqrt{\text{bqš}} \;\text{—}\; \underset{\text{appeared}}{\sqrt{\text{tb\&}}}$$
$$\downarrow \qquad \downarrow \qquad = \qquad \uparrow$$

The symbol = indicates that the lexical root was shared by both strata, namely, Biblical Hebrew and Tannaic (or Tannaitic) Hebrew. The following nesting of attributes conveys a somewhat richer representation than the diagram:

```
(EVOLUTION ( (FROM_STRATUM ( Biblical_Hebrew )  )
             (TO_STRATUM   ( Tannaitic_Hebrew )  )
             (PHENOMENA    ( (ROOT    ( dr$ ) )
                             (KEYWORD ( to_look_for  to_ask_for ))
                             (DID     ( shrink ) )
                             (INTO    ( to_propose_as_exegesis )))
                           ( (ROOT    ( n$h ) )
                             (KEYWORD ( to_exact_debts ) )
                             (DID     ( disappear )  )     )
                           ( (ROOT    ( bq$ ) )
                             (KEYWORD ( to_ask_for ) )
                             (DID     ( remain )  )    )
                           ( (ROOT    ( tb& ) )
                             (DID     ( appear  share_spoils ) )
                             (INTO    ( to_demand_peremptorily ))))
)          )
```

C) Evolution from Biblical Hebrew to Tannaic hebrew of the presence of such lexical roots, that verbs derived from them denoted the sense 'to hide':

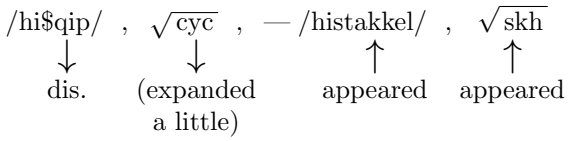
$$\begin{array}{ccccccccc} \sqrt{\text{Tmn}} & , & \sqrt{\text{Hb\%}} & , & \sqrt{\text{cpn}} & , & \sqrt{\text{str}} & -\sqrt{\text{Tmr}} & , & \sqrt{\text{kmn}} \\ = & & \downarrow & & \downarrow & & \downarrow & \uparrow & & \uparrow \\ & & \text{disappeared} & & \text{disappeared} & & \text{(shrank)} & \text{appeared} & & \text{appeared} \end{array}$$

The following nested subrelation (a subtree of attributes) conveys a somewhat richer representation than the diagram, because also some data appear that identify semantic shifts (such as specialisation into a narrower word-sense):

```
(EVOLUTION ( (FROM_STRATUM ( Biblical_Hebrew )  )
             (TO_STRATUM   ( Tannaitic_Hebrew )  )
             (PHENOMENA
               ( (ROOT     ( Tmn ) )
                 (DID      ( remain ) ) )
               ( (ROOT     ( Hb% ) )
                 (DID      ( disappear ) ) )
               ( (ROOT     ( cpn ) )
                 (DID      ( disappear ) ) )
               ( (ROOT     ( str ) )
                 (KEYWORD  ( to_hide ) )
                 (DID      ( shrink ) )
                 (INTO
                   (to_meet_alone_person_of_opposite_sex)))
               ( (ROOTS    ( Tmr  kmn ) )
                 (DID      ( appear  share_spoils ) ) )
)          ) )
```

D) Evolution from Biblical Hebrew to Tannaic Hebrew of verbs belonging to given lexical roots, and that denote 'to see'. The verb *ra'á* retained its use. The verb *hibbiṭ* disappeared in the Tannaic A stratum (i.e., the stratum mainly represented by the *Mishnah,* when Hebrew was still a living language spoken as a vernacular), but was revived in the Tannaic B stratum (i.e., in Hebrew the way it appears in the *Talmud,* by which time it was a literary language). In the diagram, we use "dis." as an abbreviation for "disappeared". The verbs *hitbonén* and *ṣafá* shrank very much. The verbs *šur, ḥazá,* and *hišqif* disappeared. The verb *heṣíṣ* expanded a little bit. The verbs *histakkél* and *sakhá,* also for 'to see', made their appearance in Tannaic Hebrew, being absent from Biblical Hebrew.

$$\begin{array}{ccccccccccc} \sqrt{\text{r\%h}} & , & /\text{hibbiT}/ & , & /\text{hitbonen}/ & , & \sqrt{\text{cph}} & , & \sqrt{\$\text{wr}} & , & \sqrt{\text{Hzh}} & , \\ = & & \downarrow & & \downarrow & & \downarrow & & \downarrow & & \downarrow & \\ & & \text{dis. in T-A,} & & \text{(shrank} & & \text{dis.} & & \text{dis.} & & & \\ & & \text{revived in T-B.} & & \text{very much)} & & & & & & & \end{array}$$



We show an excerpt from the EVOLUTION chunk corresponding to the latter diagram:

```
(EVOLUTION
  ( (FROM_STRATUM   ( Biblical_Hebrew )  )
    (TO_STRATUM     ( Tannaitic_Hebrew )  )
    (PHENOMENA      (  (VERB    ( hitbonen ) )
                       (DID     ( shrink++ ) )  )
                    .  .  .  .  .  .  .  .  .  .
  ) )
  ( (FROM_STRATUM   ( Biblical_Hebrew )  )
    (TO_STRATUM     ( Tannaitic_Hebrew_A )  )
    (PHENOMENA      (  (VERB    ( hibbiT ) )
                       (DID     ( disappear ) )  )
  ) )
  ( (FROM_STRATUM   ( Tannaitic_Hebrew_A  )  )
    (TO_STRATUM     ( Tannaitic_Hebrew_B )  )
    (PHENOMENA      (  (VERB    ( hibbiT ) )
                       (DID     ( revive ) )  )
) ) )
```

But the following:

```
( (THROUGH_STRATA ( Biblical_Hebrew
                    Tannaitic_Hebrew_A
                    Tannaitic_Hebrew_B  )  )

  (TRANSITIONS    ( (VERB  ( hibbiT ) )
                    (DID   ( disappear --> revive ) ) )
) )
```

is a shorter alternative to this portion of code:

```
  ( (FROM_STRATUM   ( Biblical_Hebrew )  )
    (TO_STRATUM     ( Tannaitic_Hebrew_A )  )
    (PHENOMENA      (  (VERB    ( hibbiT ) )
                       (DID     ( disappear ) )  )
  ) )
  ( (FROM_STRATUM   ( Tannaitic_Hebrew_A  )  )
    (TO_STRATUM     ( Tannaitic_Hebrew_B )  )
    (PHENOMENA      (  (VERB    ( hibbiT ) )
                       (DID     ( revive ) )  )
) ) )
```

Moreshet remarked that it is difficult to delimit the role of each entry (verb or root) in this example. A list of contexts is appropriate in order to try and convey their historical meaning. A suitable representational choice, at the implementational level, for this example, is indicating just the *trend* (as to diffusion and semantics) inside the `EVOLUTION` chunk in the nested relations (thus indicating the occurrence of shrinking, appearance, and so forth), possibly by distinguishing, for the verb `/hibbiT/`, between substrata of Tannaic Hebrew.

One could extend the schema described in order to account also for the influence of similar terms or roots as belonging to *adstrata* (i.e., adjacent strata), especially of languages in contact. That way, some suitable attributes in the lexical frames could provide reference to (some given stratum of) Aramaic, believed to have influenced the appearance of some term in Tannaic Hebrew.

E) This other example is about the evolution, from Biblical Hebrew to Tannaic Hebrew, of verbs expressing the lexical concept 'to fail'. The use of one root shrank, and another root disappeared, whereas a new root made its appearance.

$$\underset{\downarrow}{\sqrt{\text{kšl}}}\ ,\quad \underset{\downarrow}{\sqrt{\text{nqš}}}\ \text{—}\ \underset{\uparrow}{\sqrt{\text{tql}}}$$

(shrank) disappeared appeared, took over

Correspondingly, in the frame of $\sqrt{\text{kšl}}$ (coded as `\k$l\`); or of the verb `/nik$al/` (pronounce: *nikhšál*), itself derived from $\sqrt{\text{kšl}}$, and that denotes 'to fail'; or then in the frame of the root $\sqrt{\text{tql}}$, one may incorporate this chunk:

```
(EVOLUTION
  ( (FROM_STRATUM   ( Biblical_Hebrew )  )
    (TO_STRATUM     ( Tannaitic_Hebrew )  )
    (PHENOMENA      (  (ROOT  ( k$l ) )
                       (DID   ( shrink ) )   )
                    (  (ROOT  ( nq$ ) )
                       (DID   ( disappear ) )   )
                    (  (ROOT  ( tql ) )
                       (DID   ( appear  take_over ) ) )
) ) )
```

F) When it comes to verbs denoting 'to look for', one Biblical Hebrew lexical root disappeared, but two new roots made their appearance in Tannaic Hebrew:

$$\underset{\downarrow}{\sqrt{\text{ḥpš}}}\ ,\ \text{—}\ \underset{\uparrow}{\sqrt{\text{blš}}}\quad \underset{\uparrow}{\sqrt{\text{pšpš}}}$$

disappeared appeared, shared spoils. appeared, shared spoils.

Correspondingly, in the frame of any of the roots $\sqrt{\text{ḥpś}}$, $\sqrt{\text{blš}}$, or $\sqrt{\text{pšpš}}$ (which are respectively coded as `\HpS\`, `\bl$\`, and `\p$p$\`), denoting 'to look

for', in the subtree of the acceptation whose KEYWORD value is to_look_for, one may incorporate this chunk:

```
(EVOLUTION
  ( (FROM_STRATUM   ( Biblical_Hebrew )  )
    (TO_STRATUM     ( Tannaitic_Hebrew )  )
    (PHENOMENA      (  (ROOT    ( HpS ) )
                       (DID     ( disappear ) )  )
                    (  (ROOTS   ( bl$  p$p$ ) )
                       (DID     ( appear  share_spoils ) ) )
) ) )
```

G) Evolution, from Biblical Hebrew to Tannaic Hebrew, of verbs denoting 'to convince' or 'to entice' saw one root disappear, two roots shrink, and one new lexeme of a root make its appearance. For example, the semantics of the root $\sqrt{\text{skh}}$ shrank from 'to entice', to the legal technical term for 'to seduce' (the direct object being a maiden of minor age).

$$\underset{\text{disappeared}}{\sqrt{\text{nš'}}} \;,\; \underset{\text{(shrank)}}{\sqrt{\text{šwt}}} \;,\; \underset{\text{(shrank)}}{\sqrt{\text{pth}}} \;—\; \underset{\substack{\text{appeared,}\\ \text{took over.}}}{\sqrt{\text{šdl}_1}}$$

Correspondingly, the representation in nested relations is as follows:

```
(EVOLUTION
  ( (FROM_STRATUM   ( Biblical_Hebrew )  )

    (TO_STRATUM     ( Tannaitic_Hebrew )  )
    (PHENOMENA      (  (ROOT    ( n$% ) )
                       (DID     ( disappear ) )  )
                    (  (ROOTS   ( $wt  pth ) )
                       (DID     ( shrink ) )  )
                    (  (ROOT    ( Sdl ) )
                       (KEYWORD ( to_convince ) ) )
                       (DID     ( appear  take_over ) ) )
) ) )
```

More accurately, the following coding could be used, by incorporating it in the frame of any of the roots of the semantic field of 'convincing' or 'instigating':

```
(EVOLUTION  ( (FROM_STRATUM  ( Biblical_Hebrew )  )
              (TO_STRATUM    ( Tannaitic_Hebrew )  )
              (PHENOMENA     (  (ROOT     ( n$% ) )
                                (KEYWORD  ( to_lure ) )
                                (DID      ( disappear ) ) )
```

```
                              (  (ROOT     ( swt ) )
                                 (KEYWORD  ( to_instigate ) )
                                 (DID      ( shrink ) )
                                 (INTO
                                   ( to_instigate_to_heresy ) ) )
                              (  (ROOT     ( pth ) )
                                 (KEYWORD  ( to_lure ) )
                                 (DID      ( shrink ) )
                                 (INTO ( to_seduce_a_maiden ) ) )
                              (  (ROOT     ( $dl ) )
                                 (DID     ( appear  take_over ) )
                                 (KEYWORD  ( to_convince ) ) )
)            ) )
```

The indication — for the attribute `KEYWORD` — of the value `to_convince` (the broadest meaning in the semantic field) for the root $\sqrt{\text{šdl}}$ (coded as `Sdl`) is necessary, in order to distinguish this lexeme of the root from the lexeme instantiated in the verb `/hi$taddel/` (pronounced *hištaddél*), i.e., 'to try', 'to do efforts' (in this lexeme, the Hebrew root $\sqrt{\text{šdl}}$ is a semantic equivalent of the Arabic lexical root $\sqrt{\text{jhd}}$).

H) Barak Shemesh, at the time a student of Nissan, in late 1989 wrote a LISP program that accepts as input a `PHENOMENA` subtree of an `EVOLUTION` chunk of nested code, and transforms it into a diagram in Ascii. The input is in a file called `sample_input` and is loaded into LISP before the program is loaded. The program can only handle triliteral roots (and not, e.g., quadriliteral roots, or some specific verb such as /hitbonen/ rather than its root). Also, synonyms of attribute names were not handled (e.g., the program would take `ROOT` but not its synonym `ROOTS`).

Function `get_roots_and_events` takes the input, invokes the function `get_and_print` to find the roots to be shown, and afterwards invokes it to find the events that are connected to each root:

```
(defun  get_roots_and_events  (list)
        (terpri) (terpri)
        ( setq roots ( cdr list ) )
        ( setq counter 1 )
        ( get_and_print roots counter )
        ( setq counter 4 )
        ( setq events roots )
        ( get_and_print roots counter )  )
```

Function `get_and_print` finds all roots to be shown, and also finds the event which connects to each root. After finding all roots, it invokes the functions `print_roots` and `print_middle_lines` After finding all events, it invokes the function `print_events`

```
(defun get_and_print (object counter)
   (if
     (equal counter 1)
    then ( setq root1 ( cdaar object) )
          ( setq counter 2 )
          (setq object1 (cdr object ) )
          (get_and_print object1 counter )
    elseif
         (equal counter 2 )
    then ( setq root2 ( cdaar object ))
          ( setq counter 3 )
          (setq object2 (cdr object ) )
          (get_and_print object2 counter )
    elseif
         (equal counter 3 )
    then ( setq root3 ( cdaar object ))
              ( print_roots root1 root2 root3)
              ( print_middle_lines)
   )
   (if
     (equal counter 4 )
     then ( setq event1  (cdadar object))
          ( setq counter 5 )
          (setq object1 (cdr object ) )
          (get_and_print object1 counter )
     elseif
          (equal counter 5)
     then ( setq event2 (cdadar object))
               ( setq counter 6 )
               (setq object2 (cdr object ) )
               (get_and_print object2 counter )
     elseif
          (equal counter 6)
     then ( setq event3 (cdadar object))
               ( print_events event1 event2 event3)
)   )
```

Function `print_roots` prints the given roots in the wanted format:

```
(defun print_roots (root1 root2 root3)
 (patom '|    |)         (patom root1)
 (patom '|    ,    |)    (patom root2)
 (patom '|    ---    |)  (patom  root3)
 (terpri)  )
```

Function `print_middle_lines` handles the arrows in the output diagram:

```
(defun print_middle_lines ()
 (patom '|       !            !              ^       |)
 (terpri)
 (patom '|       !            !              !       |)
 (terpri)
 (patom '|       V            V              !       |)
 (terpri) )
```

The latter function could have been made with more sophistication, in order to handle any number of roots being involved. Besides, function `print_events` prints the given events in the output diagram:

```
(defun print_events (event1 event2 event3)
 (patom '|   |) (patom event1)
 (patom '|   |) (patom event2)
 (patom '|   |) (patom event3)
 (terpri) (terpri) (terpri)    )
```

Clearly, such a visualisation was rudimentary, even for one entirely relying on Ascii graphics. Example D, involving a large number of roots or verbs, could not have been handled. Note however that if this set is conceptualised as the first generation in a tree under one root, then one may resort to techniques for visualising even large hierarchies. One such technique was described by Lamping and Rao [48], but that one is suitable for a large hierarchy with much branching: it visualises a tree inside a circle, based on hyperbolic geometry. This is somewhat similar to graphical fisheye views of graphs [91].

Appendix C: The Code of NAVIGATION

This appendix is based on material completed in March 1990 by Jihad El-Sana, within a project supervised by Nissan. It introduces and explains the PROLOG code units of the NAVIGATE AND RETRIEVE software. For each code unit, we explain the predicate, its parameters, and the task or workings of the code unit. As can be seen, the code does handle the labels of the syntax of CUPROS, but neverthelss the code is customised for the metarepresentation of the frames of theONOMATURGE expert system for Hebrew word-formation, or then of the multilingual Semitic dictionary project that was a sequel of the ONOMATURGE project. More sophisticated software would have been really general, the way software processing XML is when using a metarepresentation of the structure of deeply nested relations.

In `navigate_and_retrieve` the parameters are `Word` (the name of the frame instance that is being searched); `Path` (a path inside the frame); and `Result` (the output information). The main program invokes the predicate `navigate` for it to look for the information using the path.

```
navigate_and_retrieve(Word,Path, Result ) :-
```

```
        consult(Word),
        assign_frame(Word,Frame_list ),
        meta('BEGIN',Meta_list),
        navigate(Path,Meta_list,Frame_list,
                 []/Fathers,[]/Keys,Result).
```

In the `meta` predicate, metarepresentation rules are stated. The first parameter is the left-hand part of the rule, the returned output being the value of the second parameter. The latter is called `Meta_list`.

```
meta( 'BEGIN'  ,    [  'IDENTITY'
                       ,'AVAILABLE_FRAMES'
                       ,'ROOTS_WITH_ALTERNANCE'
                       ,'CURSORY_LEVEL'
                       ,'DETAIL_LEVEL'
                    ]
    ).

meta('IDENTITY' , terminal ).
meta('CURSORY_LEVEL' ,  [  'ACCEPTATIONS'  ]  ).
meta('AVAILABLE_FRAMES',[  'n:','kind-of-frame_chunk']).
meta('kind-of-frame_chunk',[ 'i:','OF_KIND','ARE' ] ).
meta('ACCEPTATIONS',[ 'n:','acceptation_chunk'  ] ).


meta('acceptation_chunk',[  'i:' ,'IS'
                                 ,'ETYMOLOGICAL_ANTERIORITY'
                                 ,'ROUGH_DIFFUSION'
                             ]
).

meta('DETAIL_LEVEL' , [  'ACCEPTATION_CLUSTERS' ] ).


meta('ACCEPTATION_CLUSTERS'
                          ,[  'n:' , 'acceptation-cluster_chunk'
                             ,'n:'    ,'SEMANTIC_CHANGE'
                           ]
).


meta('acceptation-cluster_chunk'
                          ,[  'i:'   ,'IS'    /*  identifier*/
                             ,'DERIVATIVES_SPECIALIZATION'
                           ]
).
```

```
meta('DERIVATIVES_SPECIALIZATION'
                        ,[  'n:', 'particular-acceptation_chunk'
                           ,'SEMANTIC_CHANGE'
                         ]
).


meta('SEMANTIC_CHANGE'   ,[ 'x:', 'terminal_value',
                           'x:', 'n:','arc_chunk'
                         ]
    ).


meta('particular-acceptation_chunk'

                    ,[  'i:'   ,'PARTICULAR_ACCEPTATION'
                       ,'DERIVATIVES'
                    ]
).

meta('DERIVATIVES'      ,[  'n:'   ,'derivative_chunk' ] ).


meta('derivative_chunk' ,[  'i:'   ,'x:'   ,'IS'
                                   ,'x:'   ,'ARE'
                                   ,'LANGUAGE'
                             ]
).


meta('IS'       ,[  'f:'   ,['acceptation-cluster_chunk'],
                             'terminal_value'
                    ,'f:'   ,['derivative_chunk']
                         ,'x:'   ,'terminal_value'
                         ,'x:'   ,'PHONEMIC'
                              ,'VOCALIZED'
             ]
).


meta('terminal_value'
                  ,[ 'string'
                    ,'list_of_strings'
                    ,'SAME'
                    ,'logic_expression'
                  ]
).
```

```
meta('arc_chunk',[ 'i2:'   ,  'DIRECTION',
                   'i2:'   ,'x:', 'PARTICULAR_ACCEPTATION',
                            'x:', 'ACCEPTATION_CLUSTER',
                            'x:', 'SEMANTIC_CONCEPT',
                 'WHY'
              ]
   ).
```

Let us turn to the `navigate` predicate. Its first parameter is `Path`. After the parameters `Meta_list` and `Frame_list`, the next parameter is `Fathers/Next_fathers`, where `Father` is the immediate ancestry in the way down, whereas `Next_fathers` takes into account one more level of ancestry. Similarly, with the parameter `Keys/Next_keys`, by `Keys` we convey the value of the keys we meet in the way down, whereas by `Next_keys` we convey the value of the keys we meet in the way down and up. As for the `Ans` parameter, this is the information list which we are looking for.

If the meta head is terminal we finish searching and stop, and put the terminal value in the `Ans` list. If the path head is an attribute value, it's a value of a key so we invoke `key_chunk`. In order to make the path to the key name, then call `get_key_chunk` is invoked. To get the chunk which contains the key value which exist in the path, then call `navigate` recursively.

If `SAME` appear in the meta head, the info we are looking for is the head of the `Value_key` list. If `SEE` appears in the head of the meta list, then `do_see` is invoked.

It returns the answer resaulting from the SEE step. If the meta list head is the `i:` or `j:` label, then we get the value of its key by invoking the `get_one_after` predicate, then save the value thus obtained in a list. If the meta list head is `i1:` we search for two keys values and save them. We invoke `get_j_key` in order to get the first key. If the meta list head is `i2:` we take two keys from the set of of the keys by invoking `good_i2` twice. After such label processing, navigate is invoked. If the meta head is a label other than `i:` and the like, the predictae `process_label` is invoked in order to get the new meta list. Next, `navigate` is invoked recursively. In case the meta list is a list rather than an individual variable, we search for the frame in this list. If it exist, we assign its meta as being the new meta, by invoking `get_tail_of` in order to get the tail of the head in the frame list as the new frame list, and next, `navigate` is invoked. If the path head did not exist in the meta list, then it is inferred that there is something missing in the path (which is legitimate, as it is permissible to only list milestones in the path). Therefore, we get the meta of the first variable in the frame list, and `navigate` is invoked.

```
navigate([],Meta_list,Frame_list,
         Fathers/Next_fathers,Keys/Next_keys,Frame_list):-
                                  move(Fathers,Next_fathers),
                                  move(Keys,Next_keys),!.
```

```
navigate(Path,Meta_list,Frame_list,
         Fathers/Next_fathers,Keys/Next_keys,Ans):-
        list(Path,Path_head,Path_tail),
        list(Meta_list,Meta_head,Meta_tail),
        list(Frame_list,Frame_head,Frame_tail),
        ( ( ( Meta_head = terminal),
                /* if the item is terminal,
                   the value searched is reached */
              navigate([],Meta_tail,Frame_list,
                       Fathers/Next_fathers,Keys/Next_keys,Ans)
         );
         /* if the path head is an attribute value, then it is
            a key value, so we search for the key name and get
            the path to go there, then we get the chunk that
            has the same key value and continue with the rest
            of the path.
         */


          (  is_attr_value(Path_head),
             key_chunk(Meta_list,Fathers/Next_Fathers,Key_path),
             get_key_chunk(Path,Key_path,Frame_list,
                           Fathers/Next_Fathers,
                           Keys/Next_Keys,New_frame),
             navigate(Path_tail,Meta_list,New_frame,
                      Fathers/Next_fathers,New_keys/Next_keys,Ans)
          );
          ( ( Frame_head = 'SAME' ),
            head(Keys,Value_key),
            move(Value_key,Ans),
            move(Fathers,Next_fathers),
            move(Keys,Next_keys)
          );
          ( ( Frame_head = 'SEE' ),
            do_see(Frame_tail,Ans)
          );


          /* deal with the i labels or j labels */
          (   ( if_exist(Meta_head,['i:','j:']),
                head(Meta_tail,Head1),
                get_one_after([Head1],Frame_list,Key_value),
                append(Key_value,New_Keys,N_keys),
                navigate(Path,Meta_tail,Frame_list,
                         Fathers/Next_fathers,N_keys/Next_keys,Ans)
            );
```

```
   (  ( Meta_head = 'i1:'),
      list(Meta_tail,Head2,Tail2),
      get_j_key(Tail2,KK),

      append([Head2],KK,Keys_name),
      get_one_after(Keys_name,Frame_list,Key_value),
      append(Key_value,Keys,N_keys),
           navigate(Path,Meta_tail,Frame_list,
              Fathers/Next_fathers,N_keys/Next_keys,Ans)
   );


   (  ( Meta_head = 'i2:' ),
      good_i2(Meta_list,Keys1,Rest1),
      good_i2(Rest1,Keys,keys2,Rest2),
      get_one(Keys1,K1),
      get_one(Keys2,K2),
      append([K1],[K2],KEYS),
      get_one_aftre(KEYS,Frame_list,Key_value),
      navigate(Path,Meta_tail,Frame_list,
              Fathers/Next_fathers,N_keys/Next_keys,Ans)
) );


/* if it not an i label but some other label,
   then process_label will deal with this. */
(label(Meta_head),
 process_label(Meta_head,Meta_list,Fathers/New_fathers
                           ,Keys/New_key,New_meta),
 navigate(Path,New_meta,Frame_list,
          New_fathers/Next_fathers,New_keys/Next_keys,Ans)
);


/* if the meta list is a list rather than just one item,
   we check if the head of the path exists, and use this
   to continnue navigating. Otherwise, something appears
   to have been skipped in the path. */


( is_list(Meta_list),
 (  ( if_exist(Path_head,Meta_list),
      append(Fathers,[Path_head],New_fathers),
      get_tail_of(Path_head,Frame_list,New_frame),
      ((( Path_tail \== [] ),
         meta(Path_head,New_meta)
        );
```

```
              true
            ),
            navigate(Path_tail,New_meta,New_frame,
                     New_fathers/Next_fathers,
                     Keys/Next_keys,Ans)
         );
         (  meta(Meta_head,New_meta),
            append(Fathers,[Meta_head],New_fathers),
            navigate(Path,New_meta,Frame_list,
                     New_fathers/Next_fathers,
                     Keys/Next_keys,Ans)
         );
            navigate(Path,Meta_tail,Frame_list,
                    Fathers/Next_fathers,Keys/Next_keys,Ans)
   ) ) ).
```

The predicate `key_chunk` gets the path to the key in a chunk. The returned value is the value of the `Path` parameter. If the meta list head is an atom, then labels of the `i:` kind are dealt with as explained above in this appendix. By contrast, if the meta list head is `a:` or `f:` then the `processaf` predicate is invoked, and next, `key_chunk` is invoked recursively. If the meta head is a list, we invoke `key_chunk` with just the head of that list. Failing that, we invoke `key_chunk` with the tail of the `Meta_head` list. If the meta list head is an atom being an attribute name, we get the meta of this head, and invoke `key_chunk` with the new meta, and append this head to the path. If none of the above succeeds, we invoke `navigate` with the tail of the main meta list that appears as parameter.

```
key_chunk([],Fathers/Next_Fathers,Path):-fail,!.
key_chunk(Meta_list,Fathers/Next_Fathers,Path):-
       list(Meta_list,Meta_head,Meta_tail),
      ((
        /* If the head is an atom rather than a list */
        atom(Meta_head),
        ((  (
              /* if we found an i label we do as in navigate */
              if_exist(Meta_head,['i:','j:']),
              head(Meta_tail,Head1),
              move([Head1],P),
              move([P],Path)
            );

            ( ( Meta_head = 'i1:'),
              list(Meta_tail,Head2,Tail2),
              key_chunk(Tail2,Fathers/Next_Fathers,P),
              append([[Head2]],P,Path)
```

```
    );


    ( ( Meta_head = 'i2:' ),
       good_i2(Meta_list,Keys1,Rest1),
       good_i2(Rest1,Keys,keys2,Rest2),
       get_one(Keys1,K1),
       get_one(Keys2,K2),
       append([K1],[K2],Keys),
       move([Keys],Path)
 ) );


 /* If we found a: or f: then processaf is invoked */
 (   ( ( Meta_head = 'a:' );
       ( Meta_head = 'f:' )
     ),
     processaf(Meta_list,Fathers/Next_fathers,Res),

     key_chunk(Res,Fathers/Next_Fathers,Path)
)));


/* If the meta head is a list we should try to get   */
/* the path to the label via the head; failing that, */
/* we try via the rest of the list                   */


( is_list(Meta_head),
  ( key_chunk(Meta_head,Fathers/Next_Fathers,Path);
    key_chunk(Meta_tail,Fathers/Next_Fathers,Path)
) );


( atom(Meta_head),
  is_attr_name(Meta_head),
  meta(Meta_head,New_Meta),
  append([Meta_head],Fathers,N_fathers),
  key_chunk(New_Meta,N_fathers/Next_Fathers,P),
  append([Meta_head],P,Path)
);
  key_chunk(Meta_tail,Fathers/Next_Fathers,Path)
).
```

The `is_list` predicate checks whether the argument is or is not a list. If it is not a list, then failure is returned. A list has a head and a tail (i.e., whatever remains after the head of the list), so we continue until the tail is null.

```
is_list([]):- !.
is_list([H|T]):- is_list(T).
```

The `is_attr_name` predicate checks whether the variable is an attribute name, by checking that there is no lower-case letter inside it. This is because there is such a convention in the terminological and other databases that historically, RAFFAELLO had to process.

```
is_attr_name(T) :- name(T,L),!, value(L,F).
value([],F):- ( F = 1),!.
value([H|T],F):-
           ( ( H >= 65 ),( H =< 90 ),(F is 1),!,value(T,F)) ;
           (  not( low_case(H)),value(T,F)).
low_case(Char) :- ( Char >= 97 ), ( Char =< 122 ).
```

The `is_attr_value` predicate checks whether its input is an attribute value. If it is not, then the predicate returns failure. The trst carried out checks whether there is any lower-case letter inside the input string.

```
is_attr_value(Var) :- name(Var,L),!, contain_low(L).
contain_low([]):- fail ,!.
contain_low([H|T]):- ( ( H >= 97), ( H =< 122),!);
                     contain_low(T).
```

The `good_i2` predicate deals with the case that the `i2:` key label is found. If there is more than one key, then that many keys have to be handled. The program stops looking for keys once either `i_label` or the word `WHY` is found. The input parameter `I_list` is a list containing the *i*-labels to be processed. If such a label is immediately followed by an `x:` label, then the predicate `get_x_key` is invoked, in order to select the key name from the `x:` label. If there is no `x:` right after the *i*-label, it means that what does follow it is a key name. If instead of `i2:` what is found instead is `j:` then the predicate `get_j_key` is invoked, in order to get the key. If the head is an attribute name, then `good_i2` is invoked recursively.

```
good_i2(I_list,K_list,R_list):-
        list(I_list,I_head,I_tail),
        ( ( ( 'i2:' = I_head ),
             list(I_tail,Key,New_tail),
             ( ( (  Key = 'x:' ),
                 get_x_key(I_tail,K_list,R_list)
               );
               (   move([Key],K_list),
                   move(New_tail,R_list)
          )  ));
```

```
        ( ( I_head = 'j:' ),
          get_j_key(I_list,K_list),
          move([],R_list)
        );
        ( is_attr_name(I_head),
          good_i2(I_tail,K_list,R_list)
    )  ).
```

The `do_see` predicate takes as input the parameter `See_list` being the list containing the word `SEE` in its head. The output parameter is `Result` which is the processed list. The `do_see` predicate makes the path to the `SEE` information then call the predicate `do_under` to get this information according to the property and the list following the key word in the variable `Under`.

```
do_see(See_list,Result) :-
        find_by_index(2,See_list,Property),
        find_by_rang(3,See_list,Under_list),
        do_under(Property,Under_list,Result).
```

The `do_under` predicate takes as input the parameters `Property` (that is, the property that appears in the `SEE` list), and `Under_list` (that is, the list following the key word in the variable `Under`. The output parameter is `Result` which contains the information we are looking for. The variable `Under` may contain more than one under-list, so we invoke the predicate `do_one_under` in order to process just one under-list. Next, the `do_under` predicate is invoked recusively, in order to process the tail (i.e., the remnant) of the list of under-lists, once its head (i.e., the first under-list) has been taken away from the list of under-lists. The result from all the under-lists is saved in the final result list.

```
do_under(Property,[],[]) .
do_under(Property,[Under_head|Under_tail],Result) :-
             do_one_under(Property,Under_head,ResHead),
             do_under(Property,Under_tail,ResTail),
             append(ResHead,ResTail,Result).
```

The `do_one_under` predicate takes as input the parameters `Property` (being the variable following the key word property), and `Under_list`, and returns the output parameter `Result`. The `do_one_under` predicate makes the path to the `SEE` information using the `Property` and the `Under_list` it made. The path is generated by invoking the predicate `make_path` after using subservient predicates in order to collect information for path-making. (The predicate `key_chunk` constructs a path to a key.) Then, the `Property` is appended to the path, and the predicate `navigate_and_retrieve` is invoked with the root and the path.

```
do_one_under(Property,Under_List,Result):-
                     find_by_index(1,Under_List,Root_list),
```

```
                    find_by_index(3,Root_list,Root),
                    find_by_rang(1,Under_List, Under),
                    make_path(Under,Path),
                    append([Property],Path,NPath),
                    navigate_and_retrieve(Root,NPath,Result).
```

The parameters of the `get_j_key` predicate are `J_list` (the list containing all the keys to be selected) and `Keys` (the keys selected from the list). If we found the label `j:` we know that the item following it is the key name we have been looking for. If we didn't find the label `j:` as yet, then we invoke the `get_j_key` predicate recursively.

```
get_j_key([],[]):- !.
get_j_key(J_list,Keys):-
       list(J_list,J_head,J_tail),
       ( ( (J_head = 'j:' ),
           list(J_tail,Key,New_tail),
           get_j_key(New_tail,K),
           append([Key],K,Keys)
         );
         get_j_key(J_tail,Keys)
       ).
```

The parameters of the `get_key_chunk` predicate are `Path` (the path leading to the key name in the chunk thought), `key_path` (the value of the key we are looking for), `Chunks` (the list containing all those chunks which contain the key name), and `Good_chunk` (the chunk containing the key value which is equal to the value `key_path`). Once the attributes in the path are obtained (from the first three parameters in the parameter list of `get_key_chunk`), the `get_value` predicate is invoked, in order to get the the value of the key in that chunk. If this value is a sublist of the attribute list, then this is the chunk we are looking for. Otherwise, we invoke `get_key_chunk` recursively.

```
get_key_chunk(Path,Key_path,Chunks,Good_chunk):-
        get_the_2attr(Path,Attrs),
        list(Chunks,One_Chunk,Res_chunks),
        ( get_value(Path,One_Chunk,Value),
          sub(Attrs,Value),
          move(One_chunk,Good_chunk)
        );
        get_key_chunk(Path,Key_path,Res_Chunks,Good_chunk).
```

The `sub` predicate verifies whether the first list (the first parameter) is a sublist of the other list (the second parameter).

```
sub([],List):- !.
sub(List1,List2):-
```

```
        list(List1,Head,Tail),
        if_exist(Head,List2),

        sub(Tail,List2).
```

The `key_chunk` predicate constructs the path to the key in the chunk in order to return that path.

```
key_chunk([],Fathers/Next_Fathers,Path):-fail,!.
key_chunk(Meta_list,Fathers/Next_Fathers,Path):-
    list(Meta_list,Meta_head,Meta_tail),
    ( ( atom(Meta_head),
        ((  (  if_exist(Meta_head,['i:','j:']),
               head(Meta_tail,Head1), /* we have one key name  */
               move([Head1],P),       /* in order to get the   */
               move([P],Path)         /* key in []             */
           );


           (  ( Meta_head = 'i1:'),
              list(Meta_tail,Head2,Tail2),
              key_chunk(Tail2,Fathers/Next_Fathers,P),
              append([[Head2]],P,Path)
           );


           ( ( Meta_head = 'i2:' ),
              good_i2(Meta_list,Keys1,Rest1),
              good_i2(Rest1,Keys,keys2,Rest2),
              get_one(Keys1,K1),
              get_one(Keys2,K2),
              append([K1],[K2],Keys),
              move([Keys],Path)
         ) );


         (   ( ( Meta_head = 'a:' );
               ( Meta_head = 'f:' )
             ),
             processaf(Meta_list,Fathers/Next_fathers,Res),
             key_chunk(Res,Fathers/Next_Fathers,Path)
        )));


        ( is_list(Meta_head),
          ( key_chunk(Meta_head,Fathers/Next_Fathers,Path);
            key_chunk(Meta_tail,Fathers/Next_Fathers,Path)
        ) );
```

```
        ( atom(Meta_head),
          is_attr_name(Meta_head),
          meta(Meta_head,New_Meta),
          append([Meta_head],Fathers,N_fathers),
          key_chunk(New_Meta,N_fathers/Next_Fathers,P),
          append([Meta_head],P,Path)
        );
          key_chunk(Meta_tail,Fathers/Next_Fathers,Path)
       ).
```

The `get_2attr` predicate gets the first two attribute values. The input parameter is `List` and contains the set of attributes to be chosen from. The output parameters is `Attrs` which contains the chosen attributes.

```
get_2attr(List,Attrs):-
       list(LIst,Head,Tail),
       list(Tail,Head1,Tail1),
       list(Tail1,Head2,Tail2),
       append([Head],[Head1],L1),
       append(L1,[Head2],LIST1),
       ch_attr(LIST1,Attrs).
ch_attr([],[]).
ch_attr([H|Tail],R):-
        ( is_attr_value(H),
          ch_attr(Tail,R1),
          append([H],R1,R)
        );
          ch_attr(Tail,R).
```

The `get_one` predicate gets one item `R` from a list `[R|Tail]`.

```
get_one([R|Tail],R).
get_one([R|Tail],Result):-
            get_one(Tail,Result).
```

The `get_one_after` predicate is used in order to get a key value after its name. The `get_one_after` predicate takes as input the parameters `Key_names` (the names of the chunk-identifying keys), and `Frame` (the list from which we select the item), and returns the selected item, this being the value of the output parameter `Values`.

```
get_one_after([],Frame,[]):- !.
get_one_after(Key_names,Frame,Values):-
   list(Key_names,Key_name,Keys_names),
   get_tail_of(Key_name,Frame,Tail),
   list(Tail,Value,New_tail),
   get_one_after(Keys_names,Frame,N_values),
   append(Value,N_values,Values).
```

The `get_tail_of` predicate take a list (this is the parameter called `Variable`) and a frame list which contains the list which is the value of `Variable`, and returns (in the output parameter `Result`) the tail of `Variable`. If the list head is an atom, then if it equal to `Variable`, then the result is the tail of the list. If the head is a list instead, then we invoke the procedure `get_tail_of` recursively, passing to it as input the head of that list. If this fails, we look for `Variable` in the tail of the main list by invoking the procedure `get_tail_of` recursively.

```
get_tail_of(Variable,[],Result):- fail,!.

get_tail_of(Variable,
       [[ Attr_head | Attr_head_tail] | Attr_Tail],Result):-
    (  atom(Attr_head),
       (    (Variable = Attr_head ),
             put_off(Attr_head_tail,Attrheadtail),
             move(Attrheadtail ,Result)
        );
        get_tail_of(Variable,Attr_head_tail,Result);
        get_tail_of(Variable,Attr_Tail,Result)
    );
    get_tail_of(Variable,Attr_head,Result)  ;
    get_tail_of(Variable,Attr_head_tail,Result);
    get_tail_of(Variable,Attr_Tail,Result).


get_tail_of(Variable,[ Attr_head | Attr_Tail],Result) :-
   (atom(Attr_head),
            (  (Variable = Attr_head ),
               debracket(Attr_Tail,Attrtail),
               move(Attrtail ,Result)
             );

         get_tail_of(Variable,Attr_Tail,Result)
    );
     get_tail_of(Variable,Attr_head,Result);
     get_tail_of(Variable,Attr_Tail,Result).
```

The `get_x_key` predicate handles the appearance of the `x:` label. The input parameter is `X_list` (a list containing the `x:` label). The output parameters are `K_list` (a list containing the keys found in the above), and `R_list`, this being the final tail after allocating all the keys. If `x:` is found in the `X_list`, we expect the item following the label to be a key name, so we get the rest of the keys by invoking the procedure `get_x_key` recursively. All the key names are collected in `K_list` by appending them inside it. If the head of the `X_list` is not `x:` we move the list to the `R_list`, and the `K_list` is nil.

```
get_x_key([],[],[]).
```

```
get_x_key(X_list,K_list,R_list):-
      list(X_list,Head,Tail),
      ( ( Head = 'x:' ),
        list(Tail,Key_name,New_tail),
        get_x_key(New_tail,K_list,R_list),
        append([Key_name],K_list,K_list)
      );
      ( move(X_list,R_list),
        move([],K_list)
      ).
```

The `do_xy` predicate handles the co-occurrence of the `x:` and `y:` labels. The `do_xy` predicate takes as input the parameters `Xlist` (a list containing the `x:` label), and `Ylist` (a list containing the `y:` label). `Res` is the resulting list, and is the output parameter. The `do_xy` predicate divides the `Xlist` into lists containing the items which follow every `x:` and then a permutation of `x:` and `y:` is sought.

```
do_xy(Xlist,Ylist,Res):-
                  divid('x:',Xlist,Dx),
                  divid('y:',Ylist,Dy),
                  permut(Dx,Dy,Res).
```

The `process_label` predicate deals with the occurrence od labels such as `a:` or `f:` or `i:` or `x:` or `y:` and so forth, by invoking a special predicate devised for every such case. The parameters of the `process_label` predicate include `L` (a variable whose value is the label), `Meta_list` (this being the metarepresentation list), `Fathers/New_fathers`, `Keys/New_keys`, and the processed list given as the result, `Res`. If the label is `a:` or `f:`, then the `processaf` predicate is invoked. If the label is `n:` or `c:`, then we get the the meta of the item which follows the label, and next move the new meta into the `Res` parameter, and the keys into `New_keys`. If the label is `x:` or `y:`, then the `processxy` predicate is invoked, a sublist is in the result, and moreover fathers are moved into the new fathers, and the keys are moved into the new keys.

```
process_label(L,Meta_list,Fathers/New_fathers,Keys/New_keys,Res):-
         (   ( ( L = 'a:' );( L = 'f:' )),
             processaf(Meta_list,Fathers,Res),
             move(Fathers,New_fathers)
         );


         (   ( ( L = 'n:' );
               ( L = 'c:' );
               ( L = 'g:')
             ),
             find_by_index(2,Meta_list,Item),
             append([Item],Fathers,New_fathers),
```

```
        meta(Item,New_Meta),
        move(New_Meta,Res),
        move(Keys,New_keys)
    );


    (   ( ( L = 'x:');
          ( L = 'y:')
        ),
        head(Path,Path_head),
        processxy(Meta_list,R),
        get_one(R,Res),
        move(Fathers,New_fathers),
        move(Keys,New_keys)
    ).
```

The `processxy` predicate processes the labels `x:` and `y:` by subdividing the input list `List` into two lists, one for `x:` and the other one for `y:`, and then every such list is subdivided into lists that only contain one label. The result `R` is the output parameter. The predicate `do_xy` is invoked so that `x:` and `y:` would each be dealt with.

```
processxy(List,R):-
          separate('x:','y:',List,Xlist,Ylist),
          do_xy(Xlist,Ylist,R).
```

The `processaf` predicate processes the labels `a:` and `f:` by dividing the input list `List` (being the metarepresentation list) into two lists, one for the `a:` label, and the other one for the `f:` label. The first such list is placed in `Fathers`, whereas the other list is placed in `Grands`. Then every such list is subdivided into lists that only contain one label. The parameter `Fathers` contains the fathers we meet on our way down. By invoking the predicates `do_a` and `do_f` the two labels are handled separately. The list found is returned as a result in the variable `Res)` (the output parameter).

```
processaf(List,Fathers,Res):-
        super_separate('a:','f:',List,Alist,Flist),
        list(Fathers,Father,Grands),
        head(Grands,Grand),
        (  do_a(Alist,Father,Grand,Res);
           do_f(Flist,Father,Res)
        ).
```

The `get_value` predicate gets the tail of the last attribute in the path, that is to say, the predicate gets whatever, in the path, follows the attribute. This predicate is suitable for obtaining a value. The parameters are `Path` (the path of the attribute for whose value we are looking), `Frame_list` (the actual frame

instance which contains the attribute), and `Value` (the list of the values we get). If the head of the list `Path` is itself a list, then we get the value of the key in the list if it exists, by invoking to the procedure `get_one_after` and then invoking `get_value` recursively, in order to get the value of the tail, and then we append inside `Value` the values found.

```
get_value([],Frame_list,[]):- !.
get_value(Path,Frame_list,Value):-
           list(Path,Path_head,Path_tail),
           ( ( is_list(Path_head),
               get_one_after(Path_head,Frame_list,V),
               get_value(Path_tail,Frame_list,V2),
               append(V,V2,Value)
             );
             (  get_tail_of(Path_head,Frame_list,New_frame),
                get_value(Path_tail,New_frame,Value)
           ) ).
```

The `do_f` predicate divides the input list `Flist` into a list of `f:` lists, every one containing a `f:` label, that is, `Flist` into a list of lists stating the fathers nested in the list containing the attributes. Then the predicate looks for the attribute of the list which contains the father. The father for whose list we are looking is in the variable `Father`. The list of the father is returned in the output parameter `Res`.

```
do_f(File,Flist,Father,Res):-
         divid('f:',Flist,Dlist),
         find(Dlist,[Father],Res).
```

The `do_a` predicate divides the input list `Alist` into lists containing the items following every occurrence of the `a:` label. Then the predicate finds out whether the father `Father` and the grandfather `Grand` exist in the divided list. The output is in the parameter `Res` and is the list of the father which has been obtained.

```
do_a(File,Alist,Father,Grand,Res):-
         divid('a:',Alist,Dlist),
         find(Dlist,[Father,Grand],Res).
```

The following predicate checks whether a string is a label. It concludes that it is a label, if the last character in the string is a colon.

```
label(Label):-
            atom(Label),
            name(Label,Lablist),
            last(Lablist,Last),
            (Last = 58 ).
```

The following predicate finds the last item of a list.

```
last([A],A):- !.
last([H|T],L) :- last(T,L).
```

The following predicate returns success if its argument, `LABEL`, contains any of the labels `i:` or `j:` or `i1:` or `i2:` or `j2:`

```
i_label(LABEL):-
      (  ( LABEL = 'i:' );
         ( LABEL = 'i1:' );
         ( LABEL = 'i2:' );
         ( LABEL = 'j:' );
         ( LABEL = 'j1:' );
         ( LABEL = 'j2:' )
      ),!.
```

The following predicate gets the head (i.e., the first element) of the argument list, and discards the rest of the list (i.e., the tail of the list).

```
head([Head|Tail],Head):- !.
```

The following predicate instead returns the first atomic element in its input. That is to say, if the head of the list being the input is itself a list, the predicate gets its head, and if it, too, is a list, then it gets its head, and so on, recursively.

```
main_head([HEAD|TAIL],HEAD,TAIL) :- atom(HEAD),!.
main_head([Head_List | Tail_List],HEAD,TAIL) :-
                          main_head(head_List,HEAD,TAIL).
```

The following predicate thakes a list as input, and returns its head and its tail.

```
list([Head|Tail],Head,Tail):- !.
```

The following predicate checks whether the value of the input parameter, `Item`, exists inside a list or its nested sublists, recursively. If it does exist, then the predicate succeeds, otherwise the predicate fails.

```
if_exist(Item,[]):- fail,!.
if_exist(Item,[Head|Tail]) :-
            ( atom(Head), Item = Head,! );
            ( is_list(Head), if_exist(Item,Head));
            if_exist(Item,Tail).
```

The following predicate takes a list as input, and only returns its tail, by discarding the head of the input.

```
tail([Head|Tail],Tail):- ! .
```

The following predicate appends a list L2 at the end of a list L1. Recursively, when L1 becomes nil, the tail L3 is assigned to L1 and is treated as L1 in the next step of the recursion.

```
append([],L,L):- !.
append([H|L1],L2,[H|L3]) :-
        append(L1,L2,L3).
```

In the copy predicate, the first parameter is a list. The second parameter is Until, which is where the predicate must stop when copying the input list and placing the sublist until Until in the third parameter, which is returned as the result when the recursion stops.

```
copy([],Until,[],[]):- !.
copy([Head|Tail],Til,[],[Head|Tail]):- if_exist(Head,Until).
copy([Head|Tail],Until,Res,T):-
    copy(Tail,Until,L,T),
    append([Head],L,Res).
```

The divide predicate the list of labels found at the same level, into a list of such sublists that they each begin by a label. For example, if the input is

```
[a:,hghh,hghghdghg,a:,ghghghghd,a:,dhghgdhdg]
```

then the output will be:

```
[[a:,hghh,hghghdghg],[a:,ghghghghd],[a:,dhghgdhdg]]
```

The code is as follows:

```
divide(Index,[],[]):- !.
divide(Index,List,Dlist):-
  copy(List,Index,One,R),
  length(One,L),
  ( L \== 0 ),
  divide(Index,R,DL),
  append([One],DL,Dlist).
```

The separate predicate subdivides an input list into two lists. The first resulting list contains all the label L1, whereas the second resulting list contains all the label L2 and whatever follows.

```
separate(L1,L2,[],[],[]):- ! .
separate(L1,L2,[L2|Tail],[],[L2|Tail]):- ! .
separate(L1,L2,[Head|Tail],Xlist,Ylist) :-
       separate(L1,L2,Tail,X,Ylist),
       append([Head],X,Xlist).
```

Also the `super_separate` predicate subdivides an input list into two lists. The first resulting list contains all the label `L1`, whereas the second resulting list contains all the label `L2` and whatever follows.

```
super_separate(L1,L2,[],[],[]):- !.
super_separate(L1,L2,[L1|Tail],Alist,Blist):-
             copy(Tail,[L2,L1],Befor,After),
             super_seprate(L1,L2,After,Alist1,Blist),
             append([L1],Befor,A),
             append(A,Alist1,Alist).
super_separate(L1,L2,[L2|Tail],Alist,Blist):-
             copy(Tail,[L2,L1],Befor,After),
             super_seprate(L1,L2,After,Alist,Blist1),
             append([L2],Befor,B),
             append(B,Blist1,Blist).
```

The `move` predicate scans two lists of the same length. Its code is as follows:

```
move([],[] ) :- !.
move( [H | A] , [H | B] ) :- move(A,B) .
```

The `equal` predicate succeeds if the two input lists are equal it; else it fail. If the heads are atoms, they must be equal for the predicate to succeed. If the heads are lists, then the predicate is invoked recursively. The tails of the list are checked by invoking the `equal` predicate recursively.

```
equal([],[]):- !.
equal([H | T] , [H2 | T2] ) :-
       (atom(H), H = H2 ,equal(T , T2 ));
       (not(atom(H)), equal(H,H2), equal(T,T2)).
```

The `find` and `exist` predicates subserve the need to handle lists beginning by the label `a:` or by the label `f:`, by checking whether inside those lists, a nested list exists, and then the attributes found there are returned.

```
find([],Looked,Res):- fail,!.
find([Head|Tail],Looked,Res) :-
                exist(Looked,Head,Res);
                find(Tail,Looked,Res).


exist(L,[H|T],T):-
              ( atom(L),
                if_exist(L,H)
              );
              sublist(L,H).
```

The `find_index` predicate finds the place of an item in the input list

```
find_index(Item,[Item | Tail ],I):- (  I is 1 ).
find_index(Item,[Head | Tail ],I):-
                find_index(Item,Tail,J),
                ( I is J + 1).
```

The `find_by_index` predicate gets the `I`-th item in the input list, and returns that item in the output parameter `Res`. Every time we check the head, we subtract 1 from `I` until `I` is 1 (cf. `find_by_rank`).

```
find_by_index(I,[Head | List],Head):- ( I =  1 ).
find_by_index(I,[Head | List],Res):-
                ( J is I-1 ),
                find_by_index(J,List,Res).
```

The `find_by_rank` predicate returns the entire sublist beginning immediately after the item `I` (the idea being the same as in `find_by_index`).

```
find_by_rank(I,[Head | Res],Res) :- ( I = 1 ).
find_by_rank(I,[Head | Tail ],Res) :-
         ( J is I-1 ),
         find_by_rank(J,Tail,Res).
```

The `debracket` predicate removes the outer brackets [ ] from the input list. This predicate is only successful if what remains is also a list.

```
debracket([T],T):- is_list(T).
debracket(T,T):- !.
```

The `simple_list` predicate returns success if the input list is not nested. If anything in the list is not an atom, the predicate fails.

```
simple_list([]):- !.
simple_list([ Head | Tail]) :-
      ( ( not(atom(Head)),fail);
        atom(Head)
      ),
      simple_list(Tail).
```

To load all files containing the code, the `go` predicate used to be run:

```
go :- g([head,do_lab,do_one_under,do_see,
         do_under,find,get_2att,get_j_key,
         get_one,get_one_after,get_one,good_i2,
         get_x_key,navigate_and_retrieve,navigate,
         debracket,sub,list,i_label,is_attr_name,
         is_attr_value,is_list,key_chunk,label,
         main_head,simple_list,tail,append,move,
         get_value,get_tail_of,copy,divide,
```

```
         get_key_chunk,equal,meta,process_label,
         separate,exist,find_by_index,superseparate]).
g([]).
g([H|T]):- reconsult(H),g(T).
```

Appendix D: The Metarepresentation of the Lexical Frames in ONOMATURGE

This appendix shows extensive excerpts from the metarepresentation of lexical frames in the ONOMATURGE expert system. The metarepresentation of frames of formation-rules is very similar. Semicolons precede comments, as per the syntax of LISP.

```
(setq  facet_children_schema

     '(
        (pointer_atom        ( POINTER ) )

        (atom                ( INFLECTION
                               MORPHO_CAT
                               ETYMOLOGY
                               LEXEMES
                               ACCEPTATIONS
                               TOP_RELATED_TO
                               ORTHOGRAPHY
                               ETHICAL_FILTER
                               NORMATIVITY
                               INVENTED_ON    ) )

        (NORMATIVITY         ( x: normativity_info
                               x: ARE  ) )
```

The attribute `TOP_RELATED_TO` is used for example for stating homographs. As for the attribute `ARE`, consider that for those top-level facets that may have several inner facets, we have to give the top-filler as being a single list, because of the syntax of Franz LISP. This is the reason why Nissan defined the dummy level whose attribute is `ARE`, as nested inside `ACCEPTATIONS`, `LEXEMES`, `TOP_RELATED_TO`, `NORMATIVITY`, `ETYMOLOGY` (the latter attribute was not actually codedl, let alone used in ONOMATURGE), and `ORTHOGRAPHY`. This is also the reason why Nissan defined `FORMS_ARE` as nested inside `INFLECTION`. For the same reason, Nissan introduced (in principle) such dummy nonterminals as `normativity_info`, `orthographic_info`, `etymological_info`, and so forth.

```
        (normativity_info    ( NORMATIVITY_DEGREE
                               SEE_LEXEME
                               SEE_ACCEPTATION ) )
```

```
(ORTHOGRAPHY          ( x: orthographic_info
                        x: ARE  ) )


(orthographic_info
                      ( VOCALIZED
                        PLENE_NON_VOCALIZED
                        DEFECTIVE_NON_VOCALIZED   ) )


(extended_orthographic_info     ; under INFLECTION.
                      ( VOCALIZED
                        PLENE_NON_VOCALIZED
                        DEFECTIVE_NON_VOCALIZED   ) )


(INFLECTION           ( x: terminal_list
                        x: FORMS_ARE
                        y: SEE_LEXEME
                           SEE_ACCEPTATION ) )


(FORMS_ARE            ( SINGULAR_CONSTRUCTED
                        PLURAL_ABSOLUTE
                        PLURAL_CONSTRUCTED
                        SINGULAR_POSSESSIVE_MINE
                        SINGULAR_FEMININE
                        SINGULAR_CONSTRUCTED_FEMININE
                        PLURAL_CONSTRUCTED_FEMININE
                        SEE_LEXEME
                        SEE_ACCEPTATION ) )
```

The following is a shorthand: a multi-LHS, which states at once the left-hand side of the rule for several attributes, which all have the same right-hand side.

```
(   ( SINGULAR_CONSTRUCTED
      PLURAL_ABSOLUTE
      PLURAL_CONSTRUCTED
      SINGULAR_POSSESSIVE_MINE
      SINGULAR_FEMININE
      SINGULAR_CONSTRUCTED_FEMININE
    )
```

The corresponding right-hand side is as follows:

```
                    ( x: ARE
                      x: PHONEMIC
                         VOCALIZED
```

```
                                PLENE_NON_VOCALIZED
                                DEFECTIVE_NON_VOCALIZED
                            y: SEE_LEXEME
                                    SEE_ACCEPTATION ) )
```

where the second `x:` chunk is the unparenthesised childset of the attribute `extended_orthographic_info`, and where the `y:` chunk states the possibility that pointers to the attribute `INFLECTION` be found inside acceptations or lexemes.

```
(MORPHO_CAT           ( x: terminal_list_of_lists
                        x: ARE
                        y: SEE_LEXEME
                           SEE_ACCEPTATION ) )

(ETYMOLOGY           ( x: etymological_info
                       x: ARE
                       x: SEE_LEXEME ) )

(etymological_info   ( ROOT
                       PATTERN_APPLIED
                       WORD_OF_ORIGIN
                       SUFFIX_APPLIED
                       SEM_PRE_ROOTS
                       DERIVED_ROOTS
                       DERIVED_WORDS
                       DERIVED_PATTERNS
                       COMPOSED_OF
                       CONJECTURES
                       SEE_LEXEME
                       SEE_ACCEPTATION ) )
```

The attribute `COMPOSED_OF` is for compounds. In contrast, the attribute `DERIVED_PATTERNS` concerns emulative coinage (metanalysis), something that Nissan discussed in Sec. 3.6 in his Ph.D. thesis [72]. The attribute `SEM_PRE_ROOTS` ("semantics-preserving roots: a concept explained in Sec. 3.3.3.21 of Nissan's Ph.D. thesis [72]) is for either the root being the etymon, or secondary roots derived from the entry itself and keeping its semantics, or "basic elements squeezed out of the entry itself. Such roots that have many lexemes, or at any rate are very polysemous, as derivates that are semantically disparate were formed out of them, should have a very low semantic-preservation numeric degree associated, and this holds true even though the root considered is neither an actual etymon, nor a derived root

The attribute `DERIVED_ROOTS` is for derived roots. A derived root (called *néṭa'* by the linguist Uzzi Ornan [83], or "basic element) is a kind of lexical root,

which is derived from a term which is itself the derivative of some historical root. For example, the Hebrew term for ‘donation or ‘contribution is /truma/, historically from the root $\sqrt{\text{rwm}}$ of the verb /herim/ ‘to raise, the adjective /ram/ ‘high, and the like. In Tannaic Hebrew, from /truma/ the secondary root $\sqrt{\text{trm}}$ was introduced. Therefore, in the frame instance of /truma/, under the attribite ETYMOLOGY one would have to list `rwm` under `ROOT`, and `trm` under both `DERIVED_ROOTS` and `SEM_PRE_ROOTS`.

In slangish Israeli Hebrew, the verb *hizdangef* was derived from the name of the central Dizengoff Square in Tel-Aviv, and means ‘to idle in Dizengoff Square. The derived root is $\sqrt{\text{zngf}}$ (actually the last radical is a foreign phoneme, even though it could be treated here as though it was the native Hebrew phoneme /p/). This is because the /d/ is interpreted as an infix of the verbal conjugation which forms reflexive verbs. Tberefore, /d/ is excluded from the derived root.

```
(ROOT                ( terminal
                       n: root_doubleton ) )

(root_doubleton      ( i: IS
                       PLAUSIBILITY ) )

(COMPOSED_OF         ( n: lexical_doubleton ) )

(WORD_OF_ORIGIN      ( n: lexical_doubleton ) )

(lexical_doubleton   ( TERM_IS
                       KEYWORD_IS ) )

(ACCEPTATIONS        ( ARE ) )

(LEXEMES             ( ARE ) )

(ARE                 ( f: ( SYNONYMS  PARASYNONYMS )
                          n: syno_chunk
                       f: ( ACCEPTATIONS )
                          n: single_acceptation
                       f: ( LEXEMES )
                          n: single_lexeme
                       f: ( MORPHO_CAT )
                          n: HISTORY
                          n: WEIGHTED

                       f: ( TOP_RELATED_TO )
                            ASSOCIATION_OF_IDEAS
                            HOMOGRAPHS
                            PARA_HOMOGRAPHS
```

```
                                f: ( NORMATIVITY )
                                     NORMATIVITY_DEGREE
                                     SEE_LEXEME
                                     SEE_ACCEPTATION


                                f: ( ORTHOGRAPHY )
                                     VOCALIZED
                                     PLENE_NON_VOCALIZED
                                     DEFECTIVE_NON_VOCALIZED
                                     SEE_LEXEME
                                     SEE_ACCEPTATION


                                f: ( SINGULAR_CONSTRUCTED
                                     PLURAL_ABSOLUTE
                                     PLURAL_CONSTRUCTED
                                     SINGULAR_POSSESSIVE_MINE
                                     SINGULAR_POSSESSIVE_MINE
                                     SINGULAR_FEMININE
                                     SINGULAR_CONSTRUCTED_FEMININE
                                   )
          ; for each inflection variant:
                                    n: extended_orthographic_info


                                f: ( ETYMOLOGY )
                                     ROOT
                                     PATTERN_APPLIED
                                     WORD_OF_ORIGIN
                                     SUFFIX_APPLIED
                                     SEM_PRE_ROOTS
                                     DERIVED_ROOTS
                                     DERIVED_WORDS
                                     DERIVED_PATTERNS
                                     COMPOSED_OF
                                     CONJECTURES
                                     SEE_LEXEME
                                     SEE_ACCEPTATION
     )                         )
```

The latest `f:` chunk is the childset of `etymological_info`.

```
     (WEIGHTED             ( i: MORPHO_CAT_IS
                                RELEVANCE_IS ) )
```

Whereas the attribute `HISTORY` may be found under the attribute for the morphological category, `MORPHO_CAT`, by contrast the attribute `ACPT_STRATUM`

(for the historical stratum of a semantic acceptation) may be found inside an acceptation. Owing to CONTEXT, the same MORPHO_CAT_IS may refer to several contexts, with different relevances.

```
(HISTORY
                      ( x:  i1: MORPHO_CAT_IS
                              RELEVANCE_IS
                          j:  STRATUM

                        2x:  i:  MORPHO_CAT_IS
                             n:  CONTEXT

                        3x: i:  MORPHO_CAT_IS
                                DEFINE_RELATION
                            rel: MORPHO_CAT_IS
                            rel: CONTEXT

                        x:  i:  MORPHO_CAT_IS
                                RELEVANCE_IS
                                ORDINAL

                        x:  i:  MORPHO_CAT_IS
                                RELEVANCE_IS
                                CAME_AFTER
                        y:  SEE_ACCEPTATION
                            SEE_LEXEME
)                     )
```

The value of the attribute CAME_AFTER is the value of another MORPHO_CAT_IS, or some pointer to an atom or to a vertex in the tree of properties in the frame.

```
(CAME_AFTER          ( n: MORPHO_CAT_IS
                       n: POINTER       ) )


(CONTEXT      ( a: ( (HISTORY 2x ARE MORPHO_CAT) )
                   n:  stratum_information
)             )
```

Concerning CONTEXT, note that as we Nissan was unsure whether the attribute named HISTORY will remain the only one so named, in the metarepresentation schema he gave the proper ancestry of the attribute CONTEXT. Moreover, ARE is ambiguous; therefore, its father is stated, too.

```
(stratum_information     ( i: STRATUM
                           RELEVANCE_IS ) )
```

The STRATUM chunk provides "stratum_information" that is found under CONTEXT as nested inside HISTORY, in the subtree of the MORPHO_CAT attribute. Another STRATUM facet (with a terminal, or a terminal-list), may be found directly under HISTORY. Besides, ACPT_STRATUM may be found inside an acceptation.

```
(STRATUM       ( terminal_list ) )

(ACPT_STRATUM       ( terminal_list ) )


(DEFINE_RELATION     ( a: ( (HISTORY 3x ARE MORPHO_CAT) )
                          n: relation
)                    )


(single_lexeme       ( i: LEXEME_KEYWORD
                       INFLECTION
                       MORPHO_CAT
                       ETYMOLOGY
                       RELATED_TO
                       ACCEPTATIONS ) )


(single_acceptation  ( c: MEANING
                       CONNOTATIONS
                       RELATED_TO
                       AURO
                       AURC ) )


(CONNOTATIONS        ( POS/NEG
                       QUALITIES_OF_CONCEPT
                       QUALITIES_OF_TERM/ACPT ) )
```

An example at the object-level representation could be as follows:

```
          (QUALITIES_OF_CONCEPT  ( huge
                                   strong
                                   slow
                                   unaesthetic
                                   (HAS memory)
                                   (PRODUCES ivory)
          )                      )
```

as a simplified way to state qualities of elephant. Frame-instances found may include a more complex format of this attribute, including information on typicality and necessity. Now, let us revert to the metarepresentation.

```
(LIKELY_CONTEXT      ( n: CONCEPT_IS ) )

(CONCEPT_IS          ( ATOM_IS
                       KEYWORD_IS ) )

(CONTRARY            ( ABSOLUTE
                       RELATIVE ) )

(RELATIVE   ( n: contextual_contrary_doubleton ) )

(contextual_contrary_doubleton      ( i: IS
                                      CONTEXT ) )

(RULES               ( POLYTEMPLATES
                       SUFFIXES
                       PREFIXES
                       IDIOMS
                       COMPOUNDS
                       ROOT_PATTERNS
                       REDUPLICATIVE_PATTERNS ) )
```

Root patterns are clusters of related roots, either alloroots, or a primary root with some related reduplicative root. This is rather frequent in Semitic languages.

```
(POLYTEMPLATES       ( n: rule_doubleton ) )

(SUFFIXES            ( n: rule_doubleton ) )

(PREFIXES            ( n: rule_doubleton ) )

(IDIOMS              ( n: rule_doubleton ) )

(COMPOUNDS           ( n: rule_doubleton ) )

(rule_doubleton      ( i2: IS
                       RO
                       j: INFLECTION_PARADIGM
                       j: MORPHO_CAT_IS       ) )

(INFLECTION_PARADIGM  ( FORM_IDENTIFICATION
                        INFLECTED_FORM_IS     ) )
```

```
(PECULIAR_FACETS     ( g: PECULIAR_SLOTS
                       etc    ;  the slots listed
                              ;  in  PECULIAR_SLOTS .
                       TOOLS
                       NEXUS   ;  Event/state description
)                    )         ;  a‘ la NEXUS .

(NEXUS               ( SC      ;  subclass.
                       SUBSEQ  ;  subsequence.
                       COORD   ;  coordinated action.
                       ANTE    ;  antecedent.
                       PREC    ;  precedent.
                       CONSEQ  ;  consequent.
                       SEQ     ;  sequence.
)                    )
```

NEXUS, described by Alterman in 1985 [4] (Alterman had just earned a Ph.D. at the University of Texas in Austin, then moved to Berkeley), is a text-understanding system that uses a dictionary of event/state concepts whose relation is described in a network with various links.

```
(SYNONYMS            ( ARE ) )

(PARASYNONYMS        ( ARE ) )

(syno_chunk          ( i: IS
                       ORTHOGRAPHY
                       KEYWORD_IS
                       DIFFERENCES
                       RELATIONSHIP
                       REL_FREQ_VS_FRAME_OWNER ) )
```

`ORTHOGRAPHY` as nested in `syno_chunk` is optional: it is useful in case you state a rare synonym, or a mere variant, that does not deserve a frame-instance on its own, but has such an idiosyncratic spelling that it could not be reconstructed by a "phonemic antitransform. `KEYWORD_IS` as nested in `syno_chunk` is also optional; it is necessary however when near-synonyms are listed, if keywords are not the same.

The attribute `RELATIONSHIP` may occur as a grand-grandchild of the attribute `PARASYNONYMS`, and then it indicates such a relationship that it involves concepts (as opposed to terms): for example, values could be

```
( (metonymy: kind_contains ) )
```

or

```
( ( metonymy: kind_contained ) )
```

On the other hand, RELATIONSHIP may appear as a grand-grandchild of SYNONYMS, too, even in order to provide some unsophisticated indications about the term itself in a cross-linguistic framework:

```
( (standard) )
```

or

```
( ( cross-language: Phoenician ) )
```

or

```
( ( cross-language_reconstruction: Phoenician ) )
```

The metarepresentation rule for RELATION is as follows:

```
(RELATIONSHIP        ( terminal_list ) )
```

According to the following metarepresentation rule, under the attribute COMPONENTIAL_ATTRIBUTES a local metarepresentation is given inside the frame-instance, with instance-specific facets.

```
(DIFFERENCES         ( g: COMPONENTIAL_ATTRIBUTES
                       ; a local metarepresentation.
                       etc
)                    )


(COMPONENTIAL_ATTRIBUTES
          ( terminal
            n: componential_attribute_doubleton ))


(componential_attribute_doubleton  ( i: IS
                                       VALUESET
                                       VALUE_TYPE
                                       VALUE_ACTION ) )

(VALUE_ACTION        ( n: value_action_doubleton ) )


(value_action_doubleton    ( i: IF
                              THEN  ) )


(THEN                ( terminal
                       CONSULT_ATTRIBUTE ) )

(HOMOGRAPHS          ( n: homograph ) )
```

```
        (homograph            ( IN_WHAT_WRITING
                                i: STRING_IS
                                INFLECTED_FORM_IS
                                HOMOGRAPHY_IS_WITH ) )


        (para_homograph       ( IN_WHAT_WRITING
                                i: STRING_IS
                                INFLECTED_FORM_IS
                                HOMOGRAPHY_IS_WITH ) )
```

The use of `HOMOGRAPHY_IS_WITH` is in order to deal with situations where there are roots similar to some other roots, because of either etymology, or coincidence (and then, there is potential for puns).

```
        (INFLECTED_FORM_IS    ( f: ( homograph )
                                   IS
                                   MORPHOLOGICAL_ANALYSIS
                                f: ( INFLECTION_PARADIGM )
                                   terminal
        )                     )


        (HOMOGRAPHY_IS_WITH   ( IS
                                MORPHOLOGICAL_ANALYSIS
                                ATOM_OF_NON_INFLECTED_IS ) )

      )
]
```

According to the syntax of Lisp, the] character closes all parentheses still open.