RLL-1: A Representation Language

Russell Greiner

Computer Science Department
Stanford University
Stanford, California 94305

October 1980

Representation, Knowledge, Language, Self-Description, Self Modification, Expert Systems
DISCLAIMER NOTICE

THIS DOCUMENT IS BEST QUALITY PRACTICABLE. THE COPY FURNISHED TO DTIC CONTAINED A SIGNIFICANT NUMBER OF PAGES WHICH DO NOT REPRODUCE LEGIBLY.
RLL-1:
A Representation Language Language

by

Russell Greiner
Computer Science Department
Stanford University

Expanded version of the paper published in the
Proceedings of the First National Conference
of the
American Association of Artificial Intelligence
(Stanford University, August 1980)

Heuristic Programming Project
Computer Science Department
Stanford University
Stanford, California 94305

DISTRIBUTION STATEMENT A
Approved for public release;
Distribution Unlimited
# Table of Contents

1. MOTIVATION ................................................................. 3

2. INTRODUCTION ............................................................ 3

3. OVERVIEW of RLLs ....................................................... 5

4. EXAMPLE -- Creating and Using a Type of Slot ......................... 9
   4.1 Definition of "Type of Slot" ........................................ 9
   4.2 Retrieving a Slot's Value (Husband) ............................... 10
   4.3 Creating a New Type of Slot (Father) ............................. 10
   4.4. Other Facts Stored on the Type of Slot Unit .................. 13
   4.5 Modifying an Existing Type of Slot ............................... 14

5. DESCRIPTION of RLL-1 .................................................. 16
   Main Ideas/Philosophy .................................................. 16
   5.1 Units and Slots .................................................... 16
   5.2 Cognitive Economy ................................................ 17
   5.3 Syntactic vs. Semantic Slots ...................................... 17
   5.4 Onion field of sub-Languages .................................... 18
   5.5 Economy via Appropriate Placement ............................... 18
   5.6 Clarity of Conceptual Units ....................................... 19

6. SPECIFICATIONS FOR ANY Representation Language Language .......... 19
   6.1 Self-Description .................................................... 19
   6.2 All Parts are Modifiable .......................................... 20
   6.3 Epistemological Adequacy ......................................... 20
   6.4 Linguistic Adaptability ............................................ 21
   6.5 Representational Adaptability .................................... 21
   6.6 Unbiased initial system ............................................ 22
   6.7 Codification of Representational Knowledge ..................... 22

7. COMPARISON WITH OTHER SYSTEMS .................................. 23

8. CONCLUSION ............................................................... 25

Acknowledgements ............................................................ 26

(Appendices are listed on the next page)
APPENDICES

A. Use of the *Father* Slot ................................................. 27
   A.1 How (GetValue U S) really works ................................ 27
   A.2 How (and why) caching really works ............................ 28
   A.3 Other Accessing Functions ....................................... 28

B. Additional Examples
   B.1 Creating a Whole New Family of Types of Slots .............. 30
   B.2 Overview of EURISKO system ..................................... 32
   B.3 Creating a new inheritance mode ............................... 34
   B.4 Epistemological Status .......................................... 34

C. Syntactic vs. Semantic Slots, Revisited ............................. 36

D. Implementation Principles
   D.1 Make Standard Case Fast ......................................... 38
   D.2 Least Commitment ................................................ 38
   D.3 Compile Time Macros ............................................ 38
   D.4 Special Slot Values ............................................. 38

Bibliography ........................................................................ 41

*Included in the Supplement:*

E. Special Units .................................................................... 27pp
   Naming Conventions, Legend, Actual Units, Index of Units

F. Environment ....................................................................... 12pp
   Top Level Functions, Functions needed to Bootstrap RLL-1,
   Convenience Functions, Advised Functions, Global Variables

G. Sample Session ................................................................ 26pp
RLL-1: A Representation Language Language

The field of AI is strewn with knowledge representation languages. The language designer typically designs that language with one particular application domain in mind; as subsequent types of applications are tried, what had originally been useful features are found to be undesirable limitations, and the language is overhauled or scrapped. One remedy to this bleak cycle might be to construct a representation language whose domain is the field of representational languages itself. Toward this end, we designed and implemented RLL-1, a frame-based Representation Language. The components of representation languages in general (such as slots and inheritance mechanisms) and of RLL-1 itself, in particular, are encoded declaratively as frames. By modifying these frames, the user can change the semantics of RLL-1's components, and significantly alter the overall character of the RLL-1 environment.

1. MOTIVATION

Often a large Artificial Intelligence project begins by designing and implementing a high-level language in which to easily and precisely specify the nuances of the task. The language designer typically builds his Representation Language around the one particular highlighted application (such as molecular biology for Units [Stefik], or natural language understanding for KRL [Bobrow & Winograd] and OWL [Szolovits, et al]). For this reason, his language is often inadequate for any subsequent applications, except those which can be cast in a form similar in structure to the initial task. What had originally been useful features are subsequently found to be undesirable limitations. Consider Units' explicit copying of inherited facts or KRL's sophisticated but slow matcher.

Building a new language seems cleaner than modifying the antiquated one, so the designer scraps his "extensible, general" language after its one use. The size of the February 1980 SIGART shows how many similar yet incompatible representation schemes have followed this evolutionary path.

One remedy to this bleak cycle might be to construct a representation language whose domain is the field of representation languages itself, a system which could then be tailored to suit many specific applications. Toward this end, we (Professor Douglas Lenat and I) have designed and implemented RLL-1, an object-centered Representation Language Language.

2. INTRODUCTION

A representation language language (rll) must explicitly represent the components of representation languages in general and of itself in particular. This technique of self-encoding gives the rll its flexibility and adaptability: consider the versatility inherent in the programming language LISP, which also encodes all of its constructs (i.e. programs) within its own formalisms (i.e. as S-expressions). Like all representation languages, the rll should provide an easy, natural interface to users. As such, its primitive building blocks are necessarily larger, more abstract, and more complex than the primitives of programming languages.

---

1 This should be pronounced "RLL negative one" -- as we feel this implementation does not qualify as a real representation language language, yet. Section 6 provides the definitions necessary to realize RLL-1's shortcomings.
2 This "object-centering" does not represent a loss in generality. We will soon see that each part of the full system, including procedural information, is reified as a unit.
3 As a representation language language should itself be a completely self-descriptive representation language, there is no need for an RLLL.
4 Technically, an rll is a Representation Language, that is, a "Language about Representation". It attempts to describe representation, in the same way a VLSI Language is designed to deal with VLSI components. To avoid confusion with earlier languages, notably KRL and FRL, which have already coined "RL" to mean "Language for Representation", we gave RLL-1 its second "L".
Building blocks of a representation language include such things as control regimes (ExhaustiveBackwardChaining, Agendas), methods of associating procedures with relevant knowledge (Footnotes, Demons), fundamental access functions (Put/Get, Assert/Match), automatic inference mechanisms (InheritFromEvery2ndGeneration, InheritButPermitExceptions), and even specifications of the intended semantics and epistemology of the components (ConsistencyConstraint, EmpiricalHeuristic).

The purpose of an rll is to help manage these complexities, by providing (1) an organized library of such representation language components, and (2) tools for manipulating, modifying, and combining them. Rather than produce a new representation language as the "output" of a session with the rll, it is rather the the rll system itself, reflected in the environment the user sees, which changes gradually in accord with his commands.

Rather than design his representation language from scratch, the AI researcher should be able to use an rll as the "starter" for almost any AI task -- including natural language understanding, ICAI (Intelligent Computer Aided Instruction), designing expert system, or theorem proving. The researcher's first step will be to load in his rll, and set its "parameters" to values appropriate for his domain and application. The rll will then take over, composing the desired new language from these specifications. If ever the researcher wishes to alter this language, he need only issue the appropriate command to the rll system, (using the same format he used, for example, to enter new data) and the language he sees will be modified.

This paper serves several functions, and is intended for several audiences. The first class of readers include those AI researchers who are looking for such a starter system, and are intrigued with this notion of a versatile, self-descriptive system. They should regard this document as an RLL-1 Primer, which describes the advantages and power of an rll, followed by a how-to manual describing this RLL-1 system.

A second (not necessarily distinct) group of readers are researchers interested in the epistemological and representational issues associated with designing and using a self-embedded system. This memo, therefore, includes sections which define what we mean by the term, Representation Language Language, and which show the ways our implementation, RLL-1, handles several potentially difficult problems.

We report here on the current state of both our ideas regarding rlls in general, and our RLL-1 implementation, in particular. The beginning portion, sections 1 through 4, motivates the idea of a rll, illustrating this description with examples taken from the RLL-1 language. Section 3 suggests the facilities an rll must provide, by showing what the RLL-1 user initially sees, and the ways this environment can be modified. Section 4 steps through a long example, indicative of the types of things RLL-1 can do.

This leads up to Section 5, in which a high level description of RLL-1 itself is presented. Section 6 then shows how RLL-1 fit into the class of rlls in general, by specifying a set of criteria which any rll must satisfy. These rlls are then compared with other systems, of various natures, in Section 7.

The appendices provide many of the details intentionally omitted from earlier sections. Appendices A, B and C each elaborate some earlier section of this document: A expands the example presented in Section 4, B sketches a few more examples which use RLL-1, including a coarse outline of a full application, while C demonstrates a point made in Section 5.3. Appendix D states a few of the implementation level principles we followed in designing RLL-1.

Appendices E, F and G comprise a companion paper, [Greiner]. These supply the details necessary to actually use this RLL-1 system. Appendix E lists many of the "core" units resident in the initial system. Appendix F describes the RLL-1 environment -- including the top level functions along with other information a novice can use when working with RLL-1. A sample dialogue is presented in Appendix G, to show in detail how to communicate with this current system.
3. OVERVIEW of RLLs
or HOW IS A REPRESENTATION LANGUAGE LIKE AN ORGAN?

When the user starts RLL-1, he finds himself in a LISP environment which contains functions which closely resemble those resident in the Units package [Stiefik]. There is one major difference: whereas the facilities Units offered were essentially unalterable, RLL-1’s features are designed for easy modification. If this user desires a new type of inheritance mechanism, he need only create a new Inheritance-type of unit, and initialize it with the desired set of properties. At this point, that new mode of inheritance will automatically be enabled, and usable. This can be done using the same editor and accessing functions used for entering and codifying his domain knowledge (say, VLSI design); only here the information pertains to the actual Knowledge Base system itself, as opposed to chips and diffusion layers.

The Units package has Get and Put as its fundamental storage and retrieval functions; RLL-1 also begins in that state. But there is nothing sacred about even these two “primitives”. “Get” and “Put” are themselves encoded as (modifiable) units; if they are altered, the nature of accessing a slot’s value will change correspondingly. In short, by issuing a small number of commands the user can radically alter the character of the RLL-1 environment, and thereby mold it to his personal preferences and to the specific needs of his application. RLL-1 is responsible for performing the necessary “truth maintenance” operations, (e.g. retroactive updates) to preserve the overall correctness of the system as a whole. As an example, RLL-1 recently “learned” how to deal with a more “representationally neutral” representation, one which lacked slots altogether. It was then able to deal with Assert and Match commands in lieu of its native Gets and Puts; and consider propositions, rather than slots, as its most primitive “container” for information.

An rll is more like a stop organ than a piano. Each stop corresponds to a “pre-fabricated” representational part (e.g. a slot, inheritance, format, control regime, etc.), which all reside in the overall rll system. The initial rll system is simply one configuration of this organ, with certain stops “pulled out” -- in RLL-1’s case, to mimic useful aspects of the Units package. These particular stops reflect our intuitions of what constitutes a general, powerful system. For example, some of the units initially “pulled out” (activated) define various standard inheritance regimes, such as Inherit-Along-IS-A-Links, which stores on Fido default information gathered from TypicalDog.

We chose to include a large assortment of common slots. There are fifteen types of slots in the initial RLL-1 system, including ToGetValue, ToPutValue, MyToKillUnit, and ToAddValue, which collectively define the accessing/updating functions. Another group of over one hundred types of slots, including IS-A, SuperClass, BroaderHeuristics, and TypicalExamples, are used to hierarchically organize the units. Over two hundred other additional types of slots are used to interrelate units in other useful, if non-hierarchical ways -- e.g. NeighboringCountries, RangeOfFunction, SimilarHeuristics, etc. (Each of these slots presuppose there are units representing countries, functions, range-spaces, and heuristics. We will expand on this “to every idea, a unit” motif in Section 5.5.)

The number of these slots grow daily. As new domains are explored, new concepts become realized as units. It is only natural to relate these to one another; and slots provide an obvious mechanism for describing and storing such connections. Many such domain-specific slots are used to induce (or record) an organization among these units. Examples include FunctionalExtension which points from a (unit representing a) function to those (units which represent) functions over a strictly larger domain: LocatedInRegion, which connects each geological region with the larger space in which it is located (e.g. mapping each country into a continent); or the familiar PeckingOrder, defined for poultry. Others serve to connect a new concept with other existing concepts -- such as BelongsTo or NumberOfElements. New slots also emerge as we refine the organizing relationships which were originally “smeared” together into just one or two kinds of slots. Consider, for example, the way IS-A, TypicalExampleOf and SuperClass were undifferentiated in A-Kind-Of, or the relationship

\[ RLL-1 \text{ is a frame-based system [Minsky], whose building blocks are called Units [Stiefik], [Bobrow & Winograd].} \]

In the initial RLL-1 system, each unit consists of a set of Slots, each with its respective value.
between LocatedInRegion and each of LocatedInContinent, LocatedInCountry, and LocatedInCity.

This bootstrapping system (the initial configuration of "organ stops," does not span the scope of RLL-1's capabilities: many of its stops are initially in the dormant position. Just as a competent musician can produce a radically different sound by manipulating an organ's stops, so a sophisticated RLL-1 user can define his own representation by turning off some features and activating others. For instance, an FRL (Roberts&Goldstein) devotee may choose to use exclusively the kind of slot called A-Kind-Of, mentioned above. He may then deactivate those more specialized units (viz. IS-A, Abstraction, TypicalExampleOf, PartOf, etc.) from his system permanently. Another user who did not want to see his system as a hierarchy at all could simply deactivate all of these A-Kind-Of kinds of slots. He need not worry about the various immediate, and indirect, consequences of this alteration (e.g., deleting the Inherit-Along-IS-A-Links unit); RLL-1 will take care of them.

The alterations could be constructive as well as destructive. We saw above how LocatedlnRegion can be expanded into more precise slots, and the more dramatic example in which the user abandoned slots altogether, replacing "Get" and "Put" units with others that acted as Assert (store proposition) and Match (retrieve proposition). As another example, a user who wanted to distinguish between proper and improper subsets could define the corresponding ProperSubSet and ImproperSubSet slots.

Any representational piece can be altered. In the examples which appear later in the paper we show how inheritance mechanisms, formats, datatypes, and the like can be changed. All such "tabs" can be manipulated, and it is by selectively pushing and pulling them the user is able to fabricate his personalized system. The versatility of this approach will be demonstrated by the ease with which the rll can be reconfigured to resemble any currently-used representation language, such as KRL, OWL and KLONE.6 after all, an organ can be made to sound like a piano!

To understand RLL-1's self-encoding capability, within this organ metaphor, imagine our organ is without tabs. Instead of changing the organ's tonality by pulling out some stop, we change its tonality by playing a certain prescribed melody. This sequence of keystrokes will have two effects: in addition to outputting the expected sequence of sounds, it will also cause the organ to undergo a state change, emerging, say, a glockenspiel-simulating instrument. For example, playing the first three chords of Hindemith's "Traumemusik" would instruct this organ to go into its "viola-mode." In this state, the sounds it produces all have a viola-like quality. If the organist then played the (meta-)sequence corresponding to "Greensleeves," this organ will gradually ease into its flute-mode; and so forth.

The catch is that the melody required for the Viola-to-Flute metamorphosis is different from the sequence to go from Sackbut-to-Flute -- i.e. the (meta-)effect of a given sequence depends on the current state (i.e. "instrumentality") of the organ. Playing the "Traumemusik" when the organ is not in its initial state may have no meta-effect at all; or it may, for no particular reason, switch the organ into its Oboe state. That is, the sequence which triggers a change to the organ is a function of the organ's state; so as the organ changes, so does its range of permissible alterations.

To be more precise, the organ's state follows a Markov process. Each special keystroke, by itself, may induce a minor incremental change; and it is the sum of these small effects that produces the new gestalt.

With this metaphor in mind, consider the complications of converting from slots to propositions. Initially all the facts in the knowledge base are stored as the values of slots, on various units. The first change may be to create and store the procedure for asserting some fact. Note this collection

---

6 This particular task of actually simulating various existing representation languages, has not yet been done. It is high on our agenda of things to do. We anticipate it will require the addition of many new components (and types of component) to RLL-1, many representing decompositions of the space of knowledge representation orthogonal to the ones we now use.
of facts will still be stored as slots, as RLL-1 is still in its "Slot-Mode". As such, these proposition-related facts cannot yet be utilized. They can only be "activated" after a host of other units, (such as Match and AnyProposition, which are also not yet usable) have been created. At this point, RLL-1 is able to become truly proposition-based. The first operation involves fixing up the existing knowledge base, i.e. recoding every fact now stored as a slot as a proposition, taking pains to perform these alterations in the correct order. Only then are these proposition-units -- e.g. TypicalProposition -- actually usable, and capable of describing themselves. RLL-1 has, at this point, eased its way into a new and different format for encoding information. The rll must have the smarts to guide the transformation, changing itself as it changes the data. It must know not only what to change, but when to perform this alteration, and be able to determine and use the ramifications of such changes as well.7

The initial RLL-1 system has been designed to be extremely flexible, and able to adopt readily to a new specified set of conventions. Note that this itself is just a particular convention, and is not sacred. Nothing prevents a cautious (or masochistic) user from eventually deactivating the modification facilities themselves. This would effectively lock him into the particular set of conventions then currently active.

Unlike musical organs, RLL-1 also provides each user with mechanisms for building his own stops (or even type of stops, or even mechanisms for building stops). An experienced RLL-1 user can use this system to build his own new components. Rather than building them from scratch, (e.g., from CAR, CDR, and CONS,) he can modify some existing units of RLL-1, employing other units which are themselves tools designed for just such manipulations.

The following examples may help solidify these abstract ideas:

---

7 As an intriguing research effort, RLL-1 might eventually have the "smarts" necessary to make these changes automatically. That would be like a smart organ, which knew enough to convert itself into its Lute mode whenever it started playing a Renaissance sounding piece of the appropriate harmonic range and tonality. A next step would be the ability to synthesize new sounds when appropriate, as opposed to simply retrieving some pre-existing instrumentalities. This would be like deciding a cannon boom would be appropriate for the "1812 Overture", based on its opening movements, and generating this new type of sound.
Figure #1: Selected members of the Bach family, with only Mother and Husband slots. 

\(\Delta\) refers to Mother Slots; the others, \(\triangleleft, \rightarrow, \leftarrow\), all refer to Husband slots.
4. EXAMPLE - Creating & Using a Type of Slot

RLL-1 derives a considerable amount of power from one basic idea: that each component of a representation system must itself be explicitly represented. In RLL-1's case, these pieces are encoded as units, and can be examined or modified as easily as any other bit of data in the Knowledge Base. The following example illustrates this point, showing how the user can gain increasingly more control over RLL-1's actions, by exploiting his ability first to see, and then to alter the definition of a slot.

4.1. Definition of "Type of Slot"

An associative network consists of units, which are linked to one another, and to arbitrary values, by labelled arcs, called slots. Figure 1 shows an associate network whose units each represents a member of the Bach family, and whose slots are labelled either Mother or Husband. Each of these labels is, in RLL-1's vernacular, a type of slot. There are many things one can say about each type of slot. For example, the inverse of Husband is Wife; and each type of slot belongs to the class of types-of-slots. RLL-1 devotes a unit to store this information. Figure 2 shows a few such units, each of which describes a type of slot. Many of the slot-defining slots shown below are described later in this section; they are all defined in Appendix E.

<table>
<thead>
<tr>
<th>Name:</th>
<th>Husband</th>
</tr>
</thead>
<tbody>
<tr>
<td>Description:</td>
<td>The male spouse of some woman.</td>
</tr>
<tr>
<td>IS-A:</td>
<td>(AnySlot)</td>
</tr>
<tr>
<td>ToGetValue:</td>
<td>DefaultValue</td>
</tr>
<tr>
<td>Format:</td>
<td>Single-Entry</td>
</tr>
<tr>
<td>Datatype:</td>
<td>ManType</td>
</tr>
<tr>
<td>(i.e. the entry is a unit, representing a man.)</td>
<td></td>
</tr>
<tr>
<td>Inverse:</td>
<td>Wife</td>
</tr>
<tr>
<td>SubSlotOf:</td>
<td>Spouse</td>
</tr>
<tr>
<td>MakesSenseFor:</td>
<td>Any woman</td>
</tr>
<tr>
<td>MyTimeOfCreation:</td>
<td>4 July 1980, 7:23 PM</td>
</tr>
<tr>
<td>MyCreator:</td>
<td>DaveSmith</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Name:</th>
<th>IS-A</th>
</tr>
</thead>
<tbody>
<tr>
<td>Description:</td>
<td>The list of classes to which I AM-A member.</td>
</tr>
<tr>
<td>IS-A:</td>
<td>(AnySlot)</td>
</tr>
<tr>
<td>Format:</td>
<td>List-of-Entries</td>
</tr>
<tr>
<td>Datatype:</td>
<td>ClassType</td>
</tr>
<tr>
<td>(That is, each entry is a unit representing a class of objects.)</td>
<td></td>
</tr>
<tr>
<td>Inverse:</td>
<td>Examples</td>
</tr>
<tr>
<td>UsedByInheritance:</td>
<td>Inherit-Along-IS-A-Links</td>
</tr>
<tr>
<td>MakesSenseFor:</td>
<td>Any thing</td>
</tr>
<tr>
<td>MyTimeOfCreation:</td>
<td>1 April 1979, 12:01 AM</td>
</tr>
<tr>
<td>MyCreator:</td>
<td>DougLenat</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Name:</th>
<th>Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>Description:</td>
<td>The cost of this object.</td>
</tr>
<tr>
<td>IS-A:</td>
<td>(AnySlot)</td>
</tr>
<tr>
<td>Format:</td>
<td>Single-Entry</td>
</tr>
<tr>
<td>Datatype:</td>
<td>NonNegativeRealNumberType</td>
</tr>
<tr>
<td>(i.e. the entry is a non-negative real number.)</td>
<td></td>
</tr>
<tr>
<td>DefinedAs:</td>
<td>Sum of CostA, CostB, and CostC.</td>
</tr>
<tr>
<td>MakesSenseFor:</td>
<td>Any sellable item.</td>
</tr>
<tr>
<td>MyTimeOfCreation:</td>
<td>24 April 1979, 5:27 AM</td>
</tr>
<tr>
<td>MyCreator:</td>
<td>RussGreiner</td>
</tr>
</tbody>
</table>

Figure #2 - Units devoted to the "Husband", "IS-A", and "Cost" type of slots. Many other slots are appropriate for these units; each of their values will be deduced automatically, if requested.
4.2 Retrieving a Slot's Value (Husband)

Many representational systems have the view that all slots should be treated in exactly the same way. Others partition their slots (or more generally, relations) into a fixed collection of a few basic, disjoint types; and process all members of a given partition identically. As [Brachman79] points out, these approaches pose a major limitation, and lead to problems galore. RLL-1 is built on the basic premise that each type of slot should know how it should be handled. For convenience, RLL-1 includes a host of general algorithms, which a slot will inherit by default. These functions are all most of the slots will ever need. On the other hand, the user is not constrained to only these, as he is free to define his own procedures which dictate how to handle a particular slot. Later in this example we will see that RLL-1 provides a variety of tools to simplify this task. Given this overwriting ability, these default procedures are not at the expense of those slots which require a more complex process, or those which can utilize a less general but more efficient set of algorithms.

From this point of view, RLL-1's Knowledge Base functions (in particular, GetValue and PutValue) can be very simple: To retrieve the value of U:S, GetValue simply goes to the S unit -- that is, the unit devoted to storing information about the S type of slot -- and extracts the function stored on S:ToGetValue. GetValue then applies this function to U and S; and returns the value that function returned. The real "smarts", notice, is distributed to the slots, not the GetValue function.

For example, we may ask who was the Husband of Johanna Maria Dannemana (represented by the unit, JMD). The function call (GetValue 'JMD Husband) would first retrieve the function stored on Husband:ToGetValue, and then apply that function to JMD and Husband. In the default case, Husband:ToGetValue would be DefaultGetValue. Calling DefaultGetValue on JMD looks for a value physically stored on the JMD unit, labeled Husband. Examining Figure 1, we see there is such a value, KPE (the unit which represents Karl Philipp Emmanuel Bach). Popping up, DefaultGetValue returns KPE to GetValue, which in turn simply returns this value.

We might imagine another form for this Knowledge Base, in which only marriage certificates are stored, instead of direct links from wife to husband. In this case, Husband:ToGetValue should be filled with the algorithm, FindHusbandUsingMarriageLicenses, which would scan the set of marriage certificates, looking for one whose "Wife" entry matches its first argument (here JMD). Finding such a form, FindHusbandUsingMarriageLicenses would return the value which filled the "Husband" parameter, which GetValue would then return. Note that we never wasted our time looking on the JMD unit for a slot we knew would not be there. Constructing such a definition for Husband is fairly easy, using the tools RLL-1 provides. The following example shows a more complicated way of finding a slot's value, using the slot's high level definition.

4.3 Creating a New Type of Slot (Father)

For obvious reasons, most slots are quite similar to one another. We exploited this regularity in developing a high level "slot-defining" language, by which a new slot can be defined precisely and succinctly in a single declarative statement.

Suppose we want to define a Father type of slot, in the sexist geneological knowledge base shown in Figure 1, which contains only the primitive slots Mother and Husband. Creating this new Father type of slot is easy using this language: we create a new unit called Father, and fill its HighLevelDefn slot with the value (Composition Husband Mother), "Composition" is the name of a unit in our initial system, representing a "slot-combiner" which knows how to compose two slots.

---

5 The expression, U:S, is a short hand for (GetValue 'U S), which is, by definition, the value of the S slot of the unit U. The starred S follows our convention of italicizing the name of a slot, when it is acting qua slot. This is distinguished from the unitalicized S, which refers to the unit which represents the S type of slot.

6 This recursive call to GetValue could easily lead to an infinite loop. Appendix A.1 shows how RLL-1 sidesteps this problem.
regarding each slot as a function from one unit to another.\textsuperscript{10} We also fill the new unit's \textit{IS-A} slot, to derive the unit shown in Figure 3.

<table>
<thead>
<tr>
<th>Name: Father</th>
<th>IS-A: (AnySlot)</th>
<th>Description: This unit represents K. P. E. Bach.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name: Mother</td>
<td>IS-A: (AnyMan, ...)</td>
<td>Description: This unit represents Maria Barbara.</td>
</tr>
<tr>
<td>Description: This unit represents Johann Sebastian Bach.</td>
<td>Description: This unit represents Maria Barbara.</td>
<td>Mother: MBB</td>
</tr>
<tr>
<td>Husband: JSB</td>
<td>Description: This unit represents Johann Sebastian Bach.</td>
<td>Name: JSB</td>
</tr>
</tbody>
</table>

Figure 3 - Slots filled in by hand when creating the unit devoted to the "Father" slot. Several other slots (e.g., the syntactic slots \textit{MyCreator}, \textit{MyTimeOfCreation}) are filled in automatically at this time.

Suppose the user now wishes to determine Karl Philipp Emanuel's \textit{Father}, i.e., the value of \textit{KPE:Father}. The initial knowledge base, shown in Figure 4, (a magnified portion of Figure 1) has units to represent various members of Bach's family.

In this case, \texttt{GetValue} asks the \textit{Father} unit for a function to use to extract the value of \textit{KPE:Father}. Based on what it was given, shown in Figure 3, the \textit{Father} unit inherits the default mechanism, \texttt{DefaultGetValue}, for determining a unit's \textit{Father}.\textsuperscript{11} A simplified version of this algorithm is shown in Algorithm 1 below.

1. See if there is a value physically stored on the \textit{Father} slot of \textit{KPE}.
   If so, return that value.
   Otherwise, continue:
2. Get the function stored on \textit{Father:ToCompute}.
3. Apply the function to the unit \textit{KPE}.
   {This should return the value \textit{JSB}.
4. Store this computed value on the \textit{Father} slot of the unit \textit{KPE}.
5. Return this computed value.

Algorithm 1. Deducing the value of \textit{KPE}'s \textit{Father}.

As there is no value to be found on \textit{KPE}'s \textit{Father} slot, the test on Step 1 will fail. \texttt{DefaultGetValue} now must get the value of \textit{Father:ToCompute}, which should be a piece of code, capable of calculating the value of a unit's \textit{Father}. Getting this value uses essentially the same algorithm as the one shown above, \textit{mutatis mutandis}. That is, \texttt{GetValue} recurs. (See Algorithm 2.\textsuperscript{12})

\textsuperscript{10} That is, we regard \textit{Mother: (Unit)\rightarrow (Unit)} and \textit{Husband: (Unit)\rightarrow (Unit)} Thus "Mother Goth\textit{Wilhelm} = Dorothea Elisabeth" and "Husband Dorothea Elisabeth = Wilhelm Friedmann" are both well defined phrases. \textit{Father}, as the composition of these two functions, will therefore also map units into units.

\textsuperscript{11} It is important to realize that this procedure is only one of a host of retrieval mechanisms the initial RLL-1 systems offers; and that it is NOT hardwired into RLL-1. Appendix A.1 explains how RLL-1 can support this versatility and generality without sacrificing efficiency; and Appendix B.1 gives an example in which a non-default accessing function was used.

\textsuperscript{12} This also relies on the fact that, once again, the value of the slot's \textit{ToGetValume}, (here \textit{ToCompute:ToGetValue}) is \texttt{DefaultGetValue}.
Once again the test in Step 1 will fail, as this is the first time the Father of anything has been requested. So onto Step 2 of Algorithm 2. For this, we need the value of ToCompute:ToCompute. Algorithm 3 shows the now-familiar procedure followed, in a more general form.

![Algorithm #3. Deducing the value of U's S.](image)

Binding u to the unit "ToCompute", and s to the slot "ToCompute" in Algorithm 3, we see RLL-1 then asks for the value physically stored on ToCompute:ToCompute. Fortunately there is a value there (else an infinitely recurring loop would ensue, as the value of ToCompute:ToCompute is required to deduce the value of X:ToCompute for any X, and in particular for X=ToCompute.) The ToCompute unit is shown in Figure 4.

![Figure 4 - The infamous "ToCompute" unit.](image)

We are now at Step 3, in Algorithm 2. This ToCompute:ToCompute function is applied to KPE. Omitting the irrelevant details, this uses the value of Father:HighLevelDefn, which, recall, we specified as "(Composition Husband Mother)" at the start of this example. The function HighLevelExpand then parses this high level specification into a LISP function -- one which takes a unit X, and returns the Husband of the Mother of X. That is, this derived function will follow X's Mother slot to another unit, and then return the value of that unit's Husband slot. This function is then cached (that is, "physically stored"\(^\text{13}\)) on Father's ToCompute slot, as per Step 4 of Algorithm 2, and then returned as the result of Father:ToCompute, as Step 5 instructs. At this point, the Father unit looks like:

![Figure 5 - Later form of the Father unit, after the value of the ToCompute slot has been calculated and stored.](image)

\(^{13}\) This caching facility is an essential feature of RLL-1. It is discussed at length in Appendix A.2.
We now pop back to Step 3 of Algorithm 1. Applying this Father:ToCompute function to KPE, RLL-1 follows KPE's Mother link to MBB, and there finds the value of MBB's Husband, JSB. This value, JSB, is then cached on KPE:Father, and finally returned as the result of this call.

4.4 Other Facts Stored on the Type of Slot Unit

RLL-1 was able to automatically determine the value of every additional slot shown in Figure 6 from the HighLevelDefn of Father; calculating each (like ToCompute) only as it was needed. Much of the information RLL-1 needs is stored on the Composition unit, which "knows" about the composition of two functions. For example, as the domain of \( f_1 \circ f_2 \) is simply the domain of the function \( f_2 \), the domain of Father must be domain of Mother; i.e. any person. A similar analysis of the range of Father indicates that it is simply the range of Husband; i.e. any male person.

Such facts, which describe relevant attributes of this Father slot, are stored on the Father unit itself. Hence the datatype expected of each value stored in a Father slot is indicated by the value of Father:Datatype, which Figure 6 shows to be ManType. As Section 5.3 will reiterate, this is, in fact, a pointer to the "ManType" unit, which in turn holds the declarative and procedural information associated with (RLL-1's view of) men in general. Similarly the fact that each Father entry must be a single entry (as opposed to a matrix of entries or an unordered set of entries) is encoded by the "Single-Entry" value stored on Father:Format. The various accessing functions use such information when dealing with Father's values.

```
| Name:     | Father                          |
| Is-A:     | (AnySlot)                       |
| HighLevelDefn: | (Composition Husband Mother)    |
| Description: | Value is Husband slot of Mother slot of this unit. |
| Format:   | Single-Entry                    |
| Datatype: | ManType                         |
| MakesSenseFor: | AnyPerson                      |
| DefinedInTermsOf: | (Husband Mother)               |
| DefinedUsing: | Composition                    |
| VerifyValue: | FVV                            |
| ToCompute: | \((x) \ (\text{GetValue} (\text{GetValue} x \text{Mother}) \text{Husband})\) |
```

Figure #6 - Later form of the Father unit, showing several slots filled in automatically.

For example, before storing a value of a unit's Father slot, the standard putting function will first check to be sure the value is of the correct type. That verifying function is stored on Father:VerifyType, and had been determined using the Format and Datatype of Father. That is, VerifyType:ToCompute is a function which takes a unit (here Father) and returns a value, FVV, which is then stored on Father:VerifyValue. (This FVV function takes a value, X, and returns T if X is an acceptable value to fill some unit's Father slot -- that is, if X is a single value, which is a value which refers to a man.) FVV was constructed using the values of Father:Datatype=ManType and Father:Format=Single-Entry.

Since Father is defined in terms of both Mother and Husband, using the slot-combiner Composition, the Father unit states that a value stored on KPE:Father must be invalidated if we ever change the value for KPE's Mother or MBB's Husband, or the definition (that is, the value of the ToCompute slot) of Father. Recurring one level, as this Father:ToCompute function was defined using the definition of Composition, if this Composition:ToCompute\(^\text{4} \) value is ever altered, Father:ToCompute must be updated correspondingly.

Appendix A.2 show that one easy invalidation technique is simply to erase the cached value. Section 5.2 elaborates this process, which is part of an idea called "Cognitive Economy".

\(^{14}\) Each unit which represents a function has a ToCompute slot, which holds the actual LISP function it encodes. Associating such a ToCompute slot with each slot reflects our view that each slot is a function, whose argument happens to be a unit and whose computed value may be cached away. (See Footnote 10.)
Notice how a user can "extend his representation" by enlarging his vocabulary of new slots. A similar, though more extravagant example would be to define FavoriteAunt as

\[
\text{FavoriteAunt} = \text{SingleMost} (\text{Unioning} (\text{Composition} \text{Sisters Parents}), \text{Composition} \text{Wife Brothers Parents}))
\]

"Unioning" and "SingleMost" are two other slot combiners which come in the initial RLL-1 system; their definitions and ranges can be inferred from this example. As before, this information given by the high level definition above is sufficient (1) to compute any unit's FavoriteAunt, (2) to determine whether a value proposed to fill this slot in fact qualifies or not, (3) to determine for which units the slot FavoriteAunt is meaningful and (4) to perform the necessary work to preserve consistency throughout the network whenever some slot's value is altered.

4.5: Modifying an Existing Type of Slot

Any frame-based system must allow the user to create a new type of slot. (This will require but a trivial declaration in those systems which treat all slots identically.) Few provide the user with a language to facilitate this task, or the tools required to specify such slots flexibility and concisely. RLL-1's capabilities in this capacity were demonstrated in the previous subsection. In even fewer systems is the user then able to modify such definitions. The following example shows how easy such alterations are in RLL-1.

Imagine someone wished to apply our geneological knowledge base to some polyandrous culture, in which a woman might have more than one husband. In this case, the obvious thing to do would be respectify the Husband type of slot, to allow each woman a set of husbands. This change would be as easy to make in RLL-1 as it was to describe here, and could be done as naturally. The user would simply refill the Format slot of the Husband unit (shown in Figure 2a,) with the value "Set-Of-Entries".

RLL-1 then uses its knowledge of the format units "Single-Entry" and "Set-Of-Entries", (in particular their differences) to do all the rest of the work. First, as a set consisting of one entry is stored differently than a single entry, RLL-1 would first map along the affected units, changing the value of each Husband's slot appropriately -- so MBB:Husband would be changed from "JSB" to "(JSB)". (Note the domain information about Husband, stored here in Husband:MakesSenseFor, considerably shortens this scan; it tells RLL-1 the only units which might be affected are those which represent a woman.) Next, RLL-1 would examine those other slot types which might be affected by this change. Here it would note that, based on Father's current definition, Father:Format must be changed to correspond to Husband:Format. This modification would have similar ramifications, cause other units to be altered, and so forth. For example, the value of KPE:Father would have to be listified to "(JSB)".

Notice these were the sort of dull, mundane updates a system programmer/maintainer would usually be called upon to perform. As [Winograd] points out, these are also the most standard sorts of modifications made; and it is a real loss to burden humans with so straightforward a task.

With this same ease the user could replace essentially any other slot of the Father unit, and RLL-1 would propagate the changes this evokes. In fact, this facility provides the user with another way of defining a new slot S (besides using only the High/ev/Depth); copy an existing slot, (choosing one which is functionally similar to S) and respectify just those one or two slots of this new unit whose values are inappropriate to S. Hence we could have built a HusbandSet type of slot in this incremental-change fashion: first create a new HusbandSet unit, filled with the slot/value pairs found in the unit Husband. Then change the value of HusbandSet:Format to be Set-of-Entries. This new HusbandSet slot would then perform in the manner we attributed to the changed Husband slot above; and leave the original Husband slot to do the singular thing its name implies.

A third, closely-related method of creating a new type of slot is to build the slot up in a piece-meal fashion, by filling in subordinate slots (e.g. Format, Datatype, MakesSenseFor, ToCompute) one by one. RLL-1 would then integrate these into the desired, working version, which the user may then use.
The purpose of this extended example was to demonstrate the sorts of things one can do using an RLL. In particular, it should drive home the advantages of stating facts explicitly -- even those which pertain to processes, such as the inferencing required to retrieve a slot's value. The next section will describe this particular RLL-1 in more detail.
This section provides a general description of the RLL-1 system, to two levels of abstraction. The first half describes our overall philosophy of what an RLL should be. The second part is more detailed, enumerating the particular conventions embodies in the initial RLL-1 system. These attributes may be viewed as the default positions selected for certain organ stops.

The heart of RLL-1 is its philosophy: keep every component of the system visible and modifiable. Towards this end, we encoded essentially every representational piece as a unit, so the user may use the same formalisms to examine and manipulate these that he uses when dealing with his other data. The next objective is to construct each of these pieces to be as general as possible. This is manifest in two ways: First we designed the Garden of Eden system to be sufficiently versatile that we imagine many users will be able to use it, unaltered, for their tasks. Second, the essential functions are extremely simple. The unavoidable complexities has, instead, been distributed among the appropriate units, which can be pursued and easily changed. For example, all of the accessing/writing functions (e.g. GetValue and PutValue) are trivial -- each simply asks one of its arguments for the value of a particular one of its slots, and applies the function stored there to the arguments passed to the original function. (Appendix A.1 shows this in detail.)

Of course, functions themselves might warrant being stored as units -- certainly the accessing functions qualify, as they each represent a legitimate piece of the representation. The problem is how to trick LISP to permit a user to "apply" a unit to some arguments, as if that unit was a function. At a linguistic level, one might, for example, expect Father(KPE) to return JSB, the value of KPE:Father. A hack to LISP's evaluator, described in the beginning of Appendix F, achieves this essential effect. The user can still treat these functions as he treats any other units -- only now he is creating or modifying executable processes; and doing so using the same operators which worked on arbitrary units.

The final "global" consideration is how easy such modifying operations are. RLL-1 provides a host of powerful tools, designed to perform the sort of manipulations most users will want to make. An example is the slot defining language illustrated in Section 4. Such high level specifications should be found throughout a good RLL, to enable the user to perform standard manipulations on the standard types of units. Like the other suggestions above, this facility is undefined UNLESS the parts of the representation themselves are represented, and changable -- a slot defining language is meaningless unless one has slots to describe; and impotent unless there are meaningful, usable facts to assert about what a type of slot means.

There is still considerable flexibility within the philosophical guidelines outlined above. The following characteristics describe the initial state of the current RLL-1 system, when all "organ stops" are in their default positions. Each user will doubtless settle upon his individual settings, more suited to the representation environment he wishes to be in while constructing his application program.

5.1: Units and Slots: RLL-1's initial language is basically a frame-structured system, in which information is encapsulated into units, where each unit is composed of a list of slots, each with its designated value. This bias is largely for historical reasons, as RLL-1 grew out of the Units package [Stempel].

15 In fact, for pedagogic reasons we may describe this system to a user without mentioning its ability to metamorphosize. Only if she found this system limiting -- at the point she starts to grumble at RLL-1's inflexibility, or worse yet, begins to mold her data to conform to RLL-1's expectations rather than vice versa -- would she be told she could have constructed a new language of her own specifications at any time (...) all she had to do was click her heels together three times ...
5.2: Cognitive economy: There are many time/space tradeoffs involved with any complex process. Caching a computed value -- for example, the Father of KPE -- is only worthwhile if this value will be used again; otherwise it is a waste of (at least) two list cells. Similarly, it is often expedient to generate and retain multiple representations of some fact, using each for a particular purpose -- for example, using the source code for reasoning about a function, but actually executing the corresponding compiled version.

RLL-I's basic approach is to begin with a minimal collection of facts, sufficient to compute the other facts -- note this bag must include not only (at least) one version of each necessary fact, but also the mechanisms required to deduce the other values. If a new form of the data is needed, it will be computed automatically. In fact, the user need never know whether that value was laboriously calculated, or merely retrieved. RLL-I then explicitly considers whether or not to store this new information; and if so, where it should be stored. The retrieval algorithm must cooperate, by examining the storage location before considering calculating this value; in this way storing a value insures that it will not be recomputed. (Note that Algorithm 3 in Section 4.3 demonstrates this: it first checks if there is a value physically stored in U:S; and attempts to (re)compute this value only if that location is empty.)

For this procedure to work correctly many other parts must first be in place. Evaluating this space/time tradeoff requires a nontrivial process. RLL-I delegates this computation to the function which fills the slot's ToCache slot. There are currently several pre-defined functions from which the user may chose, all shown in Appendix F. (Of course, the user always has the option of constructing his own.) Another concern is how to maintain consistency among these redundant pieces of information. Appendix A.2 shows one standard way of invalidating a no-longer-correct value; and Appendix B.2 shows an additional use for this facility. The final major problem is constructing the minimal set of facts, referred to above. This Garden of Eden must have enough of the essential functions to calculate all the others; and include all the parts necessary to modify itself. For example, Section 4.3 shows why it must include ToCompute:ToCompute (at least if the standard default accessing functions are used).

As with other RLL-I processes, the user can see just what will happen, and how. Storing the caching information on the ToCache slot of type of slot is an example of this explicitness. There are also hooks now in place for upgrading this into a more complex processes, such as the one outlined in Appendix B.2, which have not yet been used. This whole approach is labelled Cognitive Economy in [Lenat, Hayes-Roth & Klahr], and is similar the memo function idea discussed in [Michie].

5.3: Syntactic versus Semantic slots: The usual definition of a representational scheme is a structure, usually symbolic, which in some way reflects what actually happens in the real world. The nodes in most semantic nets, therefore, are intended to represent some real world entity -- i.e. something which lies strictly outside this (in-the-computer) simulation model. For efficiency reasons, a slightly augmented interpretation of "model" is required to understand the units used in RLL-I: In addition to facts about that entity outside this system, each unit is expected to house various bits of information which pertain to this particular unit, as a unit within this system. Hence, the usual definition of "the world out-there" has been extended to include our within-the-computer structure. This is a hack, and is, epistemologically, quite troublesome.

Essentially, RLL-I simulates a meta-level KPEUnit -- one which represents the unit "KPE": the way "KPE" represents a unit in the real world. This logically distinct unit is physically merged into the KPE unit. This leads to a number of complicated situations, which we have worked through, one by one. One non-trivial issue revolves around inheritance. For example, KPE should inherit values for many slots from TypicalMusician, such as Instruments, Birthdate, and Size; but not from slots which refer to TypicalMusician qua data structure -- e.g. slots such as NumberOfFilledInSlots and

---

16 An appropriate metaphor is the refrigerator light -- which is always on, every time you look. This laid-back "if-needed" attitude is not the only possible approach -- one could, for example, compute such values at first opportunity, in a "when-filled" manner.
DateCreated: RLL-1 treats these two classes of slots differently, e.g. when initializing a new unit. Appendix C justifies why we merged these two logically distinct units into one physical one, and then steps through the creation and initialization of a unit, to demonstrate how these two kinds of slots are handled separately.

5.4: Onion field of sub-languages: RLL-1 contains a collection of features (e.g., automatically adding inverse links, and unit initialization routines) which can be individually enabled or disabled, rather than a strict linear sequence of higher and higher level languages. The partially ordered set this forms is more like an onion field than the standard "skins of an onion" layering. We have barely begun the shore of designing and implementing additional facilities, and incorporating each at the appropriate layer of the appropriate onion. This is, obviously, an on going task. RLL-1 does provide the facility for physically storing a group of units into a clump, called a Knowledge Base or KB. The user can then load in just those KBs which are necessary for his task; these will collectively define his current system and language.

5.5: Economy via Appropriate Placement: Each fact is placed on the unit (or set of units) which are as general and abstract as possible. Frequently we create new units, or even new classes of units, to house this general information. In the long run, this policy of storing information towards the trunk of the tree, rather than on the leaves, will reduce the need for duplication of information.

Standard property inheritance is one example of this idea. Diagram #1 in Appendix E shows some of the classes which RLL-1 contains. There are classes which represent, among other things, classes, prototypes, slots, datatypes, formats, inheritances, and functions. Each of these class units represents the set of its members, e.g. AnySlot which represents the set of all slots. The typical example of each class, e.g. TypicalSlot, can be used to store facts common to all members of this class. So, for example, the fact that a slot is a function which maps from a unit onto a value is stored on TypicalSlot, and can be inherited, as needed, by any unit which descends from AnySlot -- that is, by any individual slot, such as Father.

Examining Figure 6, shown in Section 4.4, we see that many of the values stored on the Father unit are actually pointers to other units. For example, Father:Datatype points to the unit ManType, which codifies RLL-1's view of men; and Father:Format leads to the unit Single-Entry, which "knows" how to deal with those slots whose value is a singleton. Basically the chunks of information stored on this Single-Entry (respectively ManType) unit collectively help define what it means to be a single atomic value (respectively a man).

These format and datatype units represent the "logical placement" for these morsels of information. It is correct to assert that RLL-1 views List-of-Entries as representing "Listiness", in the way "ManType" encapsulates facts about "manishness". We saw in Section 4.4 that VerifyType was able to use this information to create a function for type checking, and Section 4.5 demonstrated that RLL-1 could reason about such singleton versus set values. RLL-1 can also exploit information derived from the organization of these units -- for example, the fact that ManType is a SubDT of both PersonType and MaleType helps to specify "manishness".

Another slot, such as Cost, can also refer to this format Single-Entry; as might a LISP function such as car. "ManType" is also general enough to be widely used: as the datatype of Uncle or MostContemptibleChauvinistPig, or as part of the domain specification for the slot Wife or Sired (which maps a man to his offspring). This idea of appropriate placement of facts allows RLL-1 to store pertinent facts in this one well-defined location, rather than smearing such data throughout the Knowledge Base.

These format units are designed to house everything we may wish to say about that format; and so may be used for many other purposes, in addition to those mentioned above. A fact on the Set-of-Entries unit indicates that a set does not contain repeated elements. This fact is, of course, relevant for type checking. It is also used to construct the function which adds on a new element to an existing set -- instructing that function to add on an element only if it was not already there. Note that List-of-Entries has no such caveat, as values may be repeated in a list. We will see in Appendix D.4 that the empty set is stored as 

Aremes: and this fact is housed, declaratively, in this
same Set-of-Entries unit.

As List-of-Entries is charged with storing all the facts which relate to (RLL-1's definition of) lists, modifying this List-of-Entries unit is sufficient to correct or augment any of the format-related characteristics (such as value verification) of any slot whose format is a list.

Another example of this idea is the use of appropriate conceptual units:

5.6: Clarity of Conceptual Units: RLL-1 can distinguish (i.e. it has devoted a separate unit to each of) the following concepts: TheSetOfAllElephants, (whose associated properties describe this as a set -- such as #OfMembers or SubCategories), TypicalElephant, (on which we might store ExpectedTuskLength or DefaultColor slots), ElephantSpecies, (which EvolvedAsASpecies some 60 million years ago and is CloselyRelatedTo the HippopotamusSpecies,) ElephantConcept, (which QualifiesAsALoadingHorse and a TusksPerHead.) ArchetypicalElephant (which represents an elephant, in the real world, which best exemplifies the notion of "Elephant-ness"). It is important for RLL-1 to be able to represent them distinctly, yet still record the relations among them.

On the other hand, to facilitate interactions with a human user, RLL-1 can accept a vague term (here, "Elephant") from the user or from another unit, and use context to automatically refine it into the appropriate more precise term. While this appears a purely linguistic feature, it is vital for the representational system to allow this flexibility as well. For example, a term which is regarded as precise today may be regarded as a vague catchall tomorrow; and RLL-1 should have the facilities to perform such a modification easily.

RLL-1 readily accommodates distinct representations; but has not yet addressed the linguistic problem of automatic disambiguation; nor built in tools for "expanding" a unit into its more precise senses.

6. SPECIFICATIONS FOR ANY REPRESENTATION LANGUAGE

The following are some of the core constraints around which this proto-rll, RLL-1 was designed. One can issue commands to RLL-1 which effectively "turn off" some of these features, but in that case the user is left with an inflexible system we would no longer call a representation language. The first five points are fairly rigid guidelines for the overall rll system, while the last two are vague wants, pertaining only to the initial system. Further details may be found in [Genesereth & Lenat]. The "current status" part which concludes each sub-section should indicate whether RLL-1's satisfies this constraint or not. (Recall the meaning of the "-1" in RLL-1's name.)

6.1: Self-description: Every part of an rll system must be visible; every one of its components, from individual slots and datatypes, to modes of inheritance, and even the data-accessing functions (e.g. GetValue and PutValue) themselves, must be explicitly represented within the system. To avoid an infinite descent, (of having to describe the tools used to describe the tools which are used ...) it is essential that these description use the same formalisms that the representation uses for its other data -- in RLL-1's case, Units and Slots. To be self-descriptive, the rll must be described in terms the system itself can understand and reason about. A system may be described using LISP code, for example, only if the system has already axiomitized facts about LISP.

Although this LISP code might satisfy the self-descriptive criteria, higher level constructs are required for the user to readily be able to reason about the behaviour or characteristics of some part of the system. As the process which translates from LISP to more natural form is a part of the overall self-described system, it will be well-described; and hence the higher level which resulted from its execution will also be well defined. In this manner we can bootstrap to successively higher level, more perspicuous, primitives.

---

17 These "axioms" need not be formal predicate calculus statements. We do insist, however, that this specification be versatile and have an unambiguous interpretation.
This approach has several intrinsic virtues. First, it guarantees that even complex operations, (such as the creation of a new Unit, or the determination of some Slot's value by default) are transparent; and does so without affecting the range or versatility of permissible techniques. This visibility also provides the user with the opportunity to modify the parts -- a task not possible if these pieces can only be viewed as black boxes. This requirement is discussed in the next subsection.

Current status: While we have encoded essentially all of RLL-1 in its own formalism -- i.e. as units -- we do not feel this system is truly self-descriptive. One problem is that we (and hence it) do not have the vocabulary to describe what RLL-1 does or how it really represents a fact, in terms of useful well-defined primitives. While RLL-1 uses slots, for example, it does not understand what they are; certainly not to the level that it could reconfigure itself into a proposition-based system unassisted. In addition to the many things which are simply not described, there are many places where a description has been begun, but bottomed-out with primitives which were not themselves adequately defined. An example of this is RLL-1's attempt to specify functions: this stops at the level LISP code, which RLL-1 is not yet able to reason about. To be self-descriptive, RLL-1 has either to develop a set of well-defined primitives above this LISP level, or gain competency in reasoning about LISP itself. Either of these tasks qualifies as a major research effort.

6.2: All parts are modifiable: These parts must not only be visible, they should be alterable as well. As these parts really represent the ril's operations, performing such changes will actually change the ril system itself.

The question of how the user actually performs these modifications, while secondary, is an important concern. The guarantee given in the last subsection -- that these parts are encoded in the same formalism used for any other bit of data -- means these alteration can be made using the same tools which serve to alter the user's domain knowledge. It does not, however, mean such adaptations will be easy to make. For example, much of RLL-1 is written in LISP code. While this is visible, it is not at a level which facilitates easy modifications. RLL-1's approach has been to provide the user with a collection of powerful tools, designed to perform certain standard manipulations. Section 4 demonstrated one type of tool: high-level languages. The user can make a small change to this level of description, and RLL-1 will automatically propagate this change down to the level of LISP code. One can thereby trivially change the behavior of that part of RLL-1.

Current status: The requirement of self-description is obviously a prerequisite for this attribute -- how can you change any part which hasn't been described? Similarly there must be a connection between the unit and what it proports to represent -- otherwise a change to that description unit will have no effect on the overall system. As RLL-1 contains many facilities which are not fully and accurately described, for both of these reasons, it is not yet self-modifiable. There are many parts, including slots, processes, functions, formats, datatypes and inheritance modes, which have been authentically encoded as units and can be readily changed. Much additional effort is required to formalize many of the other components.

6.3: Epistemological Adequacy: The requirement is simply that the ril be Turing equivalent -- i.e. that its processing power not have any obvious limitations. This guarantees that the representation language itself will not have any intrinsic restrictions on what a user can represent. This specification is automatically satisfied by any system which contains a language like LISP as a substructure.

Current status: Trivially achieved, as RLL-1 can always falls back on its LISP underpinnings.

Notice this epistemological issue does not address the question of how the data is actually encoded internally, or how it will appear to the user. The next two sub-sections discuss how the ril and the user, respectively, view the information.

\[\text{We may use rule-like chunks of procedure knowledge as the primitive level. Each of these would be encoded as a unit, and contain both declarative and procedural information, as well as a precise "axiomatic" description. Appendix B.2 begins to illustrate these entities.}\]
6.4: Linguistic Adaptability: Not only must the rll system be capable of expressing any statement, it should be able to do so in any manner the user wishes. That is, if the user wishes to encode his knowledge as units, the rll should not force him to use production rules. The actual language in which the user describes the task should be based on (i.e., user's perception of) a natural decomposition of the problem, unhampered by the rll. The reasons for such user-interface considerations will be readily apparent to anyone who has tried to write a LISP program in Fortran. Although LISP and Fortran are epistemologically equivalent, each handles the data in its own manner. Things which are trivial to express in LISP may be next to impossible to encode in Fortran; and (conceivably) vice versa.

In an attempt to sidestep this limitation, many systems come equipped with a single "universal" dialect; while others are built conversant in a host of languages. In either case, the resultant system will, (by definition if not by construction) be adequate for every task the designer could envision. There will, however, always be other types of tasks, which the designer had not considered, and hence not designed for. There is also the danger this particular designer may have some strong bias, not shared by the future user. Besides, such hard-and-fast built-in specifications are contrary to rll's basic "open-ended and extensible" philosophy.

This requirement is simple: permit the user to define whatever language he wishes. The rll will, of course, provide the tools and mechanisms for such creations, as well as a library of known interfaces which the user may use or modify as he sees fit. In effect, we insist that the rll's interfaces be as transparent and modifiable as the rest of its operations. (Hence the requirements made in subsections 6.1 and 6.2 essentially subsume this argument. Note further that the requirement stated in 6.3 insures this will be possible, as any universal Turing machine can be programmed to simulate any language.) In this way, the representation language can adapt to the user's idiosyncrasies, rather than vice versa.

Of course, the user must be able to describe such a language to the rll, and this specification must be in terms this rll can understand. The obvious medium for this communication is rll's initial language. Here, indeed, the user must conform to some externally imposed set of conventions, but only long enough to deliver his medium-changing command.

Current status: RLL-1 is currently mono-lingual -- it can only receive and convey information in terms of units and slots. Although the user is free to code up his own vernacular, RLL-1 does not yet have the tools and basic high-level constructs which would make this task easy. This requirement has been postponed until the problems and issues relating to representational adaptability (next subsection) have been seriously addressed and answered -- we currently believe this linguistic malleability should come for almost nothing, as a side effect of making RLL-1 representationally universal. Arguments suggesting why this should follow appear below.

6.5: Representational Adaptability: The last section insisted that rll be able to present the information in whatever form the user desires. This need not correspond to how rll, internally, sees the data. This is comparable to finding a person who speaks in English, but thinks in German. An rll could similarly fool the user: while it "spoke" in his language, it might actually process the data using some other representation.

Once again, this concern asks who should decide how the rll should process. Clearly the rll should be as flexible as possible. For example, it should be able to "think" in terms of wffs, production rules, or conceptual dependencies, as the situation warrants. As before, many existing systems provide one "universal" representational schema, or a fixed collection of diverse structures. This approach seems short-sighted, for reasons isomorphic to those presented in the last subsection.

To insure that an rll be able to represent the data internally in any form the user wishes, we once again place the responsibility of its design in his hands. As before, the initial rll should include a stock of very general processes which the user can then mold into one more to his liking. Imposing any additional system-convention inherently violates this goal, and may force the user to encode his data in awkward or unnatural forms. (Of course, any user may still insist on using his own, arbitrarily kludgey representation, if he wishes. The point here is that he alone is accountable for
such messes, and not the rll system.)

Current status: RLL-1 isn't yet clever enough to deceive the user. Its single unit/slot organization is all it is currently equipped to use. As other representational strategies are designed, RLL-1 will have to be told how to translate from that representation into RLL-1's Lingua Franca. We feel this translation process will eventually have other applications: First, it should provide the framework needed to simulate one language using another. It may also provide RLL-1 with the information necessary to converse in this tongue; thereby achieving the goal posed in subsection 6.4.

6.6: Unbiased initial system: The bootstrap rll, i.e. the representation language initially handed the user, is special in several respects. First this is usually the Lingua Franca for subsequent languages -- that is, each will be defined using terms described and defined in this initial language. Secondly, it is in this language that the rll system itself is encoded. For this reason, it should be as perspective and general a language as possible. Any bias here may adversely affect any representation language created on top of this underlying representation language; the way a faulty machine instruction can only be detrimental to any program which has to code around it.

Current status: As everyone has his own criteria for cleanliness and naturalness, this is more a subjective request than a hard-and-fast requirement. This hedge notwithstanding, we do not feel RLL-1's initial system, based on units and slots, is not as unbiased as it should be. Many facts, such as those presented in Appendix B.4, are clearly awkward to represent using these conventions. Subsequent systems can avoid this deficiency by using more general n-ary propositions.

There are a host of secondary concerns, which can be used to judge among several "equally unbiased" rlls. One is efficiency -- an aspect none of these seven points directly addresses. Ease of understanding is another important point. A final related issue asks what equipment should come with the initial system. A system which includes only the minimal set of representation pieces, including just those tools which are absolutely necessary to alter the system would clearly satisfy the first six points above. The next subsection addresses our view on this question.

6.7: Codification of Representation Knowledge: Many of these subsections might seem to imply that almost anything might qualify as an rll, as the user is permitted, (indeed encouraged!) to design both its external appearance and its internal operations. One may seriously question what, if anything, distinguishes an rll from, say, LISP. In terms of power or flexibility, the answer is nothing. A more meaningful response first examines the question of what separates LISP from the machine language in which it was written. The answer here is that LISP contains many useful, well-designed tools -- such as chunking commonly used bits of code into single instructions, (e.g. MAPC) or providing an environment which does nice things like spelling correction. Similarly an rll is expected to offer the user nice, natural pieces, along with a good user interface.

It should provide taxonomies of useful representational pieces, including a tree of usable inheritances, miscellaneous matchers, powerful data retrieval functions, etc. While not really required, an rll may also provide a structured array of other useful entities, such as general world knowledge, a facility to exploit analogies, or a clear consistent model of function invocation. Subsections 6.1 and 6.2 insist these should be easy to create and modify; for this, too, the rll should provide the necessary tools. This entire collection of parts and facilities correspond to all of the stops of the organ, using the analogy defined above.

Current status: This is some of the most exciting (and endless) research we foresee; as only a smattering of representation knowledge has yet been captured.6

---

6 This is NOT to imply that the underlying representation, in which the representation language language itself is described, must conform to that language, any more than a PL/I interpreter must be written in PL/I. It is enough that it can simulate that target language. (Of course, there are big wins derived from actually bootstrapping from a subset of the target language, as LISP and RLL-1 actually do.)
7. COMPARISON WITH OTHER SYSTEMS

The previous two sections described our particular rl, and defined what we meant by the term "representation language language". Given these models, we will now try to compare our system with various other systems; or rather, describe why analogous systems are difficult to find.

By its nature, RLL-1 is able to wear several hats, depending on who is viewing it, and what task this user has in mind. It may therefore useful to first describe what RLL-1 is not. It would be a type error to confuse RLL-1 with a reasoning system, (e.g. Mycin's inference engine) or with some mechanism devoted to communicating with a user -- such as a Question/Answer-ing or Natural Language Understanding facility. It is, instead, the language on which such applications can be built. As a general tool, it should not be restricted to competency in some single task. (That is unlike PUFF [Kunz et al], whose single task is quite well defined: it was designed exclusively for diagnosis of pulmonary lung disorders.) Nor is it only applicable to some particular process or strategy, as compared to systems like EMYCIN [vanMelle], which can only utilize a backward-chaining rule system. RLL-1 is, more or less, at the level of the underpinnings for such systems -- in the relation that a BlackBoard [Emman&Lesser] model is to Hearsay systems [Lesser&Emman], or production rules are to Mycin [Shortliffe]. That is, RLL-1 is a formalism for representing the knowledge which is to be used.20

Much of this paper has pushed the idea of "RLL-1, The Ultimate Representation Language". While this RLL-1 qua rl (rl = Representation Language) viewpoint is certainly accurate, it appears of a different genre from most rl's, in that it is designed to be changeable. This immediately eliminates those languages which have a host of specific built-in features (perhaps to facilitate performing some particular task efficiently,) from meaningful comparison. This camp includes Units [Stefik], FRL [Roberts&Goldstein], OWL [Szelovitz], KLONE [Brachman], DB [Genesereth] [Mackinlay&Genesereth], NETL [Pahlman], Conceptual Dependency Graphs [Shank] and OPS5 [Forgy&McDermott].

These languages, however, are comparable with the initial RLL-1 system. This, recall, is the language each user sees when he first loads up RLL-1, (and is further specified in subsections 6.6 and 6.7). The user can then design his personal rl by molding this malleable language. This comparison must be made along several dimensions. The first concern is "power" -- what tasks can be performed in each of these systems. This question soon becomes uninteresting, as the "epistemological power" of all of these systems is clearly identical -- but so is LISP's or, for that matter, a Turing machine's. Time and space efficiency is another issue. This is rather hard to measure, as it is quite dependent on the actual task chosen on which to base the comparison. Any individual rl will, of course, perform better on some tasks than on others. For example, much of RLL-1 deals with functional specification, and so we would therefore expect this initial RLL-1 system to do well on any task which requires this type of expertise. On the other hand, any of these other systems could undoubtedly outperform this bootstrap RLL-1 on any of the tasks it was designed to perform. Altogether, RLL-1 would probably lose to any other system, counting the number of types of tasks which one can do more efficiently than the other. This is by design: RLL-1 was built for generality and flexibility, not for efficiency. This is no real loss to the overall RLL-1 system (i.e. beyond its bootstrapping neonatal self), as this system has the ability to reconfigure itself into a faster form, if necessary. So this deficiency is really not that great.

Another attribute is versatility: How many tasks, and types of task, can any of these rl's handle "cleanly"? Using that "cleanly" hedge makes this characteristic quite hard to define, at least quantitatively. One measure of elegance is the absence of extraneous and artificial "things" (e.g. units, rules, arrows) and naturalness of the remaining components. (By artificial, we mean entities not intrinsic to the problem, per se, but which were created solely to enable the system to do the "right thing".) For rather circular reasons, RLL-1 would probably force its users to endure fewer artificial units than any of these other systems. That is, RLL-1 permits the user to design his own

---

20 Notice RLL-1 is also a mechanism for building such tools; as well as one such tool.
constructs; and what user would intentionally employ an entity he personally considers unnatural or extraneous?21

Like KLONE [Brachman78], Lunar [Woods], and DII [Genesereth], we have attempted to address various epistemological issues. This is a different cut at what a representation language should do from the view taken by OWL [Szolovits et al] researchers, who, following [Quillian]'s lead, were concerned primarily with linguistic issues, or from the psychological motivation which inspired much of KRL's early work. (An excellent article [Brachman79] addresses and helps delimit these diverse perspectives.)

Many previous rls have attempted to utilize such a self-referencing ability, for example, DABA [Sandewall], KRL [Bobrow&Winograd], Omega [Hewitt et al] and [Leevesque&Mylopoulos]. These approaches seemed to lack the two important ideas shown in subsections 6.1 and 6.2: First, that the elements of a representation can and should be viewed as simple data, and handled in the same manner as domain facts. Second, that this representation itself should be modifiable.

These are two of LISP's biggest assets -- as the code it runs is in the same form as all of its other data, (as S-expressions.) LISP is able to modify its own code, and thereby bootstrap indefinitely. Efficiency might seem one reason to avoid this approach. RLL-1, however, has solved this problem using a second major idea, Cognitive Economy, described in Section 5.2. This allows the system to contain many equivalent forms of its primitives -- using more declarative versions for inspection, while executing arbitrarily-efficient ones.

There is another whole class of systems which are closely related to rls, proper. These are programs which attempt to symbolically represent actions, and be able to reason about (i.e. propagate) their effects on the rest of that formalized chunk of the world. This category includes truth maintenance systems, (e.g. TMS [Doyel&AMORD [deKleer et al]) their predecessors, (such as Strips [Fikes&Nilsson], Planner [Sussman] and Connniver [McDermott&Sussman]) program verifiers, (including [Rledsoe] and [Manna&Waldinger]) and proof checkers (like FOL [Weyrauch]). To insure that every appropriate update is performed in response to some action, the program must contain a fairly complete model of that segment of the world. It will, in addition, supply the user with a deductive mechanism for determining what action caused, (or was responsible for) some result; and rules which indicate what should happen as a premise is posted or retracted. (This deductive mechanism was not always built-in. Languages like Connniver instead provided an assortment of tools, such as a slot's if/needed facet. The user was expected to assemble these into the particular deductive strategy he wished.) This entire truth maintenance problem reduces to a rather straightforward search when the space is well-defined; the complicated part is determining that initial set-up. This operation is vastly simplified if each of the components, including the current and hypothesized states, type of responsibility pointer, etc., is made explicit and appropriately encoded -- i.e. if RLL-1's basic paradigm is followed. Indeed, much of the machinery necessary to allowing the appropriate links and perform the correct updates is included in the initial RLL-1 system.

As programs become more complex, their designers began building increasingly more powerful tools to aid in the construction and recoding processes. These tools are designed to perform the mundane translation instructions, given in a high level, natural language into code the machine can understand and execute. Simple programming languages, which converted the user's commands into machine code, were the first steps in this direction; these were followed by more sophisticated languages, which did more for the user per unit keystroke. Two types of AI systems are designed

---

21 The careful reader may point out here that this comparison was to be with the initial RLL-1 system as it is presented to the user (1), and not with RLL-1 after that user has modified it (2). The point where the system (1) becomes a system (2) is rather fuzzy, in that any it must allow the user to add data; and in RLL-1's case, this data may cause the representation itself to change.

22 METAFOIL, an extension to the FOL system, was one of the first systems which managed to encode its operations within its own formalisms. It, as such, played a major role in this RLL-1 development, by providing a proof-by-existence that such a system was indeed possible.
to go one level further in providing the coder with a more abstract language for describing a task: Automatic Programming (e.g., [Green et al.]) and Expert System Building Systems or ESBSs (à la AGE [Nii&Aiello], EMYCIN [vanMelle], and EXPERT [Weiss&Kulikowski]).

Although RLL-1 has been used to build an proto-expert system (see [Hayes-Roth, Waterman & Lenat]), it is still difficult to compare it with these other systems. The major difference is RLL-1's self-modifiability. While all of these systems can assuredly construct one type of system, cleanly and efficiently, none of the other systems were designed to modify itself.

Basically, these tools appear to overlook the fact that they themselves are rather complex programs, which may be exceedingly difficult to modify without assistance. Consider what would be required to change the specification of the target language, even ever so slightly. For RLL-1, such an alteration would require editing a few (hopefully one) units; as opposed to actually going into the guts of the ESBS's code, and massaging the LISP code found there. It would be useful to build systems which "knew" how to construct automatic programming systems, for example; or which were actually experts at constructing ESBSs. This seems to lead into an infinite descent. The obvious solution is to insure that the program-generating system is capable of generating itself. As the program can reason about its own operations, (in the same way it "understands" other programming constructs) it can serve as a tool capable of modifying itself. The CHI automatic programming system, described in [Phillips], is also based on this idea.

This brief survey of AI representational systems (i.e. systems designed to facilitate the construction and manipulation of other large and complex bodies of knowledge) was intended to convey our basic philosophy: That making the internals of a system explicit, self-describing, and most importantly, modifiable is a big win. Such systems will remain viable and in common use considerably longer than their opaque cousins, and will even be easier to use throughout their lifetimes. Many confusing issues are solved, or simply become non-issues, when the system itself no longer has covers to hide under. For example, when the interpreter is visible, there is no longer any question what the semantics of, say, a link really means -- it is simply what happens when one explicit well-defined structure, called the interpreter, evaluates another well-defined construct, the link.

8. CONCLUSION

The RLL-1 system is currently at a plateau -- stable and usable, but by no means complete. It is only through continued use, by a wide cross section of researchers, that optimal directions for its future effort will be revealed. Requests for additional documentation and access to RLL-1 are encouraged. We see a myriad of future directions for RLL-1 to take. We have attempted to indicate throughout this memo what has been done, and what is still pie-in-the-sky; these are especially noted in Sections 5 and 6. In addition, RLL-1 should one day be able to do all the examples suggested in Appendix B. Much future research, in a number of sub-disciplines, will be required before it can successfully act in a "Euisko-like" manner -- in particular, effective utilization of rules and ability to derive and use apt analogies and super-concepts. Beyond all of these, a complete RLL-1 system should have a sophisticated front end, capable of handling the linguistic nuances associated with data (dare I say "Knowledge") communication; and be truly able to integrate diverse representational pieces, at a level beyond the primitive one it has now achieved.

The additional tasks fall into many different categories. Some merely require a large amount of work (e.g., incorporating other researchers' representational schemes and conventions); while others will have to wait until important philosophical decisions have been reached (such as how to handle

23 Evolution, as well, seems to agree with this last idea, preferring adaptable organisms over those which contain, built-in optimized features. Compare the extinct dinosaur, unable to adapt to new situations, with two of nature's most successful species: Man, who can modify himself to his environment using technological devices (e.g. putting on a coat), or the insect, which uses it last reproductive cycle to "reconfigure" its species as new situations arise (such as the introduction of DDT).
intensional objects and beliefs, and how to attain "epistemological purity" of the initial RLL-1 system).

To support our "universally applicable" arguments, we intend to exhibit a large collection of distinct representation languages which were built out of RLL-1; this we cannot yet do. Several specific applications systems live in (or are proposed to live in) RLL-1. Knowledge bases already started include RLL-1's original raison d'être EURISKO (discovery of heuristic rules), E&E (combat gaming), FUNNEL (taxonomy of LISP objects and functions) and PROVER (a non-resolution theorem prover written by Larry Hines). WHEEZE (a diagnosis program for pulmonary function disorders, reported in [Smith&Clayton]) has been written in a sister system to this RLL-1. ROGL (Jim Bennett: guiding a medical expert to directly construct a knowledge based system) and VLSI (Mark Steffik and Harold Brown: a foray of AI into the VLSI layout area) are two tasks in search of a representational scheme; both are seriously considering using RLL-1.

It is the philosophy of an rll, more than our particular implementation, that we hope this paper will convey. There is a great need for a flexible and extensible high level language in which to create and maintain the sophisticated procedures an AI task requires. The user should have ready access to a stockpile of commonly-used parts, together with the tools required to modify, and compose these pieces. Experience in AI research has shown this goal has been all but neglected. A representation language language addresses this challenge. In addition to providing these tools, an rll will leave the pieces of a representation in an explicit and modifiable state. By performing simple modifications to these representational parts (using these specially-designed manipulation tools) the user can build new representation languages, which can be created, debugged, modified, and combined with ease. This should ultimately obviate the need for dozens of similar yet incompatible representation languages, each usable for but a narrow spectrum of tasks.

It is our hope that RLL-1 will spawn a new generation of such foundation systems, all designed to be flexible and extensible. Eventually, we envision, such languages will together synthesize a single rll system, sufficient to handle every need a user will ever have -- much in the manner LISP has served the AI community all these years; only at a higher level of abstraction and usefulness.

ACKNOWLEDGEMENTS

The work reported here represents a snapshot of the current state of an on-going research effort conducted at Stanford University. Researchers from SAIL and HPP are examining a variety of issues concerning representational schemes in general, and their construction in particular (viz., [Vick&Hietel] and [Van.Hietel]). Professor Douglas Lenat initiated many of the ideas presented here, and supplied essentially all of the research directions. I especially thank Michael Genesereth for his frequent insights into many of the underlying issues, and for his near-continuous encouragement and assistance in preparing this document. He and David Smith have been instrumental in developing and honing many of the ideas presented here. Critiques by Tom Pressburger, Steve Tappel, Sue Angerbrandt, and Paul Cohen did much to help shape this paper, as well as many of my views. Mark Steffik, Terry Winograd, Danny Bobrow, and Rich Fikes conveyed enough of the good and bad aspects of KRL and UNITS to motivate us towards an rll. Greg Harris implemented an early system which performed the task described in Section 4 and Appendix A. Others who have directly or indirectly influenced this work include Bob Balzer, John Brown, Cordell Green, Johan deKleer, and Rick Hayes-Roth. Finally, I am grateful to David Smith, for providing the demand unit swapping package ([Smith]) we used to sidestep InterLisp's space limitation. The research is supported by NSF Grant #MCS-79-01954 and ONR Contract #N00014-80-C-0609.
This appendix expands the example shown in Section 4. It is designed to illustrate how the initial version of RLL-1 actually goes about generating and caching appropriate slot values. As we have emphasized several times earlier, this only represents one particular method RLL-1 allows; any user may choose to follow this set of conventions, or design his own.

A.I: How (GetValue U S) really works:

Suppose immediately after creating the Father unit, as done in Section 4.3, the user asks for Karl Philipp Emanuel's father, by typing (GetValue 'KPE 'Father).

The GetValue function is very simple -- it calls Father:ToGetValue on the arguments KPE and Father. Technically, this would require determining (GetValue 'Father 'ToGetValue), which would, in turn, necessitate looping endlessly on the call (GetValue 'ToGetValue 'ToGetValue).

To avoid this trap, the GetValue function has "pre-compiled" the result of this (GetValue 'ToGetValue 'ToGetValue) call in the appropriate place in GetValue. Hence, GetValue is defined as

(DEFINE GetValue (Unit Slot) (APPLY* (GetAccessFn Slot 'ToGetValue) Unit Slot),

where GetAccessFn is the function stored on ToGetValue:ToGetValue, rather than the desired, but unrunnable

(DEFINE GetAccessFn (Unit Slot) (APPLY* (GetValue Slot 'ToGetValue) Unit Slot).

Note GetValue is the only trouble-maker. All of the other accessing/modifying functions, such as PutValue or RenameUnit, avoid this added hassle.

This GetAccessFn function is also fairly simple. (GetAccessFn 'Father 'ToGetValue) first checks if a value is physically stored on the the ToGetValue slot of the unit Father, and finds none. (Had GetAccessFn found a value there, it would have returned that value.) Otherwise, GetAccessFn falls back on FindDefault's method of inheritance, (shown in Appendix B.1) and so scans the prototypes of Father, in order, searching for the first prototype it can find which has a ToGetValue slot. Here, this search will walk all the up to TypicalSlot, find DefaultGetValue stored there, and return that value.

Like GetAccessFn, DefaultGetValue first tries a simple associative lookup (essential a GETPROP), but finds there is no Father property stored on KPE. DefaultGetValue then tries a more sophisticated approach: rather than look up the prototypes of Father (as GetAccessFn would have done,) DefaultGetValue asks the Father unit how to compute the Father of any person. This information is stored on Father:ToCompute. Thus the (GetValue 'KPE 'Father) call effectively becomes

(Apply* (GetValue 'Father 'ToCompute) 'KPE).

Notice this calls GetValue recursively. Once again there is no value stored here, on the ToCompute slot of the unit called Father. The call therefore expanded into

(Apply* (Apply* (GetValue 'ToCompute 'ToCompute) 'Father') 'KPE).

Luckily, there is a value on the ToCompute slot of the unit ToCompute. The functional stored in ToCompute:ToCompute, TCTC, takes an argument, X, (the name of a slot,) and returns a function which, given a unit U, computes the value of the X slot of that unit. TCTC instructs RLL-1 apply the function HighLevelExpand to the HighLevelDefn of X. HLD. This function first finds the slot-combiner S which X employs, (this is the CAR of HLD) and applies S:ToCompute function on the relevant arguments -- the other elements of HLD. Our call is now expanded out into

(Apply* (Apply* (GetValue 'Composition 'ToCompute) 'Husband 'Mother) 'KPE).

In fact, an essential invariant of our system is that GetAccessFn be a fixpoint of Fn = (Fn 'ToGetValue 'ToGetValue). By the definition of GetValue, this implies that Fn = ToGetValue 'ToGetValue (where we have replaced "GetAccessFn" with "Fn" in GetValue).
The slot-combining unit called Composition does indeed have a ToCompute slot; after applying it, we have (roughly)

\[ \text{Apply}^* (\lambda (x) (\text{GetValue} (\text{GetValue} x \ \text{Mother}) \ \text{Husband})) \ KPE. \]

This asks for the Mother slot of KPE, which is always physically stored in our knowledge base, and then asks for the value stored in the Husband slot of that unit. The final result, JSB, is returned. It is also cached (stored redundantly for future use) on the Father slot of the unit KPE. Section 5.2 elaborated the details of this process.

A.2: How (and why) caching really works

Consider what will happen the next time the user requests KPE:Father. As before, Father:ToGetValue will be called, and return DefaultGetValue. This will be invoked and, as before, see if there is some value stashed in KPE's Father slot (this is Step 1 of Algorithm 1). This time that test returns successfully, and the result, JSB, is returned. Notice all the other running around has been avoided. Much of this work, such as determining the value of Father:ToGetValue can be done at compile time, through the use of clever macros, (see Appendix D.3) to speed up this process ever more. Our goal is for such standard cases -- i.e. retrieving a stored value -- to be about as fast as a simple GETPROP (see Appendix D.1).

We might later ask for, say, the Father of Wilhelm Adolf Bach (WAB). As usual, Father:ToGetValue is used. It, finding no value is stored on WAB's Father slot, now need Father:ToCompute, and so "enter" Algorithm 2. This time Father:ToCompute will be found in the first step and simply returned. When this function is applied to WAB, it will return the value of WAB's Father, Wilhelm Friedmann Bach; after caching that value for future use.

Note the efficiency in the mechanism -- only if a value, such as KPE:Father or Father:ToCompute, is requested will RLL-1 actually compute this value; at which point, it does only the work required. If such a request is never issued, no effort will be wasted, as this therefore-useless value, (e.g. Father:ToCompute,) will never be calculated. Subsequent calls to x:Father will not recompute this function, but will merely use the value stored in Father:ToCompute. Section 5.2 expands and generalizes this point.

There is an obvious space-time tradeoff going on here: retrieving a cached value does take much less time than would be spent recomputing that value; but it does cost that additional storage. To simplify this example, we implied RLL-1 would always decide to cache, i.e. in favor of saving time over space. In general, RLL-1 tries to weigh time versus space considerations in determining whether, and how, to preserve a value. For example, RLL-1 may then decide to "SELF-COMPILE" the function now stored in Father:ToCompute. That is, the first time this function is run, it will compile itself into machine-level code, and store this more efficient form where it will be sought before the interpreted, source code. In this way frequently used functions can become faster and faster.

A.3: Other Accessing Functions

To demonstrate another accessing function, suppose the user wished to enter a value into KPE:Mother. This could be achieved by typing

\[ \text{PutValue} \ 'KPE' \ 'Mother' \ 'AMB'. \]

As before, PutValue is a very simple function. It asks the Mother unit for its ToPutValue slot, and then applies this function to KPE, Mother and AMB. By the same process outlined above, Mother:ToPutValue requires ToPutValue:ToGetValue. This function has been stored, and is again GetAccessFn. (This same GetAccessFn function is used in all of the accessing and updating functions.) As before, GetAccessFn Mother:ToPutValue) first asks Mother for its ToPutValue slot. Finding not value there, it examines Mother's prototypes. \{P_j\}, stopping and returning the first nonNIL P_j:ToPutValue. Unless intercepted, (see Appendix D.1) this (GetAccessFn Mother:ToPutValue) will return the value living in TypicalSlot:ToPutValue. DefaultPutValue. This default function verifies that AMB is an acceptable value for a Mother slot, then stores AMB on the Mother slot of KPE, does various knowledge base truth maintenance tasks, and stops.
AnyPhysicalObject
  TypicalPhysicalObject

AnyLivingThing
  TypicalLivingThing

AnyAnimateObject
  TypicalAnimateObject

AnyAnimal
  TypicalAnimal

AnyMammal
  TypicalMammal

AnyMale
  TypicalMale

AnyPerson
  TypicalPerson

AnyMusician
  TypicalMusician

AnyGerman
  TypicalGerman

KPE
  IS-A: (AnyMusician, AnyGerman, AnyMale)
  AllIsA: (AnyMusician, AnyGerman, AnyPerson, AnyMammal, AnyMale, AnyAnimal, AnyLivingThing, AnyAnimateObject...)
  Prototypes: (TypicalMusician, TypicalGerman, TypicalPerson, TypicalMammal, TypicalMale, TypicalAnimal, TypicalLivingThing, AnyAnimateObject, ...)

Figure 9.7 - Portion of the frame hierarchy which dominates KPE.
B. APPENDIX - Additional Examples

B.1: Creating a Whole New Family of Types of Slots

RLL-1 comes equipped with several inheritance and retrieval mechanisms. One powerful and useful one exploits the [Minsky] frame-like hierarchy which structures the RLL-1's initial knowledge bases. Many slots can derive an If-Needed value by default, extracting this value from the nearest relevant prototype. Thus, the facts that "Clyde is an Elephant" and "Elephants are grey" should be sufficient to deduce that "Clyde is grey". The constructs and mechanisms which allow such deductions are shown below.

As part of a frame-like system, every RLL-1 unit has an IS-A slot which lists the classes to which this unit belongs. For example, KPE:IS-A is the list (AnyMusician, AnyMale, AnyGerman). Each of these units, in turn, points to a list of superclasses of that set -- AnyPerson is included in AnyMusician:SuperClasses. Note, as these SuperClasses links encode the superset relation, KPE will be a member of the class represented by AnyPerson, as well as AnyMusician, AnyMale, and AnyGerman. The AllIsas type of slot encodes this more complete "E" relation. See Figure 7.

Associated with each class, "AnyX", is a unit which houses facts which are typically true of each member of that set, by convention named "TypicalX".25 So TypicalMusician:HasInstrument will be True, whereas TypicalPerson:HasInstrument is False, as most people do not own an instrument. The Prototypes of a unit, U, is a list of the typical members of each member of the list U:AllIsas, (in order of increasing generality). So KPE:Prototypes is (TypicalMusician, TypicalGerman, TypicalPerson, TypicalMammal, TypicalMale, TypicalAnimal, TypicalLivingThing, TypicalPhysicalObject, TypicalThing).

As suggested above, facts true of each26 instance of AnyX should be stored on the unit TypicalX. If the user then asks whether some member of AnyX, U, has this property, P, this FindDefault retrieval mechanism will go up U's Prototypes until it finds a prototype which says something about P. Unless intercepted, this search will terminate when it encounters TypicalX, and return the value P implied. So we would expect KPE:HasInstrument to be True, as TypicalMusician:HasInstrument is True. The fact that TypicalPerson:HasInstrument is False is, here, irrelevant, as the FindDefault search will never reach TypicalPerson. (This inheritance mechanism is essentially the same as the one FRL and KRL use.)

Thus the AnyMusician and TypicalMusician units were used to implicitly define a new class of objects, by indicating new default values for members of this class. (In this case, the value for the HasInstrument slot.) We can exploit the way FindDefault works to define a new subclass, AnyS, of an existing class, AnyC. In the same way all musicians (such as KPE) "preferred" to get their values from TypicalMusician over TypicalPerson, values placed on TypicalC override those stored on TypicalS. The semantics of TypicalExample corresponds to our intuitive ideas. Figure 8 below shows how we, and RLL-1, can allow Duckbill Platypuses to lay eggs, even though every duckbill platypus is a mammal, and "all" mammals give live birth.

---

25 We have attempted to follow a few naming conventions: The unit "AnyX" represents the class of all Xs. "TypicalX" is a unit which stores information which is typical (that is, defaulted) for each element of the class "AnyX".

26 This "each" criterion is not really required. We saw earlier in the TypicalMusician and TypicalPerson example that assertions stored on TypicalX units reflect default, as opposed to universal, statements. In addition, the right thing would happen if we later add to the Elephant example above that "Clyde is an Albino" and "Albinos are by definition, white". The value of Clyde:Color would then be White, not Grey. This requires placing an epistemological mark on TypicalAlbino:Color to indicate this definitional quality: Appendix B.4 will show one way of doing this. RLL-1 can handle the standard problems which arise when dealing with non-monotonicities [AI Journal] and "default reasoning" [Reiter]. The above example intentionally avoids such complexities.
This same mechanism can be used for types of slots as well. As slots are themselves encoded as units, they too can have defaulted properties. By creating a new class of types of slots, we can provide its members with a different set of defaults; inherited only by these members.

Examples 4.2 and 4.3 above intentionally glossed over details of the actual retrieval mechanism used. As shown in Appendix A.1, computing KPE:Father actually requires first asking Father for its GetValue value, and then applying this function to KPE. The value of this Father:GetValue will be inherited from one of Father's Prototypes. Similarly, storing a value in the Father slot of KPE (or any other unit) requires first determining the value of Father:ToPutValue. As with GetValue, the value of (GetValue 'Father 'ToPutValue) will be inherited from the first one of its prototypes which has some stored ToPutValue value. Unless intercepted, this will reach the value DefaultValue, stored on TypicalSlot:ToPutValue. This information is portrayed in Figure 9. Note (GetValue 'Father 'ToPutValue) = (GetValue TypicalSlot 'ToPutValue) = DefaultValue.

Suppose we want to create a whole different category of types of slots -- one which, for example, prints a message before performing each of its knowledge base consistency modifications. (In Appendix A.2 we saw that a unit's Father slot is deleted whenever that unit's Mother slot is updated -- this is such a modification.) A general way of doing this requires first creating a unit representing a new class -- named, say AnyChattySlot -- and its associated typical example,
TypicalChattySlot. Now TypicalChattySlot:ToPutValue is filled with the function ChattyPutValue, which resembles DefaultPutValue, (the function found on TypicalSlot:ToPutValue) but which prints that updating message where appropriate. In Figure 10, we show Mother as an Example of AnyChattySlot, by replacing the value AnySlot with AnyChattySlot in Mother's IS-A slot. In this state, the user will be told that KPE's Father slot was being deleted if KPE:Mother is ever reassigned. Here (GetValue 'Mother 'ToPutValue) = (GetValue 'TypicalChattySlot 'ToPutValue) = ChattyPutValue. Note (GetValue 'Mother 'ToGetValue) is still DefaultGetValue, inherited from TypicalSlot; this is true only because TypicalChattySlot had nothing to say about ToGetValue.

```
Name: AnySlot
IS-A: (AnyClassOfObjects)
SuperClasses: (AnyFunction, ...)
SubClasses: (AnyChattySlot)
Description: Represents all slots.
TypicalExample: TypicalSlot
AllExamples: (Father, ...)
```

```
Name: Mother
IS-A: (AnyChattySlot)
SuperClasses: (AnySlot)
Description: Represents all chatty slots.
TypicalExample: TypicalChattySlot
Examples: (Mother, ...)
```

```
Name: TypicalSlot
TypicalExampleOf: AnySlot
ToGetValue: AnySlot
ToPutValue: DefaultPutValue
```

```
Name: TypicalChattySlot
TypicalExampleOf: AnyChattySlot
ToGetValue: ...
ToPutValue: ChattyPutValue
```

Figure 10 - Altered framework for Mother slot.

In terms of this inheriting mechanism, there is nothing special about slots, or classes of slots. We could, with equal ease, have generated new categories of poultry by this same mechanism. All slots whose "inheritance criteria" matched ToGetValue's (i.e. whose ToCompute was set to FindDefault,) would inherit features from this new typical chicken, as the descendants of AnyChattySlot inherited slot values from TypicalChattySlot, examples of AnyMusician could derive properties from TypicalMusician, or examples of AnyDuckbillPlatypus found default values on TypicalDuckbillPlatypus.27

B.2: Overview of EURISKO System
Use of Agendae, Tasks and "Generalized Production Rules"

This RLL-l work developed from a desire to produce (the foundations) of EURISKO -- a system capable of discovering new heuristics. To facilitate this pursuit, a very flexible processing structure had to be created. One mechanism selected employed multiple agendae, each containing a host of domain-specific tasks, pertinent to a single coherent topic. To "run" a task, EURISKO first finds the set of relevant heuristic rules, and then invokes each rule as appropriate, until reaching some termination criterion (such as exhaustion of all allotted resources). What complicates this process is the fact that determining such rule sets is itself a task, which appears on an agenda. Similarly,

---

27 It is worth repeating the comment stated in Footnote 11: This inheritance mechanism is NOT built into RLL-l, and is never forced on the user. It is simply one of a large collection of parts he may use in whatever manner he chooses. In fact, only some of RLL-1's initial slots use only this inherit-by-default method to deduce a needed value; many others will use it as a last resort, after a preferred alternate method (such as parsing the HignLevelDefn) has failed; and yet others never even consider it.
deducing how to execute such a collection of rules is a task, as is essentially every other non-trivial process.

The standard flavor of production rules, containing just an "IF" predicate, followed by a set of "THEN" actions, is unnecessarily restrictive. Furthermore, in most systems, these rules are placed in an unordered list, which must be scanned, in toto, to find all relevant rules. Several other recent projects have also found standard rule systems severely lacking. Attempting to address (and hopefully solve) such inadequacies, [Fagan] put additional data into each rule; and [Lenat], [Aikins], and [Smith&Clayton] found natural (and useful) ways to organize collections of rules into a hierarchy. These, and other limitations of "Pure Production Rule Systems", are discussed in [Lenat&McDermott].

Rules still possess too many good qualities to pass them up. One would still like to store the procedural as well as the declarative knowledge in small chunks, each including not only this morsel of information, but also some idea of when it should be used, and how. The obvious drawback to this is efficiency: considering just the overhead each subunit will require, this segmented body has to take more space to hold than a comparable single program. Furthermore, it will usually require more time to interpret a collection of rules than to run that equivalent piece of code. EURISKO has finessed this dilemma by retaining both versions (i.e. distinct encodings) of an operation (and possibly others) and utilizing each when that encoding is most appropriate. The most inefficient, but easiest to modify, version will be this set of rules.

Maintaining consistency among these versions, of course, is a non-trivial problem. However, the mechanism shown in Appendix A.2, with respect to slots, is adequate for the task. Basically, we would need to associate with each successively more complex version the procedure for building it out of simpler forms -- a compiler of sorts. Then when, for example, the collection of rules is modified, EURISKO would simply invalidate (e.g. erase) the source code version. (This might start a cascade, which would cause the compiled version and possibly other things dependent on this function to be erased as well.) If this function is later needed, EURISKO would try to retrieve the now-erased function. This would, in turn, trigger an IF-Needed procedure, which would construct (and cache) this source code from the rules; and EURISKO goes on.

To accommodate a system as complex and self-referential as EURISKO, it is necessary to store a vast assortment of information in each rule; and to organize them into a usable structure. For obvious reasons, we chose to represent each rule as a full unit, whose slots indicate not only how and when to fire it, but other usable, if not executable, facts as well (e.g. HowCreated, AverageRunningTime or OverallWorth).

A task's first step is to search the knowledge base for the set of relevant rules. (Recall this selection and gathering process is itself just another example of a task to be performed, which in turn, will be based on a collection of rules, and so forth. By appropriately caching necessary values, RLL-1 avoids the problem of infinite regressions.) Running this task will, at first, require slowly interpreting these rules, with a meta-level process observing. During such early runs this collection will be modified to optimize the order of rule firing, and prune away extraneous and superfluous rules. Once a stable rule set has been determined, this collection will be composed into another encoding of this procedural information: a single piece of runnable code. If at this point, the generating body of rules can simply be swapped out, in much the same way source code need not be present when its compiled form is being executed.

In general, the result of this arduous process will be cached. The next time this task, or any sufficiently similar one, is to be run, this "compiled information" will be used, (which avoids...
rerunning the entire task-generating process outlined above). If trouble is encountered later, or when considering applying this operation to a new domain, the original rule set can easily be retrieved and reanalyzed, to be corrected or augmented as required. In most cases, however, the efficient code is simply executed. Hence, in the long run, all of this flexibility is at essentially no long-term cost, using only the mechanism we saw used for Example 4.3!

With this model, rules fit naturally into RLL-1's Cognitive Economy framework (see Section 5.2) and we have attempted to incorporate them extensively in RLL-1's internal processing as much as possible.

B.3: Creating a New Inheritance Mode

Suppose a geneticist wishes to define a type of inheritance, one which skips every second generation when determining the properties of a new unit. He browses through the hierarchy of units descending from the general one called Inheritance, until he finds an existing unit, InheritFromEvery2ndGeneration, which closely resembles his goal inheritance facility. This he copies into a new unit, InheritFromEvery2ndGeneration. Editing this copy, he finds a high level description of the path to be taken during the inheritance. To achieve the generation skip, he replaces each single occurrence of "Parent" by "GrandParent" (or by two occurrences of Parent, or by the phrase (Composition Parent Parent)) in this part specification. After exiting from the edit, the new type of inheritance will be active; RLL-1 will have translated the slight change in the unit's high-level description into a multitude of low-level changes. If the geneticist now specifies that Organism#34 is an "InheritFromEvery2ndGeneration offspring" of Organism#20, this will mean the right thing: that Organism#34 has about the same chance of glaucoma as Organism#20 had, and that we should expect their political slants -- e.g. Radical/Conservative -- to be similar. It is worth noting that the tools used (browser, editor, translator, etc.) are themselves encoded as units in RLL-1.

It is no harder to create a new type of slot format (OrderedNonemptySet), slot combiner (TwoMost, Starring), or datatype (MustBePersonOver16), than it was to create a new slot type or inheritance mechanism. Explicitly encoding such information helps the user (and us) understand the precise function of each of the various components. We do not yet (and probably never will) have a complete set of any of these components, but are encouraged by empirical results like the following: The first two hundred slots we defined required us to define thirteen slot combinators, yet the last two hundred slots required only five new slot combinators.

B.4: Epistemological Status

There are many ways of handling the wealth of problems associated with representing the epistemological status of things like facts, assertions and beliefs. We assumed, in designing the initial RLL-1 system, that most statements are, in fact, facts. Based on this, it is efficient to leave these completely "unmarked", and realize the additional cost of handling statements with other "epistatuses" (shorthand for "epistemological statuses"). Hence, to represent the statement that John believes that Mary is 37 years old, RLL-1 adds the ordered triple (*Do* SeeUnit AgeOfMary0001) to the the Age slot of the Mary unit. RLL-1 creates a unit called AgeOfMary0001, fills its *value* slot with 37 and its Epistatus slot with "John believes". See Figure 11. Note this mechanism suffices to represent belief about belief (just a second chained ScUnit pointer), quoted belief ("John thinks he knows Mary's age", by omitting altogether the *value* slot in some AgeOfMary0001 subunit), situational fluents, etc. This mechanism can also be used to represent arbitrary n-ary relations (such as "John gave the ball to Mary"), thereby escaping the limitations associated with associative triples (i.e. Unit/Slot/value). Other ways of handling these are discussed in [Genesereth & Lenat].

---

This "*Do*" prefix is special. Appendix D.4 shows this notation sufficient to handle effectively all special cases of slot value, if awkwardly.

---
Figure #11: Representing "John believes that Mary is 37, but she’s really 39. When she was married, she was 21."
In Section 5.3 we distinguished semantic slots from syntactic ones. Technically, syntactic slots of a unit U belong in the meta-node above U, as they refer to the unit U qua unit. We could create such a meta-unit, e.g. FactsAboutTheUniKE, to store all the facts about the KPE unit, and store this within our system. This FactsAboutTheUniKE unit would have slots like TimeOfCreation, or Size. The values stored there, such as 3 October 1980, or 96 Lisp Cells, are facts about the KPE unit, and NOT about KPE himself, who was born on 3 March 1714, and is actually 5'10" tall. The obvious question is "When was the FactsAboutTheUniKE unit created?". To find this answer, one should logically go to the FactsAboutTheUniKEFactsAboutTheUniKE unit. And when was that unit created? and so on...

So we have chosen to compress these two, logically distinct units into a single physical entity. In addition to the argument above, empirical evidence has shown that these two units consistently are used together; and so should be, for example, swapped into core together. Epistemological marks are used to distinguish those slots which refer to that external entity, from those which, in fact, pertain to this unit itself. Slots in the first category, the semantic slots, work cleanly for all of our functions. There were several difficulties which arose when dealing with Syntactic slots - those which pertain to this unit qua unit.

Diagram #1 in Appendix E, shows that essentially all of the units of the system descend from the "AnyConcreteThing&Unit" class. Our intent was that each such "concrete-thing&-unit" represent both some entity in the "external" world, and itself. The cleanest way of achieving this was as follows:

Consider first the class of all centaurs. To indicate that each centaur is both a man and a horse, we would place on the TypicalCentaur unit a ComposedOf slot, whose value is the list (TypicallMan TypicalHorse). The meaning of this is straightforward: each centaur should inherit characteristics from both its manliness and its horseiness.

Now examine the TypicalConcreteThing&Unit unit. This unit qua unit, is in fact composed of two parts -- its pieces come from the fact it represents a unit, and the fact that it represents a concrete thing. So its meta-unit, the FactsAboutTheUniKETypicalConcreteThing&Unit, should have the ComposedOf value of (TypicalUnit TypicalConcreteThing). As this meta-unit is NOT disjoint from the object-level TypicalConcreteThing&Unit unit, we give it a MyComposedOf slot, whose value is that (TypicalUnit TypicalConcreteThing) list. (Note TypicalAbstractThing&Unit is handled the same way.)

The next difficulty arose when dealing with inheritance. Consider the task of creating a new unit, say KPE, which is defined to be an Example of AnyMusician. A simple scheme, which would work ideally for semantic units, would be to map through the set of inheritable slots of TypicalMusician (note this is value stored on TypicalMusician:MyInheritableSlots) and, for each slot, S, store on KPE:S the value held in TypicalMusician:S. So KPE:Size would receive the Size of TypicalMusician, which is what you would want.

Now we know that every unit representing a musician will be created by Procedure #114, the musician creating routine. It would be nice to store this syntactic slot, MyCreator, on TypicalMusician, and let each new musician inherit this information. Unfortunately, this TypicalMusician unit was itself created by the routine which specialized in creating typical examples, Procedure #21.

---

10 We indicate this difference by a simple naming convention: The name of each syntactic slot will begin with "My" -- i.e. TimeOfCreation versus MyTimeOfCreation, or Size versus MySize.
The solution we used is for each type of slot to know to which category it belongs - Syntactic vs Semantic. Semantic slots work unhindered. However, the unit for each syntactic type of slot has a special slot, StoreTypAs, which points to the name of another slot, on which the value to be inherited is stored. Here, MyCreator:StoreTypAs had the value Creator-Instances. The value stored on TypicalMusician:MyCreator is Procedure #21, and refers only to this particular TypicalMusician unit. The inheritance scheme, on seeing MyCreator on TypicalMusician:InheritableSlots, knows enough to use the value stored on TypicalMusician:Creator-Instances, which, happily, is Procedure #114. This is the value to which KPE:MyCreator is initialized.

(NOTES: The Inheritance mechanism demonstrated above is a vastly oversimplified version of one of the several inheritance schemes resident to RLL-1. Like the rest, it is encoded as a unit, and can be modified by any user to serve his particular needs.)
D. APPENDIX - IMPLEMENTATION PRINCIPLES

D.1: Make Standard Case Fast: This system should perform not only correctly, but expeditiously as well. Towards this end, a self-imposed design criterion is to optimize what we expect to be the standard, most common case: even at the anticipated increased cost for less common situations. Note this philosophy is reflected in CORLL's retrieval strategy as well -- we have minimized the speed of accessing a slot's value when that unit is in core, and accept the longer times required when that unit must be read in. The underlying assumption is that after a unit has been accessed once, it will probably be used several more times before being read out. Furthermore, this reading process has to be fast, as it will undoubtedly be central to the inner-most loop of RLL-1's operations.

For similar reasons, we have chosen to optimize retrieving a slot's value, at the expense of storing such a value. RLL-1 will, in general, also spend extra time storing a computed value; as this is a one time expenditure, and it is likely this value will be needed often.

D.2: Least Commitment: Throughout RLL-1's ongoing design we have attempted to minimize the constraints the RLL-1 functions force upon the unit, as every eventual user will have to tolerate these. (This point is closely related to the issue of having an unbiased initial system, discussed in Section 6.7.) To accomplish this, while still permitting the basic RLL-1 package to perform the various detailed tasks expected of a usable knowledge base system, this system includes a versatile KB manipulating subsystem. The functions this requires are stored as the values of various slots of certain high-level units, where they belong (see Section 5.5). As shown above, the user with a more specialized task in mind can readily override these general functions, by simply rewriting these slots' values with functions of his own.

(These specific functions, while essential for the starting system to work, are not sacred. Indeed a self-serving designer may use them for the single task of storing a different set of functions in their places. Thereafter all GETs and PUTs would go through these just-entered functions. The bootstrapping functions used to prime the pump are documented in Appendix F.2. These were designed to demonstrate one possible, easily extensible, way in which this system could work.)

D.3: Compile Time Macros: Associated with each type of slot is a pointer to the routine which this slot uses to perform write and retrieve its value from a unit. (See Section 4 and Appendix A.) Determining, say, the \textit{Wife} of JSB requires first asking the unit \textit{Wife} for its \\textit{ToGetValue} slot, and then applying that function to the unit JSB. (Similar RLL-1 functions are used for putting a value, or initializing a unit, etc.) In most cases, \textit{Wife:ToGetValue} will remain fairly constant over long periods of time -- in particular, from the time a function which asks for the \textit{Wife} of some unit is compiled, until the time that function is actually executed.

To shorten the run-time cost, we employ various sophisticated macros for these basic accessing functions. Whenever possible, these run-around the Knowledge Base at compile time, determining the specific getting or putting function associated with this slot. This function is called directly at run time, avoiding the overhead mentioned above. Note that RLL-1 has stored on the Wife unit the set of "macro-ed" functions, which enables it to invalidate the compiled form of these functions if \textit{Wife:ToGetValue} is later changed.

D.4: Special Slot Values: By definition, the value of the \textit{S} slot of the unit, \textit{U}, is what (Getvalue \textit{U} \textit{S}) returns: and this, in turn, depends on the function stored (or virtually stored) on \textit{S:ToGetValue}. Unless intercepted or overwritten, the value of a \textit{S:ToGetValue} will be DefaultGetValue. With this in mind, this section discusses what this standard retrieval function, DefaultGetValue, regards as values.

The standard value of a slot (i.e. the value returned by DefaultGetValue) is exactly the value physically stored there, except for the following few cases. To disambiguate the case when a unit does not have a slot from when the value of that slot is actually NIL, the values \textit{NoEntries} (indicating that this value is known to be the empty list) and \textit{NoEntry} (meaning an empty single value,) are used. (Following the delegation of responsibility comment mentioned in Section 5.5, it is the various
individual formats which "know" and use this fact -- N.B. it is NOT built into any of the retrieval functions.) When the value of this slot is not known, the value RecomputeMe will be stored as its value. This is built into the basic defaulted accessing functions, in that none of them distinguish this RecomputeMe value from the case of finding that this slot does not physically appear in the unit.

The only other special case allows the user to specify a more arbitrary format for the value of this slot. The value (*Do* "<format-name> v1 v2 ... vN") indicates the values (v1, v2, ..., vN) should only be regarded with respect to the format (or epistemological mark) "<format-name>". For example, (*Do* OneOf Red Yellow) indicates the value of this slot is either Red or Yellow. Note the semantics of this statement are based on the unit OneOf, which is, of course, accessible and modifiable by the user.

As Appendix B.4 shows, there are times a full unit is devoted to store the information associated with Unit:Slot. Here, the entry stored associated with Slot on the unit, Unit, is of the form (*Do* FSeeUnit SlotOfUnit). This uses the format unit FSeeUnit which knows that the real value of Slot is stored in the "value" slot of the unit, SlotOfUnit. RLL-1 is equipped with several other subspecies of indirect pointers; and the user, of course, is free to concoct his own.

Everything described above has been implemented. There is, however, considerable concern that this mixes and confuses epistemological and notational marks. This is now being hammered out; and the way subsequent generations/releases of RLL-1 will handle this situation will depend on these discussions.
Diagram #1 (of report)

Classes of Units
BIBLIOGRAPHY


Davis, Randy, "Applications of Meta Level Knowledge to the Construction, Maintenance and Use of Large Knowledge Bases," Stanford AI Laboratory, Memo AIM-283.


Hewitt, Carl F., "Description and Theoretical Analysis (using schemata) of PLANNER: a language for Proving Theorems and Manipulating Models in a Robot," MIT AI Laboratory, TR-258,


Lenat, Douglas B., "AM: Automated Discovery in Mathematics", 5-IJCAI, August 1977


Sandelwall, Erik, "Ideas about Management of LISP Data Bases", 4-IJCAI, Tbilisi, Georgia, USSR, 3-8 September 1975, pp. 585-591.


*SIGART Newsletter*, February 1980 (Special Representation Issue; Brachman & Smith, eds.).


