TR 901 ARLO: Another Representation Language Offer Kenneth W. Haase, Jr.

Technical Report 901

ARLO: Another Representation Language Offer

Kenneth W. Haase, Jr.

MIT Artificial Intelligence Laboratory

11

This blank page was inserted to preserve pagination.

ARLO

(Another Representation Language Offer)

The Implementation of a Language for Describing Representation Languages

by
Kenneth W. Haase Jr.
MIT Artificial Intelligence Laboratory
Cambridge, Massachussets 02139

This paper describes **ARLO**, a *representation language language* loosely modelled after Greiner and Lenat's **RLL-1**. ARLO is a structure-based representation language for describing structure-based representation languages, *including itself.* A given representation language is specified in ARLO by a collection of structures describing how its descriptions are interpreted, defaulted, and verified. This high level description is compiled into lisp code and ARLO structures whose interpretation fulfills the specified semantics of the representation. In addition, ARLO itself — as a representation language for expressing and compiling partial and complete language specifications — is described and interpreted in the same manner as the languages it describes and implements. This self description can be extended or modified to expand or alter the expressive power of ARLO's initial configuration. Languages which describe themselves — like ARLO — provide powerful mediums for systems which perform automatic self-modification, optimization, debugging, or documentation. AI systems implemented in such a self-descriptive language can reflect on their own capabilities and limitations, applying general learning and problem solving strategies to enlarge or alleviate them.

This empty page was substituted for a blank page in the original document.

Acknowledgements

In the last four years at MIT, so many people – both in the Institute and outside of it – have touched my life in special, magical ways. These acknowledgements are long, but they only begin to offer appropriate thanks for gifts of knowledge, support, and care from all those who have so freely given.

One's footing always feels firmer when secured on a certain foundation, and the advice and guidance of Marvin Minsky and Patrick Winston has been just such a constant support and foundation during my career at MIT. The research described here and the research and studies that led to it would not have been possible but for their advice, inspiration, and well-placed admonitions. More than contributing to this research, they have contributed to *me*, and my debt is both to them and the intellectual environment they have created at MIT.

Without particular aims – in the form of specific scenarios or particular problems – artificial intelligence goes little beyond "engineering for philosophers". The ends of the research describe in this thesis have often been shaped and molded by the dreams and quests of researchers at ATARI's labs in Cambridge and Sunnyvale. The environment and "dreams in the making" at these institutions were the products by many people, but I especially wish to thank Alan Kay, Margaret Minsky, and Cynthia Solomon for their support and inspiration. These individuals – and the many others once or still at ATARI's labs – helped shape an environment in which tomorrows could be made.

MIT's Artifical Intelligence lab has been another constant source of intellectual support and inspiration in my career at MIT. The students, faculty, and staff of the AI lab have greatly enriched the intellectual content of the author's life. Particularly, Phil Agre, John Batali, Dave Chapman, Ken Forbus, Dave Levitt, and John Mallery have – with their constant interest and dialouge – both sustained the author and greatly improved the intellectual aims of the research here described.

Despite these firm supports, the sanity of the author has often been strained by the barrage of deadlines, classes, and the prodigous bulk of MIT as an institution. Saving me from this barrage are many friends who have enriched and blessed my life. Thank you Gumby, Margret, Danny, Gary, Charity, Hazel, and Jim. To you and all the others I fear I have forgotten, I love you all.

The Beacon Hill Friends Meeting has been a source of joy and inspiration – of a different kind – since my discovery of it nearly two years ago. To all of you – but especially Howard, Ginny, Gordon, and Mary – I bear a debt of joyous reflection and blessed quietness. God bless you all, dear Friends.

Much of the support for this research was provided by the Advanced Research Projects Agency of the Department of Defense under Office of Naval Research Contract N00014-80-C-0505. Of course, the opinions in this thesis are those of the author and in no way reflect the opinions of the Department of Defense or the US Government. In turn, the opinions of the Department of Defense also in no way reflect the opinions of the author.

Finally, but most constantly, my family has been a support – before and above all other supports – of my studies and research. They have endured my many missteps and failings, and returned love and support. I wish I could adequately return what they have given me.

Ken Haase
Cambridge, Massachussets

This empty page was substituted for a blank page in the original document.

Dedicated to
The Shapers, Teachers, and Dreamers
Of
Atari's Cambridge Research Lab

This empty page was substituted for a blank page in the original document.

CONTENTS

Chapter 1: ARLO: Representing Representations . . . 1
1.1: What Good is Representing Representations? . . . 2
1.1.1: RLL's as implementation languages . . . 2
1.1.2: RLL's as Mediums for Programs Which Grow . . . 3
1.2: Representation as Inference . . . 4
1.2.2: Spontaneous and Deliberated Inference . . . 4
1.2.2: Characterizing Spontaneous Inference . . . 4
1.2.3: The Evolution of Spontaneous Inferences . . . 5
1.4: What is ARLO? . . . 6
1.4: Basic Concepts: A User's Introduction . . . 6
Chapter 2: ARLO's Implementation . . . 10
2.2: Units and Knowledge Bases . . . 13
2.2: The Value Dependency Mechanism . . . 13
2.2.1: Dependency Mechanism Protocols . . . 15
2.3: ARLO Errors and Conditions . . . 16
2.3.2: Anticipating errors . . . 17
2.3.2: Classes of Errors . . . 17
2.5: Reflexive Operators . . . 19
2.5.0: Staunching an infinite regress . . . 19
2.5: Representing Representations: The Details . . . 19
2.5.1: Generic Objects & Shadow Slots . . . 20
2.5.3: Type Restricted Slots . . . 21
2.5.3: Defaulting Slots . . . 21
2.6: The ARLO Coder . . . 22
2.6.1: Representing Programs . . . 23
2.7.0: ARLO's Coders . . . 24
2.7.0: User Defined Functions . . . 24
2.7: The Type System . . . 24
2.8: Archives and Layers: Saving Representations . . . 25
2.8.1: Layers . . . 27
Chapter 3: An Example: Representation . . . 28
3.1: Building Basics . . . 28
3.2: Defining Slots . . . 29
3.3: Inheritance Mechanisms . . . 30

3.4: Shadowing Slot Definitions . . . 32
3.5: Building the data base . . . 33
3.6: At the Console . . . 35
3.6.1: Defaulting of Slots . . . 35
3.6.2: Dependencies and Decaching . . . 37
3.6.3: Other slots . . . 42
3.6.4: Errors . . . 44
3.6.5: Shadowing Definitions . . . 46
3.6.6: Modifying our language . . . 49
Chapter 4: An Example: Introspection . . . 54
4.1: Explanation Structures . . . 54
4.3: Textual Generation . . . 55
4.3: Graphical Presentation . . . 55
4.4: An Explanation of the INQUIR system . . . 57
4.4.4: Units without any prototype. . . . 57
4.4.4: Units with a prototype of **Hacking** . . . 57
4.4.4: Units with a prototype of **Hand Coded Function** . . . 57
4.4.4: Units without any prototype. . . . 57
4.4.7: Units with a prototype of **Hacking** . . . 58
4.4.7: Units with a prototype of **Hand Coded Function** . . . 58
4.4.7: Units with a prototype of **Person Slot** . . . 58
4.4.8: Units with a prototype of **Person** . . . 59
4.4.12: Units with a prototype of **Slot** . . . 60
4.4.12: Units with a prototype of **Shadow Slot** . . . 60
4.4.12: Units with a prototype of **Type** . . . 60
4.4.12: Units with a prototype of **Winner** . . . 60
Chapter 5: Conclusion . . . 61
5.1: Flaws in ARLO . . . 61
5.1.2: Flaws in the Dependency Network . . . 62
5.1.2: Flaws in Combining Slot Actions . . . 62
5.2: Why RLL's are no good . . . 63
A-0.3: Why RLL's Aren't So Bad . . . 65

Appendices

Chapter A-1: An ARLO 'Explanation' . . . 66
A-1.1: Units defined in Arlo: SOURCES; BOOT . . . 66
A-1.1.2: Units with a Makes Sense For slot of `Any-Type` . . . 66
A-1.1.2: Units with a Makes Sense For slot of `Slot-Type` . . . 66
A-1.1.3: Units with a Makes Sense For slot of `Unit-Type` . . . 68
A-1.2: Units defined in Arlo: SOURCES; CODERS . . . 70
A-1.2.2: Units with a Data Type slot of `Any-Type` . . . 70
A-1.2.2: Units not classifiable by `Data-Type` . . . 70
A-1.3: Units defined in Arlo: SOURCES; CODING . . . 71
A-1.3.2: Units with a Makes Sense For slot of `Coded-Function-Type` . . . 71
A-1.3.2: Units with a Makes Sense For slot of `Coder-Type` . . . 71
A-1.3.4: Units with a Makes Sense For slot of @T[Function-Type] . . . 72
A-1.3.4: Units with a Makes Sense For slot of `Implemented-Function-Type` . . . 72
A-1.5.2: Units not classifiable by `Makes-Sense-For` . . . 74
A-1.5: Units defined in Arlo: SOURCES; LISP . . . 74
A-1.5: Units defined in Arlo: SOURCES; TYPES . . . 74
A-1.5.2: Units without any prototype. . . . 74
A-1.5.2: Units with a prototype of **Coder** . . . 74
A-1.5.5: Units with a prototype of **Function Descriptor** . . . 75
A-1.5.5: Units with a prototype of **Hand Coded Function** . . . 75
A-1.5.5: Units with a prototype of **Slot** . . . 75
A-1.5.6: Units with a prototype of **Type** . . . 76
A-1.6: Units defined in Arlo: SOURCES; WHISTLES . . . 77
Chapter A-2: An Explanation 'Explanation' . . . 79
A-2.1: Units defined in Arlo: AI; DOCUMENT . . . 79
A-2.1.2: Units with a prototype of **Explanation Slot** . . . 79
A-2.1.2: Units with a prototype of **Hand Coded Function** . . . 79
A-2.2.2: Units with a prototype of **Slot** . . . 81
A-2.2: Units defined in Arlo: AI; EXPLAIN . . . 81
A-2.2.2: Units without any prototype. . . . 81
A-2.2.4: Units with a prototype of **Explanation Slot** . . . 82
A-2.2.4: Units with a prototype of **Explanation** . . . 82
A-2.2.4: Units with a prototype of **Hand Coded Function** . . . 82
A-2.2.6: Units with a prototype of **Slot** . . . 83
A-2.2.6: Units with a prototype of **Type** . . . 83
A-2.2.8: Units with a prototype of **Unit Explanation Slot** . . . 84

A-2.2.8: Units with a prototype of Unit Set Explanation Slot 84
A-2.3: Units defined in Arlo: SOURCES.LISP 85
Chapter A-3: References 86

Reading this thesis

The paper in your hands began as a primer for ARLO users, but with time it has — much like ARLO itself — grown, mutated, and gone through rearrangements. The bulk of this paper discusses representation language languages in general, and the detailed implementation of ARLO in particular. These discussions are followed by two examples presenting ARLO as both a system-building tool and as a framework for building auto-analytical systems.

The first chapter presents motivations for representing representations, and makes some first steps towards generally characterizing what is meant technically (as opposed to philosophically) – in the AI community – by "representation." It then introduces ARLO as a language for representing representation languages. The chapter closes with a scenario of a new user being slowly introduced to ARLO's functionality, features, and faclities.

The second chapter steps behind the scenes to talk about ARLO's internal construction, detailing how the mechanisms of the preceding scenario actually operate.

The third and fourth chapters of the thesis portray ARLO in two different roles. In the third chapter, a toy language for describing people and their interrelations is implemented; this language is then used to describe the members of an imaginary research lab. This embedded language and database is then examined and extended in an annotated script of a user's interaction with the system. This script illustrates ARLO's facilities for accessing, modifying, and extending its representations.

The fourth chapter presents an example of tools which examine representations and descriptions developed in ARLO or its extensions. It describes an *explanation system* which takes a collection of ARLO structures — describing either some domain, some representation language, or both — and produces an english description of the structures. The focus and organization of this description is generated from general properties of its topical structures extracted for the structures themselves. The explanation mechanism is then applied — as a demonstration — to automatically generate a description of the in-core implementation of the previous example (the laboratory database).

Finally, in the appendices, this explanation mechanism is applied to both itself (the explanation system) and the core of ARLO's default configuration.

This empty page was substituted for a blank page in the original document.

Chapter 1
ARLO: Representing Representations

In [Hof80], Hofstader makes the sweeping claim that *AI advances are language advances.* While this is certainly too broad a generalization, it has a hefty component of truth: we develop languages which reflect our developing theories so that we may actually bring those theories to the touchstone of implementation. As our proposals and theories advance and change, so do the languages — the abstractions and primitives — used to implement them. If we are really engaged in *experimental epistemology*, as some have characterized AI, then the languages and representations we develop are the burners, flasks, lasers, and spectrographs of our experimental laboratory.

But what precisely is an "AI language"? What distinguishes an AI language from a conventional programming language used to write intelligent programs? One distinction we might draw is that AI languages are languages embodying some *theory of AI programs.* The facilities which an AI language provides generally grow from observation of the sorts of things which AI programs — written in either conventional languages or other AI languages — tend to do: pattern matching, heuristic search, property inheritance, etc. A given AI language combines a collection of these extracted primitives with a few organizational principles — motivated both theoretically and technically — to provide a framework in which writing intelligent programs — encoding human knowledge and expertise — is easier and more elegant.

Yet in a deeper sense, an AI language does not merely provide a framework for expressing knowledge and expertise in convenient ways; it implicitly embodies some knowledge itself. The knowledge it embodies is the ontological foundation upon which programs or systems in it build: properties inherit in *this* way, two things are similar (match) by *this* criterion, logical inferences are invalidated when *this* happens, and so forth. In this sense, a given AI language is an AI program itself, embodying a particular theory of how a particular part of the world works. It's just that, in the case of representation languages, the parts of the world captured are techniques for representation and reasoning. But if an AI language is itself an AI program, might we build a language whose "domain" is these AI languages?

This paper describes ARLO, a **representation language language** which describes and implements representation languages, *including itself.* Descriptions of languages in ARLO are compiled into implementations, so that describing a given language in ARLO — in a sufficiently precise way — generates a reasonably efficient implementation of the language as well as a manipulable description of its semantics and behaviour.

The first RLL (*representation language language*) was developed at Stanford by Greiner and Lenat [Gre80]. Their implementation was dubbed RLL-1, and a version of it is used in Lenat's automated discovery program, Eurisko[Len83]. Eurisko uses an accumulated body of heuristics to guide the mutative evolution of representations and heuristics for various domains. A *reflexive* AI language — able to talk about and modify itself and languages embedded in it — is ideal for this sort of evolutionary development of concepts and expertise. ARLO was originally conceived as a Common Lisp [Ste84] version of the RLL-1, but has diverged from it in several important directions.

1.1 What Good is Representing Representations?

Why do we need — or want — a language for describing representation languages? Our programming languages already procedurally describe the representational mechanisms we use. What is the point of having an intermediate language for describing those mechanisms?

An answer to this challenge may be revealed by describing what an RLL offers two distinct classes of users: the expert systems developer and the AI researcher. To the expert systems programmer tailoring a representation for some understood domain, an RLL provides *systems programming tools* for developing a system's representational paradigms and primitives. To the AI researcher building a program whose understanding of its domain evolves through exploration and inner cogitation, an RLL makes a program's understanding of a domain into a manipulable stuff which the program itself can access.

1.1.1 RLL's as implementation languages

The tools which an RLL gives the implementor of an expert system are primarily system programming tools— tools which make the task of developing and debugging a given representation for a particular domain both faster and easier. The features that make an RLL a powerful development environment for expert reasoning systems are largely the same features that make modern LISP systems ideal for fast prototyping of any sort of complicated system. LISP is a powerful development environment because (among other factors):

- Programs and data are uniformly represented; the same tools used to describe and modify programs can be used to describe and modify data structures.
- Embedded Languages — building on LISP's data and program structures — can take advantage of already existing facilities of the LISP environment.
- The language can be dynamically and incrementally extended, as experiments with the implementational or theoretical feasibility of new ideas fail or succeed.

An RLL provides these same sort of features for higher level *representational* constructs:

- The description of a representation is accessible via the same mechanisms (in ARLO's case the accessing or modification of values in slots on structures) as the representation itself; indeed, these same mechanisms can be used to access and modify ARLO's description of the language ARLO!

- Tools built for describing, examining, or massaging a given representation can be easily generalized to other representations. System tools — used for describing, defining, and modifying representations — can be just as easily applied to the structures of the representation as to the representation itself.
- The definition of a representation — since it is stored as a mutable representation itself — can be dynamically modified. An RLL can — and ARLO generally does — perform the appropriate bookeeping to diminish unfortunate or fatal repercussions of such changes, while still supporting the intent of the change.

The implementation of a given representation or program takes a high level task or goal and reduces it to separately implementable parts. An RLL provides a tool kit and supply closet of such parts, where the interchangability of its representational components makes mechanism or experience from one application transportable to another.

1.1.2 RLL's as Mediums for Programs Which Grow

To the researcher developing intelligent systems which grow and learn by themselves, an RLL offers a way to let a program examine and extend its representation and understanding of a domain. The same properties of an RLL which support design efforts of human expert system programmers make simpler the design of *mechanical* expert system programmers which design and debug both other systems and themselves. While an RLL does not neccessarily embody any fundamentally new learning or problem solving technology, it does provide a framework for describing such technologies generically and reflectively (so that any sufficiently general mechanism can be effectively applied to its own description).

For any level in an RLL-based system — described problem, specified language, or the RLL itself — the same mechanisms for accessing and modifying its description are available. Due to this homogeneity of representation, faculties and tools built to operate on a given level may be applicable to other levels in the system as well. Lenat's Eurisko [Len83] system applies the discovery mechanism pioneered in AM [Len82] to such diverse domains as three dimensional VLSI design, space gaming fleet design, and number theory. But since the discovery mechanism — largely a collection of eclectic generation heuristics — operates in and is described in an RLL, it can be applied to itself, improving its behaviour with the accumulation of "experience" and examples in the application of heuristics.[1]

Analytic and descriptive tools developed in an RLL can generate summaries and descriptions of implemented or evolved systems which are useful to both human and mechanical programmers modifying or extending them. Further, since developments in the RLL are generally extensions or modifications of existing structure, descriptive and manipulative mechanisms and metaphors may be automatically extended to new applications in new domains. For instance, the grammar and dictionary of a natural language interface might be automatically extended to cover newly developed or assimilated concepts or relations, growing by extensions based on those concept's derived semantics. Such an extensible language interface would explain newly constructed or proposed concepts by using terms and grammatical forms extracted from the components which the concepts were developed from. In the same manner, the operations and presentations

[1] Unfortunately, Eurisko's experience is primarily embodied in the numeric *worths* assigned to its synthesized or *a priori* heuristics. We might wish for a more symbolic description of the systems failures and successes as the basis for this reflective modification.

offered by a graphical interface might be automatically extended to concepts and relations barely anticipated in the interface's conception. The interpretation of what some generic operation (characterized perhaps by a particular gesture or vocalization to the interface) on an object *should* do may be derived from the system's description of it and of that description's underlying semantics. An interface which represents what it is interfacing to regularizes the user's access to a changing program, making the implementation and debugging of self-developing systems — programs which grow — an easier task.

1.2 Representation as Inference

Before leaping into the question of how representations may be represented, we may wish to characterize exactly what we mean by **representation**. While we probably won't find — and perhaps don't desire — a complete definition, we would like some sort of intuitive grasp of what an RLL should — and shouldn't — try to represent. This section presents a characterization of representation as a special sort of inference, and briefly treats the consequences of this characterization.

1.2.1 Spontaneous and Deliberated Inference

In the beginning of this chapter, we slipped from describing AI programs to describing representation languages. But it would be hopelessly naive to claim that an AI program is merely its representation; what issues have we glossed over in sharpening of our focus to representation? Which part of what AI programs do is representation and which parts are something else?

Many, and perhaps most, of the actions of an AI program actively solving problems or operating in some particular domain (including itself) can be classed as *inferences* connecting one partial description of the world to an extension of that description. Such inferential actions further seem to fall into fairly distinct classes: **spontaneous inferences** and **deliberated inferences**. Spontaneous inferences are the sorts of inferences generally described with terms like *inheritance* or *defaulting*, while deliberated inferences are those inferences to which we apply terms like *hypothesis* or *counterexample*. This distinction between spontaneous and deliberated inference is one made by nearly all AI programs, but is it merely an implementational distinction, or is there a deeper semantic motivation behind it?

Certainly there is no such distinction implicit in the product of such inferences; spontaneous and deliberated inferences seem to share an identical character of belief or rational integration. The difference instead lies in the act of inference itself, in the character of the *action* by which we extend our representations of the world. If we characterize inferences as mental actions, spontaneous inferences might be looked at as the *basic actions* of a rational agent making inferences. This introspective notion places a *psychological*, rather than semantic, motivation behind the distinction between spontaneous and deliberated inferences. While a formal analysis of any given inference system may best treat the two sorts of inference identically, any implementation or psychological theory must retain the division.

1.2.2 Characterizing Spontaneous Inference

Are there other characteristics — besides the intuitive and psychological characterizations presented above — for the distinction between spontaneous and deliberated inferences? One important clarification is that the distinction is not identical to the psychological distinction between conscious and unconscious mental

activities. Conscious and unconscious activities each involve both sorts of inference; conscious reasoning might be those deliberated inferences which are verbalizable, but even this may be going too far. In any event, spontaneous inference is not psychologically subconscious activity; the knowledge that Clyde the elephant is gray, while a spontaneous inference, is certainly not a "subconscious" inference.

One distinction which we might make between these two sorts of inference is that *spontaneous inference never builds large mental structures.* If we believe that there are aggregated collections of ideas in the mind — structures — then spontaneous inferences may complete, fullfill, or verify these structures, but will never construct them *ex nihilo.* Deliberated inference, on the other hand, has no such restriction; indeed, a large part of its operation is the construction of just such loci of assumption and assertion.

Our names for the two sorts of inferences suggest another distinction which we should clarify. The adjective *spontaneous* suggests that such inferences happen quickly, while *deliberated* suggests a more extended process. While this is largely the intuition intended, we need to make clear when this inferential interval actually occurs. Both spontaneous and deliberated inferences occur *on demand*; this demand may be of physical neccessity or psychological intention, but the terms spontaneous and deliberated characterize these inferences as *actions* carried out on demand, rather than as valid possibilities of action in a represented world. Spontaneous — in our usage — does not mean that when I tell you Clyde is an Elephant you automatically infer that he is gray. It does mean that if I ask you what color Clyde is you can tell me quickly, without needing to go through any complicated intellectual machinations.

1.2.3 The Evolution of Spontaneous Inferences

Does deliberated inference become spontaneous over time? Is the distinction the same as the mechanomorphic distinction proposed between "compiled" and "interpreted" knowledge? Without a more precise definition of such "compilation", it is hard to decide this latter question one way or another, but I suspect the answer will be no. The process of compilation — as generally described in modern computer science — allows programs to run faster by removing general character and making assumptions of context and reference. Spontaneous inference does not rest on local assumptions of context or reference, as its execution may reach across expanses of representation and structure. Taking the point about mental structures offered above, spontaneous inferences are "fast" because they do not need to generate or access intermediate structure created on the fly.

On the other hand, the answer to the first question above — about deliberated inference becoming spontaneous — is probably affirmative. The way deliberated inference becomes spontaneous involves the *change of representation*; deliberated inference in new domains works with structures formally representing the domain and its principles — the manipulative principles of algebra or the force-motion axioms of hitting baseballs; with time, however, the representation for the problem becomes specialized, as individual objects and subproblems are placed in broad and powerful classes. It is by this process that deliberated inference — referring to objects and general rules — becomse spontaneous inference — referring to properties and particular paradigms.

Finally returning to the topic at hand, an RLL is a lanaguage for describing and implementing spontaneous inference systems. The classification of inferences as spontaneous does not exempt them from the restrictions we typically place on inferences. We still can demand consistency, accountability, and robust

non-monotonicity. As such it must offer facilities for maintaining what we demand from inferences with a minumum cost in their execution.

1.3 What is ARLO?

As suggested in the previous sections, ARLO embodies a theory of how representation languages work. But like any theory tied to an implementation, ARLO's carries its baggage of prejudices, leanings, and restrictions. Most obviously, ARLO is prejudiced by an initial configuration as a *frame-based* language. As initially configured, ARLO is a language for constructing data-structures — objects with properties — which describe the operation and performance of other representation languages also based largely on data-structures. In turn, ARLO is itself described by its own data-structures and this description is referred to in compiling and interpreting the descriptions of representations described in ARLO. ARLO's compilation and description of a given representation references the in-core structures which define ARLO itself, rather than being hardwired as LISP procedures. In compiling a given representation, ARLO is partially interpreting its own description of itself.

How can such a self-referential interpretation process ever run efficiently? ARLO compiles the representations it describes — including itself — into LISP code cached in quickly accessible locations in the language's description. This caching of values allows ARLO and representations described in it to run efficiently once compiled. A *value dependency mechanism*[2] assures the accuracy of ARLO's cached compilations by updating or retracting them when the descriptions from which they were compiled are changed. Because of this bookkeeping, representations described in ARLO — including ARLO itself — can be dynamically modified with relative impunity.

Greiner and Lenat's RLL-1 is cast as a representational "organ," whose stops and settings can be modified by a performer or user, mutating RLL-1 into a language with some set of particularly desired features. ARLO, while supporting this sort of fundamental mutation by providing access to its representation of itself, is primarily designed to support *extension* into new representational paradigms, without supplanting its basic core. Instead of an organ, ARLO might better be perceived as a factory of synthesizer components and patches, from which a user constructs whatever representational tools or paradigms she will.

The ability to mutate ARLO and languages described in it means that, in some ultimate sense, ARLO is not really restricted by its intial configuration; ARLO could be used to define another RLL based on assertions rather than frame-like data-structures. Such a representation, however, might not be acceptably efficient because of the way ARLO compiles its descriptions: the way a frame system is compiled and optimized is very different from the way an assertion based language would be compiled and optimized. ARLO could be radically mutated to do such optimizations, but I certainly don't claim to have done this, and in some strong sense such an accomplishment would be a wholly different language.

[2] A value dependency mechanism is a generalization of propositional dependency mechanisms like [Doy77] or [McA78]. It keeps track of what elements of the environment a given environmental side effect depends on, updating or undoing that side effect when those elements are changed. A propositional TMS is a specialized sort of value dependency system which performs this maintenance function over just the truth values of rules and propositions.

1.4 Basic Concepts: A User's Introduction

ARLO is a frame based language.

A new user approaching ARLO in its initial configuration finds a classical "frame based" language much like FRL [RG77] or UNITS [Ste79]. In this language, she may create, examine and manipulate data structures — called **units** — which possess properties — called **slots** — to which are attached **values** — which are lisp objects of various sorts. Each ARLO unit has a unique name relative to some *knowledge base* (a namespace grouping many related units together), and its slots map symbolic names — each again relative to some particular knowledge base — to single values. As in other frame based languages, the value of a slot is sometimes computed on demand; a slot's attempted retrieval may compute a value (a default) for the slot if one is not already available.

Defaulted values are cached and justified.

When the value of a slot of some ARLO unit is defaulted, the newly computed default is saved — *cached* — on the unit itself. This caching allows subsequent references to the slot to return a value immediately, without having to recompute a default value. Each of these cached values also records the *justifications* of its derivation: the function used to compute it and the slots referenced in its computation. When the user later changes one of these supporting justifications, she finds that the cached default — typically listed when the unit is described to her — disappears. When she asks again for its value, a new up-to-date default is computed, and once more cached on the unit. The justifications of each of these defaulted slots are explicitly available to the user: when she asks for a description of some particular slot's value, its justifications are listed along with the description of its value.

Different slots have different semantics.

From the justifications ascribed to various slots, our user discovers that different slots derive their defaults in different ways. For instance, she finds that the `Telephone-Number` slot of a person-description defaults through its `Organization` slot, while the description's `Home-Address` slot defaults through the `Spouse-Equivalent` slot attached to it. Further, in the process of creating and modifying various units, she finds that certain slots will accept only certain types of values and will attach to only certain kinds of units. The `Supervisor` slot of a person-description — for instance — accepts only other person-descriptions (determined by some inheritance criterion in some hierarchy) for its values and attachments.[3] When she accesses ARLO's descriptions directly from LISP (using a small repetoire of top level functions for accessing and storing values in slots) the user discovers that the way in which slot values are printed and described also varies from slot to slot; a `Birthdate` slot may store its value as a number in seconds since 1900, but this value is always printed out in a more human-palatable form. Different slots in ARLO, she concludes, have different semantics: different sensible values and attachments, different mechanisms for defaulting, different methods for describing their values, etc.

These semantics are explicitly described in ARLO.

[3]The *attachment* of a slot is the *unit* it attaches its value to. For the `Home-Address` slot of the unit `Kris-Kringle`, its attachment is the unit `Kris-Kringle`, and its value is "North Pole, Earth"

When at some point our user wishes to know more about the semantics of a particular slot, ARLO reveals its accessible underside. To get a description of a particular sort of slot, she need only examine an ARLO unit describing the slot to see a summary of its intentions, mechanisms, and restrictions as specified by its human or mechanical implementor. *Each slot in ARLO*, she discovers, *is described by an ARLO unit.* For example, if she wants to use a "color" slot defined by some other user, she can describe the the slot-defining unit named `COLOR` to see its complete specification. This `COLOR` unit details many aspects of the "color" slot: what types of objects can be stored as colors, which sorts of units may have colors attached to them, how a color should be described to a user or even precompiled problem solving "cliches" for discovering or changing the color of an object.

Modifying this description can alter the semantics of the language.

But these descriptions are not merely one-way windows on the semantics of the language; if the user is dissatisfied with some part of the definition of the slot, she can modify the ARLO unit describing it and that its semantics have changed. For instance, having the definition of the color slot "in hand," she can extend or modify different aspects of its semantics — such as how it is defaulted, restricted, or described — by using established and familiar functions and utilities for modifying ARLO units.

ARLO represents implementation as well as semantics.

The ARLO description of a slot specifies not only its semantics — its restrictions and assumptions — it also specifies its *implementation.* Since the methods for storing or fetching the value of a slot are explicitly described in ARLO, different slots may be implemented in different ways. For instance, some slots might store their values in a high speed "connection memory" [Hil85], while others might store their contents on a shared storage device across a local network. While the initial ARLO configuration uses only immediate storage techniques (storing values directly on the unit data structure), this in no way limits its ultimate configuration or organization.

ARLO also represents its own semantics and implementation.

ARLO represents not only other representation systems, it also represents *itself.* The slots and units used to describe the semantics of a given representation are themselves described in ARLO. This means that the unit describing the `To-Verify-Type` slot [4] has a `To-Verify-Type` slot which is referred to whenever a `To-Verify-Type` property is defined for a slot.

ARLO's self-representation is made possible by an elaborate and circumventive bootstrapping process that occurs when it is compiled and loaded. In this process, slot-describing slots — such as `To-Get-Value` or `To-Verify-Type` — are defined as units with preemptively stored `To-Get-Value` or `To-Verify-Type` slots referenced by run-time ARLO. ARLO's bootstrapping process sneaks around the self-referential interpretation mechanism to prepare a pre-compiled runtime environment which refers to itself in compiling and interpreting other representation languages, including the remainder of itself.

The ability of ARLO to easily modify itself allows introspective activities like self-modification, self-

[4] The `To-Verify-Type` slot stores the function which a slot uses to determine if a given value and attached unit are acceptable.

documentation,[5] or self-explanation to be performed with ARLO structures. Not only may a program written in ARLO examine or modify its own representation language, it may examine, extend, and modify (within limits) the language in which that representation language is described.

[5]For instance, the documentation in the appendices was produced by ARLO examining and describing its own description.

Chapter 2
ARLO's Implementation

Most AI languages are implementation towers; it is popular to diagram the construction of a given AI program as a tiered construction of implementation layers resting on a foundation of vanilla LISP. (Occasionaly some clever wag also sketches in the machine language, microcode, logic circuitry, and semiconductor physics beneath this LISP foundation.) Figure 2-1 is such a diagram for ARLO's construction, illustrating the foundational role each level plays in the next. This chapter describes these components of ARLO's implementation and the boostrapping process which consolidates them into a working self-referential implementation.

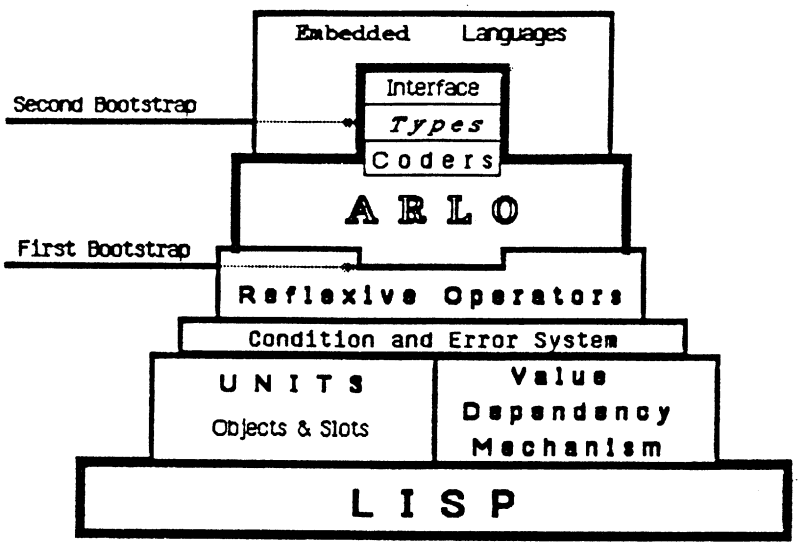


Figure 2-1. The Layered Implementation of ARLO

But Figure 2-1 is not quite the typical "layers of implementation" diagram; its details offer more content than simply illustrating levels of embedded languages. The horizontal arrows on the figure indicate two important phases of ARLO's deployment; each corresponds to the *boostrapping* of some component of ARLO's self-representation. The first bootstrap is the definition of ARLO as a representation defining language. The second boostrap is the completion of ARLO's type restriction system, which constrains the values and attachments of various slots.

As intimated above, a language implemented in ARLO remains reasonably efficent by caching its compiled implementation on quickly accessible properties of its description. We might view this compilation process as pushing ARLO's execution down the tower of Figure 2-1. While a given representation is described at the level just above ARLO's definition, it is implemented and executed at the more efficient levels below it.

The tower in Figure 2-1 has 11 distinct components, each of which plays a foundational role in the components above it:

1. **The LISP underpinning**
   ARLO is implemented in LISP Machine LISP [WM82] for a variety of special purpose LISP Machines. The version of ARLO described here is ARLO Version 25.30, running in Symbolics Release 5.2. ARLO uses a variety of facilities developed for the LISP Machine, providing (among other capacities) special capabilities for formatted output and "impatient i/o".
2. **UNITS: A Data Structure Facility**
   LISP is used to implement a data-structure facility for creating and accessing named objects with named properties. These structures — called **units** after RLL — are implemented as fixed-length hash tables which pair symbolic names to single values (which may of course describe sets of values). The names of units and slots are organized by a namespace system which divides units into **knowledge bases**; particularly, a knowledge base provides a many-to-one mapping from symbolic names to unit structures.
3. **The Value Dependency Mechanism**
   Also implemented in LISP — or precisely, in Lisp Machine *flavors* — is a *value dependency mechanism* for keeping track of dependencies between various properties and bindings of the LISP environment, particularly the values assigned to the slots of ARLO units. This mechanism is used by a deployed ARLO to keep track of its changing defaults as well as its changing semantic definition. The value dependency mechanism is described in Section 2.2.
4. **The Error Facility**
   No large system is perfectly bug-free, and ARLO's self-referential implementation makes catching and dealing with its internal problems a tricky task. Tracking and repairing an internal ARLO bug is often like trying to climb out of a sand pit; each exploratory modification may shift or shatter the foundations beneath you. Despite this, ARLO retains a degree of robustness through two mechanisms: the first is the value dependency mechanism which ensures that changes in mechanisms described in ARLO from component to component in the implementation; the second is a rich taxonomy of errors and conditions which are signalled when ARLO detects itself going wrong. These errors describe conditions such as obviously fatal recursions, type conflicts, or violations of bootstrap requirements. ARLO's facilities for handling and signalling these unusual conditions is described in Section 2.3.

5. **Reflexive Operators**

– ARLO's self-reference is centrally embodied in a mechanism called *reflexive operators*. Reflexive operators refer to ARLO unit structure to determine how to operate on and access other unit structures. When the description of an embedded language (or of ARLO itself) is compiled, it is assembled into a set of units whose interpretation by reflexive operators fullfills the intended semantics of the langauge's description. Reflexive operators are an interpreter for frame like data-structure languages; the ARLO language itself (interpreted by these mechanisms) is a compiler for turning high level representation descriptions into structures for this interpretation process.

6. **ARLO's Definition**

These are the units which define ARLO's core, specifying a language — interpreted by reflexive operators — which describes how the slots of a frame-based representation language default, restrict, and describe. The detail of Figure 2-1 illustrates how the definition of these central units, skirting ARLO's self-reference mechanism, extends below the level of reflexive operators at ARLO's first boostrap. The essentials of ARLO's definition — how it describes and defines the slots of various representations — are documented in Section 2.5.

7. **The ARLO coder**

ARLO's ability to define representation languages is used immediately in implementing ARLO's coder mechanism, specifying a language for describing the implementation of LISP functions. ARLO's *coders* expand partial descriptions of common representation functions (inheritance, composition, type restriction, etc) into completely specified LISP implementations. These tools for function-building are detailed in Section 2.6.

8. **The TYPE system**

On top of the coder mechanism, ARLO's *type system* is implemented. The type system implements a non-excepting generalization hierarchy for predicates; these are used to specify restrictions on the attachments and values of slots defined in ARLO. ARLO's own initial description (which is used to implement this hierarchy of types) refers to the type system by referencing the names of particular types. The bootstrapping of the type system (the second dotted line on Figure 2-1) maps over every unit in ARLO's description of itself and replaces all of its symbolic type *names* with now-assembled type *descriptions*. ARLO's utility package extends the type system into a *class system* for organizing units into overlapping description categories to which particular methods and heuristics are attached. The basic form of the type system is detailed in Section 2.7.

9. **Archives and Layers**

A representation language language allows a complicated program and representation to be extended (or to extend itself) over time; but if the program must be rebooted and restarted each morning, its scope is limited by its short lifetime. Archives and layers are a mechanism for wholly and incrementally dumping ARLO representations and descriptions. The knowledge of a sophisticated program is a dynamic and interconnected network of descriptions; archives and layers are tools for preserving those networks between sessions and even (if any projects are sharing particular representational tools) between domains. The implementation of archives and layers is documented in Section 2.8.

10. **The User Interface**

In the previous chapter, one of our arguments for the utility of RLLs was the expressive flexibility they might bring to a user interface. ARLO's user interface explicitly accesses and refers to the semantic description of the descriptions it is presenting, offering different displays and options based on the underlying definition of what it is describing. ARLO's interface — operating through a variety of "interface modes" — determines its presentations and presumptions by its own description of the concepts and relations it is presenting.

11. **Embedded Languages**
Languages embedded in ARLO are finally built on the top of this edifice, taking advantage of the descriptive and debugging facilities beneath them. Many representations built in ARLO (including extensions of ARLO itself) do not build very high over the mechanisms which ARLO natively uses to describe representation languages. These mechanisms — simple property inheritance, single hierarchy type restrictions, etc — may be all a user needs for her application; but on the other hand, she may easily implement more complicated representational constructs and abstractions at need.

2.1 Units and Knowledge Bases

Units are LISP structures which map named properties to LISP objects. Implemented as fixed length hash tables, they can be thought of as a fast implementation of property lists. The implementation of units imposes no semantic restrictions on what may be represented, outside of presuming that their exist objects with named properties. The semantics of ARLO comes from the interpretation of descriptions constructed from these units, much as the semantics of LISP comes — in a sometimes illusory sense — from the interpretation of list structures. ARLO's units — like LISP's global function and variable definitions — are more or less global definitions, but they are organized into many separate distinct knowledge bases.

Each ARLO unit has a name and is attached to a particular *knowledge base*, which is a structure containing a collection of related units.[6] Within this knowledge base, the unit's name is unique, though it may possess other *aliases* in the same or different knowledge bases. To support this, each knowledge base provides a many to one mapping from names to units; but for each unit, one of these mappings is it's unique *true-name* used (by default) in printing and archiving it.

A unit's printed representation looks like this:

`{#>EXPLAIN:SUB-DIVISIONS}`

where `SUB-DIVISIONS` is the name of a unit in the `EXPLAIN` knowledge base. A user refers to a unit in a given knowledge base by using the lisp reader macro "`#>`". For example, the expression `{#>EXPLAIN:SUB-DIVISIONS}` refers to the unit whose printed representation appeared above.

ARLO's knowledge bases are arranged in a hierarchy from the root `CORE` knowledge base, as pictured in Figure 2-2. All units defined in a knowledge base are also defined in the knowledge bases below it. For instance, every knowledge base contains the units of the `CORE` knowledge base; similarly, all of the units defined in the `EXPLAIN` knowledge base will be defined in the `THESIS` knowledge base beneath it. Knowledge bases are a namespace mechanism and not a real "representational context" mechanism; user code cannot easily refer to "X in the current context," but only to "X in the EXPLAIN context".

[6] Knowledge bases are implemented on top of the Common LISP *package* system, a facility for maintaining multiple namespaces in a single LISP environment.

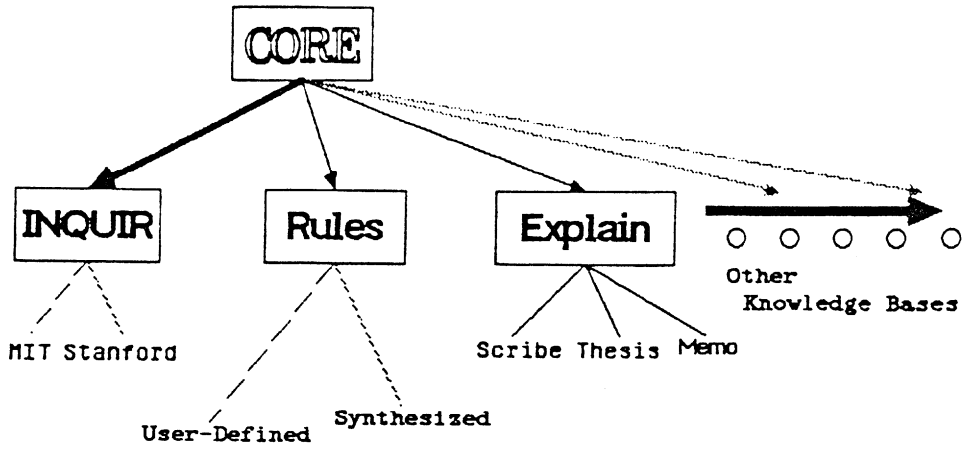


Figure 2-2. ARLO's knowledge bases are organized into a hierarchy of name inheritance.

2.2 The Value Dependency Mechanism

ARLO's slots are interconnected with a value dependency mechanism. When the value of a slot is defaulted and cached, a **dependency-record** for the value is created referring to the dependency records of the values accessed in computing the cached default. Each of these referenced dependency records is also given an inverse pointer to the newly created dependency record. Later, if one of these referenced dependencies — an "assumption" the cached default depends on — is invalidated, the dependency record for the cached value is also invalidated. This invalidation *decaches* the out of date default, removing it from the unit structure on which it was cached. The next attempt to access the value will then — in the absence of a cached value — be forced to recompute a valid value for the slot.

The tracking of a slot's dependencies is quite simple. When a slot is being defaulted, the global variable `SLOT-BEING-DEFAULTED` is bound to a dependency record for the slot being defaulted. Every slot access occuring during the computation of this default calls the form:

(`ASSUMING` *unit slot*)

to make the dependency record kept in `SLOT-BEING-DEFAULTED` dependent on the current *slot* of *unit*. This dependency tracking may be disabled by the macro form `AS-A-SIDE-EFFECT`, which binds `SLOT-BEING-DEFAULTED` to `NIL` for the dynamic scope of its body, thus protecting any default computations in progress from dependence on slots accessed in execution of its body. In addition, the call to `ASSUMING` is part of each slot's description, so individual slots might be defined to not reference the dependency creating form.

Dependency records for slots are stored in a non-reflective network (i.e. simply as named properties of unit structures) defined in special knowledge bases associated with the knowledge base of the slot's whose values they describe. For instance, the dependencies for the `#>CORE:To-Default-Value` slot are stored on

the #>CORE-DEPENDENCIES:To-Default-Value property (*not* slot) [7] of the unit whose #>CORE:To-Default-Value slot they describe. A given slot's dependency record may be accessed by the form:

(get-dependency-record *unit slot*)

which gets the dependency record describing the current value of *unit*'s *slot*. These dependency records are implemented as *flavor* instances [Can83] which accept messages defining an invalidation, justification, and description protocol.

2.2.1 Dependency Mechanism Protocols

ARLO's value dependency mechanism uses the message passing facility of flavors to define a protocol for the propogation of slot-value invalidation. In addition to this role, other protocols define ways of recording justifications (which may later lead to invalidations) and documenting or describing the supports of an assumed or deposited value. These protocols, however, never refer to slots or units in particular and is easily extended beyond this; while most of the nodes in the dependency network describe the values of slots, many do not. Some, for instance, describe value or function bindings in the LISP environment; others play critical roles in the presentation — to the user — of the slot network.

In particular, several graphical interfaces to ARLO have the visual presentations of ARLO slot bindings "wired into" the dependency network running between slots; the appearance of a given presentation in the interface then changes with the validity of the slot value it represents. The graphical representation — a flavor object — is defined to handle the invalidation protocol for dependency records and then connected into the active dependency network just like any other node.

The invalidation and justification protocol is defined by six messages which are sent to and passed among nodes in the dependency network:

- :INVALIDATE-SELF invalidates a given dependency record and the dependency records which depend on it. This is generally sent by an outside function, rather than from one dependency record to another.
- :RETRACT-DEPENDENTS invalidates the dependents of a given dependency record. It does this by sending all of its dependents a :SUPPORT-RETRACTED message (with itself as an argument), normally causing the dependent value to be undone and spinning off another wave of :SUPPORT-RETRACTED messages.
- :SUPPORT-RETRACTED is sent to a dependency record when one of the dependency records it depends on is invalidated. The response of a dependency record to this message will typically go and alter the value or assignment to which the dependency record refers. (This in turn will typically invalidate the node receiving the message, and spin off new :RETRACT-DEPENDENTS and :SUPPORT-RETRACTED messages.)
- :ADD-DEPENDENT adds a dependency record (its single argument) to the records *depended on* by the record this message is sent to.
- :REMOVE-DEPENDENT removes a dependency record (its single argument) from the records depended on by the record this message is sent to.
- :REPLACE-SELF replaces the dependency record it is sent to with a new dependency record (its single argument). In order to side effect its value, the dependency record which receives this message should know where the value it refers to is stored.

[7] The distinction between a 'property' and a 'slot' is that a slot has an abstract description specifying the interpretation and semantics of its values and a 'property' is simply a named attribution to a unit.

Dependency records also support a collection of messages for debugging and explanation of the values they represent. There are four basic messages for describing dependency records:

- :DESCRIBE-CONTENT describes the value its record represents. This is used by all the descripton functions. This description is sent to the stream which is the messages single argument.
- :DESCRIBE-HISTORY describes the record it is sent to, as well as the record that record replaced, thus producing a history of the value the dependency describes. It takes a stream as a single argument, as above.
- :DESCRIBE-DEPENDENTS describes the other dependency records which depend on this dependency record. It takes a stream as a single argument, as above.
- :DESCRIBE-JUSTIFICATION describes where this value came from. If it was deposited by some person, computed from some other slots, etc.

In the development of this protocol, it was neccessary to overcome the confusion of having two distinct networks: the unit-slot network and the dependency network. Early versions of the protocol did all propogation of invalidation through the dependency network, causing numerous problems with slots which wished to avoid or affect their invalidation in various ways. The final solution was the separation of the :SUPPORT-RETRACTED and :RETRACT-DEPENDENTS messages by reference to the unit-slot network. This barrier finally allowed the dependency mechanism to avoid enforcing certain semantics on the unit-slot network.

ARLO's initial configuration defines three basic sorts of dependency records: `Slot-Computation-Records`, `Slot-Citation-Records`, and `Slot-Boot-Records`. `Slot-Computation-Records` are records of slot computations (such as the computation of a default) which referred to other slots and can be invalidated by the invalidation of those slots' values. `Slot-Citation-Records` are dependency records which refer to a particular *source* and *attribution* responsible for them. Typically these are references to users or text files. `Slot-Boot-Records` describe slots defined before ARLO's critical bootstrap period; attempting to invalidate these records results in a proceedable error. This warning sometimes avoids fatal self-modification by programs in ARLO or unsuspecting users.

2.3 ARLO Errors and Conditions

ARLO uses the lisp machines' *condition system* [Wei83] to define a taxonomy of conditions with which it complains when it comes across unexpected or unusual situations. These conditions include both external conditions (such as a particular user or machine not responding to requests) and internal conditions (such as fatal recursions or type conflicts). Code using ARLO may anticipate and catch these conditions and there is a standard facility — an ARLO *coder* — for doing this preemptive preparation. Futher, these conditions are defined so as to offer standard ways to proceed from various situations as well as providing pertinent information to the user when she is asked to handle the condition (typically by entrance to the LISP Machine debugger).

In order to handle and report errors effectively, ARLO keeps track of various parts of its dynamic state. For instance, the current *access state* (the unit operations currently in progress) is always available to

the program in the variable `SLOT-ACCESSES`. ARLO uses this dynamic record for, among other purposes, identifying when it is fatally recursing on some slot access. [8]

The function `where` can be used to look at this part of ARLO's current dynamic state: It produces a trace that looks like this:

```
ARLO is currently:
4:   trying to compute the Supervisor slot of {#>KYLE}
3:    while getting the Supervisor slot of {#>KYLE}
2:     while trying to compute the Hacking slot of {#>KYLE}
1:      while getting the Hacking slot of {#>KYLE}
```

If you use the debuggers Control-M command to send a bug report on an ARLO condition, a version of the above trace is included in the bug message. In addition, you can type the keystroke command Control-? to get a `where` trace while in the debugger.

2.3.1 Anticipating errors

ARLO's errors signal not only unexpected conditions — such as type conflicts arising from sloppy generated or user code — but also "unfortunate" conditions such as failed searches or absent users. For both of these, the program itself might want to take action when the error occurs. In the case of unexpected conditions (what we might call true errors), the program might wish to repair or banish a definition or description; in the case of unfortunate conditions, the program might wish to apply another method or simply assume a harmless default. Harnessing the Lisp Machine's condition handling system, ARLO is able to answer both of these demands.

Unfortunate conditions are generally conditions of failed methods, for which there are alternative responses or actions. In ARLO, unfortunate conditions are handled by "try and try again" functions, which possess many distinct methods for performing their computation, moving from one to the next if an error occurs. These functions are typically synthesized by ARLO *coders* 2.6 such as the `METHODS` or `EXPECTING` coders 2.7. When errors occur when these functions are executing, they throw out of the erring method and advance to another or signally a final error if no more methods exist.

Unexpected conditions, on the other hand, generally arise from ill-formed programs or descriptions; their occurence generally demands the alteration or generation of relatively large programs or descriptions. As such, the reaction to such errors falls into the class of operations which we identified in 1.2.1 as *deliberated inferences*. Here we perceive a powerful pattern to the interaction of spontaneous and deliberated inferences: deliberated inferences arise from the failure of spontaneous inferences. It is only when our cached, compiled, and common methods fail that we turn to the carefully constructive process of deliberation in our problem solving. We must at least — if we wish to build mind-like systems with ARLO — provide explicit classes of these unexpected conditions which reveal precisely how the languages definition and description have been strained.

[8] If ARLO is about to perform a slot access, it first checks that it is not already (further up the stack) performing it— if it is, it signals a `Fatal-Recursion` condition which may either be caught by ARLO's "expectations" or reported to the user.

2.3.2 Classes of Errors

A newly loaded ARLO defines a small collection of special conditions. As ARLO (and programs using it) venture into new domains, new techniques and new methodologies, this collection of conditions should grow to become both more "worldly" and more tightly connected to the structure of ARLO.

All ARLO conditions inherit from the condition flavor `ARLO-CONDITION`. Currently, the following conditions are defined in a newly-loaded ARLO:

- `Fatal-Recursion` is signalled when ARLO notices that it is trying to perform some operation which is already being attempted. The user is offered the option to try the operation using non-reflexive sub-primitives, or she may use the standard debugger commands to re-evaluate or return a value from the fatally recursive call.
- `Slot-Not-Found` is signalled when an attempt to inherit some slot through some relation fails— often this error does not reach the user, but is caught and handled by ARLO itself. If it does reach the user, she can proceed by either providing a value to cache locally, trying to inherit through another slot, or looking on another unit for the value.
- `Unacceptable-Value` is signalled when a value being stored on a slot is of the wrong type for that sort of slot. If she wishes, the user may tell ARLO to go ahead and store the value anyway.
- `Unacceptable-Unit` is signalled when a slot is being attached to a unit of the wrong type. As with `Unacceptable-Value`, the user may tell ARLO to go ahead and store the value anyway. The abstract condition flavor underlies both the `Unacceptable-Value` and `Unacceptable-Unit` condition flavors.
- `Boot-Conflict` is signalled when a slot which was defined before ARLO's second boostrap is being invalidated. This will typically happen when a new value is being placed there. Going through with such a replacement might cause a problem because such a slot may — in its boostrapped configuration — *implicitly* depend on itself. E.G. ARLO may have to reference the slot being invalidated in order to finish retracting it or put a value in it. While ARLO is generally robust about changes whose dependencies are explicit (and thus non-circular), all bets are off for pre-bootstrap definitions which ground ARLO's self-description.
- `Cant-Default-Slot` is signalled when the value of a slot cannot be defaulted; this might happen if the slot was never intended to default, or if all known methods for defaulting the slot have failed. Often this may be caught by a prepared handler which then deposits its own "default" as a replacement value.
- `Out-Of-Methods` is signalled when a try-and-try-again function [9] runs out of methods to try in computing some value. The user can proceed from this by providing either a value to use as a result or another method to try. When this condition is reported to the user, it describes the methods it has already tried in computing the value. Often this error is caught by higher level try-and-try-again functions which move on to other higher-level approaches when this is signalled.

[9] A try-and-try-again function tries one method after another to compute a value, moving onto the next one if the previous fails. ARLO supports two sorts of try-and-try-again functions: one moves onto the next method only if the current method fails in some "expected way"; the other is a blanket version of the first, that tries the next method when any sort of error occurs.

- `User-Not-Found` is signalled when a query to the user times out. This should be connected to ARLO in a more intimate way, using a user model to decide when to quit, and being able to figure other methods of contacting the user. (Such a model should also clearly incorporate some theory of etiquette!)

2.4 Reflexive Operators

ARLO's operation refers to the in-core description of its own semantics in such a way that when its description is modified, its performance changes. This is done via a data-directed mechanism called *reflexive operators*. [10] Reflexive operators are functions of the form:

```
(<operator> <unit> <slot> . <remaining-arguments>)
```

(where `<operator>` is a reflexive operator) and working by applying the `To-<operator>` slot (a slot also defined in ARLO) of `<slot>` to the arguments `<unit>`, `<slot>`, and `<remaining-arguments>`. For example, the form:

```
(Put-Value #>Jane #>Age 25.)
```

takes the result of `(Get-Value #>Age #>To-Put-Value)` and applies it to the unit named `Jane`, the unit named `Age`, and the number 25. This application might then (for instance) verify the suitability of 25 as the value for Jane's age or perform various dependency maintenance functions in addition to — or perhaps in place of [11] — actually depositing the value.

In the same manner, the form:

```
(Retract-Value #>Jane #>Age)
```

works by taking the result of `(Get-Value #>Age #>To-Retract-Value)` and applying it to the units named `Jane` and `Age`. This will then — typically — retract the value on the `Age` slot of the unit named `Jane`.

2.4.1 Staunching an infinite regress

The one significant exception to the reflexive operator mechanism is the `Get-Value` function. The mechanism described above runs into a snag when we try to define `Get-Value` as a reflexive operator; we would like `Get-Value` to work like any other reflexive operator, evaluating:

```
(Get-Value <slot> #>To-Get-Value)
```

to get an appropriate accessor, and applying this accessor to `<unit>` and `<slot>` to get a result. Unfortunately, this approach ends up infintely recursing [12] on:

```
(Get-Value #>To-Get-Value #>To-Get-Value)
```

To get around this problem, `Get-Value` is only partially reflexive: instead of calling `Get-Value` to find a `To-Get-Value` slot, it checks `<slot>` and its *prototypes* — a relation defined as part of ARLO's initial configuration — for a `To-Get-Value` slot. A slightly cleaner version of this might look at the `To-Get-Value` data structure itself for the function to use in its search, rather than using a hard-wired definition.

2.5 Representing Representations: The Details

[10] This terminology originates with ARLO.

[11] If the value being deposited were inappropriate by some criterion, it might signal an error instead of depositing the value.

[12] ARLO usually catches such fatal recursions and signals an error condition.

The reflexive operator mechanism is an *interpreter* for structures specifying the implementation of frame-based languages. From a partial description of a given representation language, ARLO generates — by inheritance from abstract specifications and the synthesis of standard representation functions — the precise details of its implementation. ARLO's basic definition specifies the components of this generation process: inheritance mechanisms, automatic coders, descriptive constraint predicates, etc. These primitive mechanisms for language definition are themselves described in ARLO's "pre-configured" representation and are interpreted by reflexive operators in specifying and compiling other representations. The primitive definition of this core can thus be extended or changed — carefully! — to alter or expand the capabilities of the language.

ARLO's central core is bootstrapped by setting up an initial description — to be interpreted by reflexive operators — for a simple representation language. Facilities like coders and more complicated representation compilers are then described (and executed) in this representation langauge.

In ARLO's central core language, the primary inheritance mechanism — the mechanism by which properties are declared abstractly and then propogated to particulars — is `Prototype` inheritance. This sort of inheritance generates defaults for values by searching along the `Prototype` relations between units. The `Prototype` hierarchy is an exception-shadowing hierarchy of slot inheritance which keeps dependencies for its inherited and cached values. While representation facilities built in ARLO define and use other sorts of inheritance mechanisms, ARLO itself goes little beyond this simple mechanism.

When a user begins building a representation in ARLO, she generally uses the `Prototype` relation to refer to a collection of pre-defined abstract slot descriptions, from which the particulars of ARLO and its embedded representations inherit. A newly bootstrapped ARLO has a small collection of these prototypical slots, defining simple classes of relations whose implementation details inherit through the `Prototype` hierarchy; extensions to ARLO may well define entirely new such classes of slots beyond these.

The most basic sort of slot is the `Primitive-Slot`; `Primitive-Slot` is a non-defaulting, non-restrictive slot, and lies at the root of the `Prototype` hierarchy of slots. The `Prototype` relation is a primitive slot, but most other slots lie deeper in the slot inheritance hierarchy (the `Prototype` hierarchy) than this. The first level of slot-types defined beneath `Primitive-Slot` are `Generic-Slots`. Generic slots are the way ARLO implements *generic objects*, an object oriented (as opposed to slot oriented) method of dispatching certain slot and unit operations.

Beneath `Generic-Slot`, ARLO defines `Typed-Slots` whose values and attachments (the units they attach their values to) must satisfy certain predicates. Beneath `Typed-Slot` is defined `Slot`. the protoypical slot referred to by most of ARLO's definition. `Slot` is a generic type-restricted slot which may compute "assumed" values for its assignments.

The functional properties of these slots are not — unfortunately — automatically merged from components along the hierarchy, but are hand-coded into implementation functions at for each new type of slot. The `To-Put-Value` slot of `Typed-Slot` for instance. is hand-coded to operate generically, rather than automatically accquiring the generic nature of `Generic-Slot`'s modifiers. Of course, this hand-coding is only neccessary because they share the functional role of slot modification; the `To-Put-Value` slot of the defaulting `Slot` need not be specially coded, since `Slot` defines no new modification functionality and may just inherit `Typed-Slot`'s `To-Put-Value` without merging.

2.5.1 Generic Objects & Shadow Slots

ARLO implements generic objects — as in `SmallTalk` [GR84] or *flavors* [WM82][Can83] — with a mechanism called *shadow slots*. In languages like SmallTalk or the Lisp Machine flavor system, the primary functional operation is message passing, where an object is sent a message in order to perform an operation on or with it. These languages are *generic* in that each object (or more precisely, each class of objects) has local definitions for handling the messages it receives in different ways. In ARLO, on the other hand, the primary functional operation is slot access (though the slot accessed may contain the definition of some functional operator), and the character of the operation is determined by the global description of the slot being accessed. Slots which are *generic*, however, permit a unit to shadow their global definition with a locally specified redefinition; these redefinitions are other full-fledged slot descriptions which supersede the global defaults. Thus, particular units may redefine some slot's definition (where the definition is an ARLO description) for themselves.

Shadow slots are implemented as a non-invasive extension of ARLO. By non-invasive, I mean that the implementation does not modify ARLO's reflexive operator mechanism but simply builds upon it. This is done by having the implementation of a generic slot (as functions stored on the slot's description) explicitly check for replacement definitions of themselves on the the unit they are operating on. Most of ARLO's slots (and most of the slot accessing functions offered to users) contain this explicit check, encoded by the macro `Shadowing-Slot`.

A `generic` slot looks for any "shadowed" definitions of itself by extracting its own `Shadow-Slot-Slot` from the unit it is operating on. For instance, the `Home-Phone` slot might have a `Shadow-Slot-Slot` of `Shadowed-Home-Phone-Slot`. The `Shadowed-Home-Phone-Slot` of any particular unit then contains the redefinition of `Home-Phone` — if any — to use on that unit. Then, descriptions of people with unusual phone numbers — overseas or buried in extensions — might have a `Shadowed-Home-Phone-Slot` whose definition would make their numbers acceptable or accessible despite some assumed global standard defined on `Home-Phone`.

2.5.2 Type Restricted Slots

Another abstract slot is the *type restricted* slot. The type restriction mechanism in ARLO refers to *types* defined in a *non-excepting* generalization hierarchy of predicates; the value and attachment (the unit a slot attaches its value to) of a type-restricted slot are constrained by a pair of these types. (This hierarchy is described in further detail in Section 2.7.) The `Data-Type` slot of a type-restricted slot determines what types of values the slot may accept; its `Makes-Sense-For` slot determines what types of objects (typically units) the slot may be attached to. The type checking predicates of a slot's `Data-Type` and `Makes-Sense-For` slots are merged into its `To-Verify-Type` slot; this value is a function of a unit, slot, and value about to be combined which signals an error if either of the predicates fails. This error is proceedable, but of course such an action may have dangerous repercussions.

Most of the slots of ARLO's initial configuration are type restricted slots, constraining themselves by reference to the predicate generalization hierarchy; but the relations forming this hierarchy (in their recursive turn) are described and defined by ARLO. This circularity of reference is initially established when the type hierarchy is bootstrapped (recalling Figure 2-1), a major event in ARLO's compilation and deployment.

2.5.3 Defaulting Slots

SLOT is the abstract slot first referred to by most representations built on top of ARLO. As well as having type restrictions as described above, units inheriting from SLOT have *defaulting methods* for generating absent values. When no value for this sort of slot can be found directly on a unit, a default is generated by calling the function stored on the slot description's To-Default-Value slot. This function is called on both the unit being referenced and the slot being defaulted and returns the value computed and a truth-value (T or NIL) to indicate the success of the computation.[13]

Often, the To-Default-Value property of a particular slot must also be generated by default; the To-Default-Value slot of **To-Default-Value** first tries to get a LISP implementation off of the slot's High-Level-Definition and failing this, ascends the hierarchy of abstract slot specifications (the Prototype hierarchy) looking for a To-Default-Value slot to use. A slot's High-Level-Definition — if it has one — is an abstract function description which may be implemented by a lisp coder as described in Section 2.6 below.

In the final analysis, the semantics of most slots built on ARLO's core (those inheriting from CORE:SLOT) are determined by the two components of ARLO just introduced: the *coder mechanism* which describes how "assumed" properties are computed and the *type mechanism* which constrains the values of slots by predicates in a generalization hierarchy. Both of these modules are described in more extensive detail below.

2.6 The ARLO Coder

ARLO implements a facility called *coders* for generating lisp code from high level functional descriptions. This facility is implemented by a representation language — described and implemented in ARLO — for describing LISP functions. Using this language, a user — or a sophisticated program — describes how partial specifications of particular sorts of function are expanded into fully implemented lisp definitions. Coders allow common representation functions — like property inheritance, network searches, function composition, or value restriction — to be generated from their functional specification. Every coder generated function begins with an ARLO unit which partially describes the function to be generated; the operational slots of the function description — its lambda-definition, documentation, etc — are generated as *defaults* from this description. When a user or program defines a particular *coder*, she is actually defining the way in which certain slots — such as Lambda-Definition or **Documentation** — default for a particular sort of functional description.

Each time a coder implements a particular function, it constructs a unit describing the function; the LISP definition, documentation, and name of the function are then generated by referring to methods stored on the Coded-By slot of the description. The value of the Coded-By slot is also an ARLO unit — a *coder* — which has functional properties like Implementor, **Documentor**, or To-Name-Function. Coders — with these

[13]This second value uses the *multiple value returns* of LISP Machine LISP.

relevant slots — are defined by a `Define-Coder` form:

```
(Define-Coder ( Coder-name . description-parameters)
              documentation-for-coder
   ( function-name-specification . arguments-for-function)
    documentation-specification
   body-specification)
```

`Define-Coder` constructs a unit named *coder-name* which describes how to generate functions of some particular type from specified *description-parameters*. These functions are actually generated as appropriate `Lambda-Definition` slots for descriptions which are initialized with some given *description-parameters*. Description parameters are slots stored on the functions' descriptions, and it is by reference to these properties of the description that the coder generates implementations, names, and documentation.

Each *description parameter* is either a symbol — in which case an indistinguished function-describing slot with that name is created — or a list whose first element specifies a slot/parameter name, and whose remaining arguments are slot-value pairs to be deposited on the slot's description (perhaps affecting it's implementation).

Function-Name-Specification specifies how to generate names for the functions the coder generates. If it is a symbol (such as MATRIX-MULTIPLY), each function name is an iterated gensym of that symbol (e.g. MATRIX-MULTIPLY-7). If the specification is a lisp form, it is evaluated to produce each function name, referencing the *description-parameters* of the coded function as free variables, and the function description itself by the variable *coder-name*.

Arguments-For-Function is the argument list for each function the coder generates. ARLO also knows how to extract the argument list for general system functions not synthesized by ARLO, allowing operations which use the argument list — such as functional composition — to be applied to functions defined by either the user or other resident systems.

Documentation-Specification is a form which, accessing the description parameters and function description as free variables, prints documentation for the function to the stream `STANDARD-OUTPUT`.

Finally, *body-specification* is a lisp form returning the body forms of the function generated by the coder. As with the previous structure generating forms, this form may reference the description parameters and function description as free variables.

The `Define-Coder` form creates a *coder description* — an ARLO unit named *coder-name* — which describes how to generate names, documentation, and lambda definitions for the functions it codes. It also implements a generator function, named *coder-name*, which constructs a function description with the appropriate `CODED-BY` slot and with description parameters from its arguments. The function defaults the lambda-definition — and LISP compiled definition — for this function description, finally returning the generated name of the function.

2.6.1 Representing Programs

The coder mechanism was originally conceived as an embryonic, poor man's version of the *plan cliche* representation used in the Programmer's Apprentice project at MIT [SR76][Ric80][Wat78] By representing typical representation functions in this explicit way, the task of understanding or intelligently modifying ARLO

definitions is far more straightforward. Mutative systems such as AM and Eurisko generally modified LISP functions by heuristically munging s-expressions which encoded LISP definitions of relevant functions. The coder mechanism was designed to make explicit and accessible — in an ARLO representation — descriptions of the implementations of many of the system's functions and operations.

2.6.2 ARLO's Coders

The initial ARLO configuration defines 7 coders:

- `Slot-Composition` takes a list of slots and constructs a function which is their composition. For instance, a composition of the `Father` and `Mother` slots would be a function for extracting ones paternal grandmother.
- `Inherit-Through` generates a function for inheriting through a particular relation.
- `Methods` constructs a composite function from a list of other functions, which may also be generated by coders. The function generated tries each function — one after another — until one succeeds (returns without error). This function is called a *try-and-try-again* function, trying one method after another until one finally succeeds, being careful about the accumulated dependencies of each attempt (it resets the dependency tracking mechanism before each attempt.)
- `Expecting` is like `Methods`, but the function it constructs only "tries again" if a preceding method fails in an "expected" way. Of course, if an unexpected error occurs inside of an `Expecting` function, it may well be caught by other `Expecting` or `Methods` coded functions above it in the calling sequence.
- `Test` is a coder which generates a predicate function which is the conjunction of its component functions.
- `Inherits?` is a coder for predicates which determine if one unit inherits from some other through some relation. (For instance, if some person is directly above some other in some hierarchy.)
- `Type-Checker` generates the function for verifying a slot's assignment — its value and attachment — from the slot's `Makes-Sense-For` and `Data-Type` properties.

2.6.3 User Defined Functions

The function description language used by the coder is also used to record user function definitions. The function `DEFINE` has the syntax of LISP's `DEFUN`, but builds an ARLO description with appropriate Lambda-Definition and Documentation slots. The function $\lambda\lambda$ is an inline version of `DEFINE` which returns the name of the function it defines.

The function `GET-FUNCTION-DESCRIPTION` finds or generates an ARLO description of the function specified by its single argument. Of course, if the function was not appropriately defined (by `DEFINE`, $\lambda\lambda$, or some automatic coder), some information (such as lambda definitions) may not be available.

2.7 The Type System

The coder mechanism is used by ARLO in two roles: the implementations of "defaulting functions," and the specification of ARLO's hierarchy of types. In this second role, coders are defined which implement common representational predicates (such as checking inheritance over various relations) and particular conjunctions of such predicates. These generated predicates are defined in a *generalization* hierarchy, descending from broader predicates (satisfied by large numbers of objects and units) into progressively more particular

predicate categorizations. Each of the predicates in this hierarchy is represented by a "type," an ARLO description which consolidates a predicate function with associated functions for describing and operating on objects which satisfy the predicate. ARLO's "type hierarchy" is the predicate generalization hierarchy imposed between these type descriptions.

The type hierarchy also fills two distinct roles in ARLO's initial configuration. First of all, its predicates serve to constrain the "sensible" attachements and values for particular slots; secondly, it provides information about how to print, describe and parse the sorts of values known to belong to certain sorts of slots. For systems implemented in ARLO, beyond the definition of ARLO itself, it both provides constraints on the generation of new slots from old and serves as a hook for hanging type specific knowledge in the form of inference procedures or heuristics.

The generalization hierarchy between types is determined by two slots: the `Generalization` slot and the `Specification` slot. The `Generalization` of a type is the type upon which a type is built; a type's `Specification` determines what additional criterion objects of the type must satisfy. The predicate for a given type is hence the conjunction of the type's specification and the predicate of its generalization. This principle yields a strict generalization hierarchy — any instance of T is also an instance of $Generalization(T)$ — which simply supports operations like *classification* (quickly finding the types which an instance satisfies by traversing downwards the tree of generalizations) or `property clustering` (automatically generating new types from old by specializing around particular property regularities in their instances). In addition to providing a formal framework streamlining these sorts of operations, the generalization relation is used to inherit type specific properties such as display functions, description parsers, or inference procedures.

The type system presents its own bootstrap problems; it is described in ARLO (as as a representation langauge for hierarchically organizing predicates and their properties), but is used (in turn) to constrain the values and attachments of ARLO's own definition. As a result, ARLO's type bootstrap is more complicated than its representation bootstrap (which was described in Section 2.5). When ARLO is originally defined as a representation describing language, its type restrictions and constraints are represented by symbolic tokens referring to type *names*. ARLO's second bootstrap — its type bootstrap — takes these symbolic tokens and replaces each type name in ARLO's self-description with a pointer to the actual type-describing unit it refers to. The timing of this bootstrap is critical, as the type system uses both ARLO and the coder mechanism in its definition, and enough of these mechanisms must be compiled and cached before the type system is completely enabled.

The package of ARLO utilities implemented for CYRANO significantly extends the type system into a general *classification* system. This extension includes a *classifier* for placing instances in the hierarchy of predicates (similar to the `KL-ONE` classifier) and an implementation of heuristic and inferential rules whose "IF" components refer to the type hierarchy. This innovation automatically places rules in a generalization/specialization hierarchy from which they may be indexed to particular objects or tasks. A new variety of automatic predicate coders accompanies this extension, permitting the specification of constraints on and between various sub-parts of descriptions.

2.8 Archives and Layers: Saving Representations

Upon the edifice described in the preceding sections, users and clever programs build both new representation schemes and particular representations within those schemes. Much of this construction takes place in the same manner as ARLO's own initial construction: through top level forms which side-effect the environment to install particular representations and representations of representations. But much of the structure built on top of ARLO is a dynamic stuff, constructed by interactive editors, database parsers, or thoughtful programs. The preservation of these structures — defined in no particular file, but only by the accumulation of definitions and mutations over time — is critical if any of our programs is to have a life beyond a single session or a handful of examples.

Archives and layers are ARLO's tools for saving out collections of in-core units and their relations; units and relations are *dumped* as data files from which they may later be restored. An archive stores a collection of units and their connections, a layer stores the *changes* in such collections of units and connections. Archives are used to store bodies of knowledge and the representation schemes (in ARLO, another sort of body of knowledge) in which they are expressed; layers are used to store personal modifications or incremental changes to these archives.

The implementation of archives and layers posed many difficulties, most of the arising from the circularity and complexity of ARLO's inter-relations and dependencies. It is worth noting that the Lisp Machine system, not designed to support the structured preservation of partial environments, had to be significantly extended to permit dumping of ARLO structures. This section, however, will concern itself only with the dumping abstractions used by ARLO, and not the implementation particular details of their realization.

Like nearly every other process in ARLO, the dumping of ARLO units and relations is data-directed. The archive to which a unit belongs is a slot of the unit; every ARLO unit is given (or assumes by default) a My-File-Of-Definition slot. For units defined by top-level LISP forms, this is the file in which the LISP forms appeared; for other units, this slot is the **archive** the unit is defined in. A unit's archive is either an explicitly deposited pathname or (by default) a logical pathname fo the form `''ARLO: KBases; kb BIN >''`, where *kb* is the *knowledge base* the unit was originally defined in. The `#>My-File-Of-Definition` slot is defined (as an ARLO slot) to maintain backpointers from archive pathnames to the units defined in them. Thus, when the user asks to dump and archive (specified by its pathname), the set of units to be dumped are avaliable as a property of the pathname.

An archive is dumped through forms which bind — at load time — particular unit names to unit objects; the reference to each unit object is then realized in forms which access or regenerate the unit. Any given unit reference is either *local* or *external*: a local unit reference refers to a unit in the current archive; an external unit refers to a unit in some other archive. External unit references dump as a pair of unit name and unit archive; if — at load time — the unit name is unbound, its archive is loaded, and the unit is then directly referenced.

Local unit references dump as either per-file dumped-object indices (supported natively by the ZetaLisp binary dumper) or as forms which regenerate the unit. In the first case, a regenerating form has already appeared in the file and the restored object is directly referenced; in the second case, the regenerating form must be produced, and this production is done by calling the `#>My-To-Dump-Self` slot of the unit on the unit. The value returned by this function is a form which regenerates the unit and any attached portions of its environment. For instance, when a function describing unit is regenerated, its definition is recompiled

into the load-time environment; if a unit describing an active process is loaded, that process might be instantiated and started when the unit is restored. The `#>My-To-Dump-Self` slot of a unit need only take care of reestablishing particular parts of the LISP environment dependent on, or depended on by, the unit dumped. A collection of canonical dumping functions (such as `DEFAULT-UNIT-DUMPER`) provide regeneration forms which handle reestablishing the ARLO environment connected to a particular unit. These forms must not only reestablish a frame with its connected slots, but must reestablish the units and slots those slots refer to; when this reestablishment must reach between archives, it becomes insoluble in general and difficult in particular.

The problem can be characterized in the following way. Every archive has an *edge* where it connects to other archives; a given archive has certain assumptions about what lies over its edge, but it only has limited information about the content of these bordering archives. When an archive is reloaded, it is not reloaded in a vacuum, but must be established with its original edge connections intact. When inconsistent changes have been made to multiple archives (an archive X refers to a unit in an archive Y which was never dumped) the problem is insoluble; but if a degree of consistency is maintained, then the problem of establishing an archive amongst its neighbors requires dumping the archive to *just beyond its edge.*

Most of the responsibility for reestablishing the cross-archive network is carried by ARLO's dependency network. Since this network specifies most of the explicit or implicit connections in the ARLO slot network, reestablishment of the dependency network reestablishes parts of the ARLO unit-slot network as well. By using references to dependencies over a given edge, many of the problems of dumping partial environments are finessed or solved; "assumptions" of the network just outside a particular archive — just over its edge — are found or recreated when the archive is loaded. When this search or recreation fails (when an external dependency is assumed that does not exist) the loader "fakes" the dependency and warns the user of the temporarily patched inconsistency.

2.8.1 Layers

Layers are the way ARLO records incremental changes to its descriptions. Their mechanism is quite simple: at some point (typically after an archive or set of archives is loaded) the state of a collection of archives is *frozen* into a "layer". Then, at some later point after a series of introductions and modifications to the archives, the differences between the frozen layer and the current state of the archives is computed, and appropriate modifying forms are dumped in much the same manner as an archive. In this process the data direction and cross-archive connection proceeds as above.

Chapter 3
An Example: Representation

This section describe a toy ARLO database of researchers and their interrlations. It is part of the default system, residing in the `Inquir` knowledge base, useful for testing and demonstration. The first section of this example describes and explains what the code in the file looks like; the second is a script of an interactive examination -- on the LISP Machine — of the domain and its representation langauge.

3.1 Building Basics

The first step in building a representation system in ARLO is to define the basic essential units and relations on which the individuals and relations of your representation will build. If you are building on top of raw ARLO, the inheritance mechanism you are likely to use is `#>Prototype` inheritance; if you are using a system built on top of ARLO (for instance, an FRL or KLONE clone) you may be using an entirely different mechanism. Of course, if you wish, you can easily implement your own inheritance scheme in ARLO and use that.

The following expressions describe the prototypical person, construct a unit describing the "person type", and build a prototypical slot from which slots referring to people will eventually inherit.

```
(DefUnit Person
  (Description ``This is the prototypical person.''))
```

```
(DefUnit Person-Type
  (Description
    ``This is a type satisifed by any unit inheriting from Person.'')
  (Prototype #>Any-Type)
  (Generalization #>Unit-Type)
  (Specification (Inherits? #>Person #>Prototype))
  (#>Function-To-Find-Interactively 'get-person-from-menus)
  (#>Function-To-Read 'read-person))
(Put-Value #>Person #>My-Specific-Type #>Person-Type)
```

```
(DefUnit Person-Slot
  (Description
    ``This is the prototypical slot which attaches to people.'')
  (Prototype   #>Slot)
  (Makes-Sense-For #>Person-Type))
```

The definition of the `#>Person` unit constructs a "placeholder" to which individual people descriptions will refer. Later, we may burden this unit with a variety of information which those individual people descriptions will inherit of refer to. For instance, the `#>Person` unit may be used to shadow some slot definitions in order to accomodate the restrictions and potentials of people.
`#>Person-Type`

is defined as a specialization of `#>Unit-Type` which requires inheritance — via the `#>Prototype` relation — from the unit `#>Person`. The generalization hierarchy used for types is a non-excepting hierarchy of predicate specifications. ARLO's utilities implement a KLONE-style classifier for this generalization hierarchy, determining which types in the hierarchy are instantiated by a given LISP object or ARLO description.

The `#>My-Specific-Type` slot of a unit is an ARLO type description subsuming all ARLO units inheriting (via the `#>Prototype` relation) from the unit. `#>Person-Type` is deposited there as a forethought; if we had asked for the `#>My-Specific-Type` slot of `#>Person` without storing `#>Person-Type` there beforehand, an appropriate type description would have been generated on the fly. One thing we will exploit `#>Person-Type` for is defining the way references to people are parsed, printed, and described.

Finally, `#>Person-Slot` is a version of `#>Slot` which embodies a particular constraint on the units it may be attached to.

3.2 Defining Slots

The following expressions define slots for the various appellations for individual people; these slots present a variety of different value defaulting mechanisms.

```
(DefUnit Full-Name
  (Description ``This is the full, formal name of a person.'')
  (Prototype #>Person-Slot)            ; Attach to people
  (Data-Type  #>String-Type)           ; Accept strings
  (To-Default-Value 'ask-user-for-slot)
  (To-Prompt-For-Value
    (λλ ask-for-full-name (person slot stream)
      (format stream "What is the full name of the person described by  a?"
              person))))

(DefUnit Personal-Name
  (Description ``This is the informal name of a person.'')
  (Prototype #>Person-Slot)            ; Attach to people
  (Data-Type  #>String-Type)           ; Accept strings
  (To-Default-Value
    ;; The λλ macro — briefly mentioned on page 24 — internally defines
    ;; an external function constructing an ARLO description of the function at the same time.
    (λλ To-Generate-Personal-Name (unit slot)   ; Extract her first name
      (if (Ignoring-Errors (get-value unit #>Full-Name))
          (get-first-word (get-value unit #>Full-Name))
          ``Friend''))))

(DefUnit Last-Name.
  (Description ``This is the last name of a person.'')
  (Prototype #>Person-Slot)            ; Attach to people
  (Data-Type  #>String-Type)           ; Accept strings
  (To-Default-Value
    (λλ (unit slot)                    ; Extract her last name
      (if (Ignoring-Errors (get-value unit #>Full-Name))
          (get-last-word (get-value unit #>Full-Name))
          ``Random''))))
```

The above are examples of slots which compute their defaults in different ways. The `#>Full-Name` slot, for instance, asks the user for a person's full name if it isn't already specified. The `Personal-Name` slot, on the other hand, extracts the person's first name from her full name if possible and otherwise defaults to a friendly solution. The `Ignoring-Errors` form used in the definition catches difficulties with inaccessible slots or formats, returning nil if any errors were encountered in the execution of its body. The `#>Last-Name` slot is almost a copy of `#>Personal-Name`, extracting a last name from the `#>Full-Name` slot if possible and otherwise defaulting to a random solution. In both of these slots we see an explicitly defined lambda-definition specified instead of an automatically coded high-level description.

The `#>Makes-Sense-For` slot for all of these units defaults from `#>Person-Slot`, and each accepts only LISP strings for values.

3.3 Inheritance Mechanisms

The following slots illustrate how ARLO supports explicitly defined inheritance mechanisms of various sorts.

```
(DefUnit Supervisor
  (Description ``This is the supervisor of a person.'')
  (Prototype #>Person-Slot)            ; Attach to people
  (Data-Type  #>Person-Type)           ; Chauvinist, but....
  (To-Default-Value 'ask-user-for-slot)
  (To-Prompt-For-Value
    (λλ ask-for-supervisor (person ignore stream)
     (format stream "Who is  a hacking for?"
             (get-value person #>Personal-Name)))))

(DefUnit Hacking
  (Description ``This is what a person is hacking on.'')
  (Prototype #>Person-Slot)            ; Attach to people
  (Data-Type  #>String-Type)
  (To-Prompt-For-Value
    (λλ ask-for-hacking-slot (person ignore stream)
      (format stream "What is  a hacking?"
              (get-value person #>Personal-Name))))
  (High-Level-Definition
    ;; Default from ones' supervisor, and otherwise ask...
    ;; (The character macro #$ returns a DESCRIPTION of the
    ;;   function whose name follows it.)
    (METHODS (list (Inherit-Through #>Supervisor) #$ask-user-for-slot))))

(DefUnit Working-in-Field
  (Description ``This is the field a person is working in.'')
  (Prototype #>Person-Slot)            ; Attach to people
  (Data-Type #>String-Type)
  (High-Level-Definition               ;Another way to say it
    (Slot-Composition (list #>Hacking #>Supervisor))))

(DefUnit Wedging
  (Description ``A monkey wrench in the works.'')
  (Prototype #>Person-Slot)
  (Data-Type  #>String-Type)
  (To-Default-Value (λλ Wedge (un sl) (get-value un sl))))
```

The first of the slots defined above is the `#>Supervisor` slot, which is used to default the values of a variety of other slots. The type restriction of `#>Supervisor` demands that its value be another person-describing unit, since other slots will be looking at its value — with unit accessing functions — to derive their own values.

The second and third slots defined above perform inheritance (or defaulting) in different ways. The #>Hacking slot attempts to inherit its value by searching through the #>Supervisor relation, but if it fails — for any reason — it asks the user for the value. The #>Methods coder used to define this mechanism takes its clauses and constructs a try-and-try-again function. (Try-and-Try-again functions are briefly described on page 18.)

The #>Working-In-Field relation refers to one's supervisor for its value also, but if this fails, the entire attempted computation fails. In addition, the #>Slot-Composition coder is not characterized as a search, so the function it generates will be implemented somewhat differently. (It will not, for instance, signal a #>Slot-Not-Found condition if it fails.)

Finally, the #>Wedging slot is merely there for purposes of demonstrating how fatal-recursion detection works. Since the wedging slot defaults by getting its value, trying to compute a default for it will recurse indefinitely.

3.4 Shadowing Slot Definitions

To demonstrate the ARLO mechanism for shadowing slots, we construct two special units. The first, #>Shadowed-Hacking, describes how to find and store a shadowed definition for the #>HACKING slot; this descriptions is another slot, defined to get its value by searching (with the LISP function Find-Value through the #>Prototype slots of a unit. To redefine the definition of #>Hacking for a group of units, we merely arrange that they have as a prototype some unit with the appropriate #>Shadowed-Hacking slot. In this particular example, we define a unit #>Winner with a shadowed definition of #>Hacking which asks the user for the slot's value, without first trying to inherit a value through the #>Supervisor relation.

```
(DefUnit Shadowed-Hacking-Definition
  (Description ``A replacement definition for HACKING.'')
  (Prototype #>Shadow-Slot)
  ;; Search through prototypes for a value.
  (To-Default-Value
    (λλ Find-Hacking-Slot (unit in-slot)
      ``Looks for a replacement hacking definition.''
      (or (find-value unit in-slot) #>hacking))))
(Put-Value #>Hacking #>Shadow-Slot-Slot #>Shadowed-Hacking-Definition)
```

```
(DefUnit Winner
  (Description ``Somone who doesn't always follow their supervisor.'')
  (Shadowed-Hacking-Definition
   ;; When we construct a unit with a #>My-Name slot, the true name
   ;; of the constructed unit will be an enumerated gensym of the My-Name
   ;; slot (e.g. #>Hacking-0, #>Hacking-1, etc).
   (make-unit (My-Name '#>Hacking)
              (Prototype #>Hacking)
              (Makes-Sense-For (get-value #>Winner #>My-Specific-Type))
              (To-Default-Value 'ask-user-for-slot))))
```

As a result of the above machinations, any person descriptions which have a prototype of `#>Winner` instead of `#>Person` will use this alternate definition of `#>HACKING` in place of the one originally defined at the top level.

3.5 Building the data base

The process of creating "individuals" in this example builds on the slots and prototypes constructed above. Currently, there are two standard ways to build individuals in ARLO. One may either call `DefUnit` explicitly from top level (the manner in which the slots above were created), or write support functions calling `Make-Unit` internally to construct units with particular properties. For purposes of clarity and brevity, this example uses only the first of these techniques, explicitly defining each individual person description at top level.

The following `DefUnit` forms build a small database of people-describing units for an imaginary AI lab.

```
(DefUnit Calvin
  (Description "This is a well known robotics hacker.")
  (Prototype #>Person)
  (Full-Name ``Susan Calvin'')
  (Hacking ``Robots''))

(DefUnit Rodgers
  (Prototype #>Person)
  (Full-Name ``Robert Rodgers'')
  (Supervisor #>Calvin)
  (Hacking ``Emotional Analouge Robots''))

(DefUnit Charo
  (Prototype #>Person)
  (Full-Name ``Elizabeth Charo'')
  (Personal-Name ``Beth'')
  (Supervisor #>Calvin)
  (Hacking ``Cognitive Fundamentals''))
```

```
(DefUnit Lee
  (Prototype #>Person)
  (Full-name ``Pat Lee'')
  (Supervisor #>Calvin)
  (Hacking ``Engineering Design''))

(DefUnit Kyle
  (Prototype #>Person)
  (Full-Name ``Kyle O'Shea''))

(DefUnit Arthur
  (Prototype #>Person)
  (Full-Name ``Arthur Pendragon'')
  (Hacking ``Fantasy Games''))

(DefUnit Alice
  (Prototype #>Person)
  (Full-Name ``Alice Adams''))

(DefUnit Brian
  (Prototype #>Winner)
  (Full-Name ``Brian Walking-Song'')
  (Supervisor #>Chare))
```

3.6 At the Console

3.6.1 Defaulting of Slots

```
(kb-goto 'inquir) [Change the default knowledge base. ]
#<Package CORE:INQUIR 66156707>
(examine-unit #>Kyle) [Let's look at Kyle's description. ]
Description of the ARLO unit {#>KYLE}:
Description:              The description of KYLE was not provided.
Prototype:                {#>PERSON}
Prototype Of:
My Creator:               Ken Haase
My File Of Definition:    ARLO: SOURCES; INQUIR * >
My Time Of Creation:      Saturday the twenty-eighth of July, 1984; 12:02:01 am
Full Name:                Kyle O'Shea
My Name:                  #>KYLE
My To Describe Self:      #'LOOK-AT-UNIT
My To Print Self:         #'DEFAULT-UNIT-PRINTER
```

The slots of Kyle's description are tabulated above: slot names on the right, values on the left. Note that the values are printed out based on the semantics of the slot. #>MY-TIME-OF-CREATION, *while stored as an integral number of seconds past New Year's Day 1900, prints out in a standard English format. Each unit is also annotated with the file of creation and (if provided) a string describing the unit in English. In Kyle's case, there is no description provided so a default (describing the lack of an ascribed description) is provided. But the description above has no real information about Kyle: what he does, who he works for, etc. So we begin our interaction by querying about these things...*

```
Editing {#>KYLE} >>G Describe Slot Value
Which slot of {#>KYLE} would you like to see?Hacking
```

Here we ask for the **Hacking** *slot of the unit. Since there is not one there already (as we can tell from the description just provided) its value must be defaulted using the function on the* To-Compute *slot of* **Hacking**. *This function – as described by its high level definition provided above – first looks through the* **Supervisors** *of the person and then – if that fails – asks the user at the console for a value. But in order to search through the supervisors of* **Kyle** *it must first know who his immediate supervisor is. Since the* **Supervisor** *slot defaults by asking the user at the console, we are asked...*

```
Who is Kyle hacking for?Pat
```

Pat is the first name of "Pat Lee", the person we are referring, but since the value of the **Supervisor** *slot is of* **Person-Type**, *ARLO knows to read its value with a function which looks for people under their personal names (as well as last names and their names qua description). It finds the unit named* **Lee**, *based on our information, and caches it as Kyle's supervisor. Having this information, it looks on Lee's description for a* **Hacking** *slot, and discovers....*

```
The Hacking slot of {#>KYLE} is:  Engineering Design
This is justified by:
The Hacking slot of {#>LEE} is:  Engineering Design
The To Get Value slot of {#>HACKING} is:  #'TYPED-DEFAULTING-GET
The Supervisor slot of {#>KYLE} is:  {#>LEE}
The To Default Value slot of {#>HACKING} is:
#>INHERIT-THROUGH-SUPERVISOR-OR-ARLO:ASK-USER-FOR-SLOT-OR-ELSE
The To Get Value slot of {#>TO-DEFAULT-VALUE} is:  #'TYPED-DEFAULTING-GET
```

As promised, ARLO keeps track of the dependencies – the "assumptions" – of its derivations. In this case, Kyle's hacking slot depends on his supervisor being Lee, Lee's hacking of "Engineering Design," the mechanism by which the hacking slot defaults, and the implementation of that mechanism for defaulting. These four dependencies are summarized by ARLO below. Note that if any of them were to change the "assumed" value of Kyle's hacking slot would be invalid. Thus, in the event that any of these values is retracted or otherwise invalidated, ARLO can use its dependency information to make sure that the value just computed is retracted and invalidated as well.

3.6.2 Dependencies and Decaching

The dependencies ARLO recorded for the **Hacking** *slot above allow it to keep the slot's value up to date. This is neccessary because its value is cached on the unit, as we can see from its description below:*

```
Editing {#>KYLE} >>Describe [Describes the unit.. ]
Description of the ARLO unit {#>KYLE}:
Description:            The description of KYLE was not provided.
Prototype:              {#>PERSON}
Prototype Of:           None
My Creator:             Ken Haase
My File Of Definition:  ARLO: SOURCES; INQUIR.LISP
My Time Of Creation:    Saturday the twenty-eighth of July, 1984; 12:02:01 am
Full Name:              Kyle O'Shea
Hacking:                Engineering Design [The hacking slot, cached. ]
Last Name:              O'Shea
My Name:                #>KYLE
My To Describe Self:    #'LOOK-AT-UNIT
My To Print Self:       #'DEFAULT-UNIT-PRINTER
Personal Name:          Kyle
Supervisor:             {#>LEE}
                        [Kyle's supervisor, cached also.  We won't be asked for it again. ]
Editing {#>KYLE} >>Edit
Which slot of {#>KYLE} would you like to edit?Supervisor
```

Now let's go look at Kyle's supervisor and change his **hacking** *slot. The change should propogate back to Kyle.*[14]

```
Description of the ARLO unit {#>LEE}:
Description:              The description of LEE was not provided.
Prototype:                {#>PERSON}
Prototype Of:
My Creator:               Ken Haase
My File Of Definition:    ARLO: SOURCES; INQUIR * >
My Time Of Creation:      Saturday the twenty-eighth of July, 1984; 12:02:01 am
Editing {#>LEE} >>Describe [Describes the unit.. ]
Description of the ARLO unit {#>LEE}:
Description:              The description of LEE was not provided.
Prototype:                {#>PERSON}
Prototype Of:
My Creator:               Ken Haase
My File Of Definition:    ARLO: SOURCES; INQUIR * >
My Time Of Creation:      Saturday the twenty-eighth of July, 1984; 12:02:01 am
Full Name:                Pat Lee
Hacking:                  Engineering Design
                          [The value which Kyle's Hacking slot defaulted from. ]
My Name:                  #>LEE
My To Describe Self:      #'LOOK-AT-UNIT
My To Print Self:         #'DEFAULT-UNIT-PRINTER
Personal Name:            Pat
Supervisor:               {#>CALVIN}
```

Now we store a value in Lee's `#>HACKING` *slot. When reading a hacking slot, whose value must be a string, ARLO knows to use the LISP* **readline** *function. One might imagine that - if ARLO were connected to a natural language interface - the same sort of knowledge might be used to generate discourse goals.*

```
Editing {#>LEE} >>Set Slot Value
Which slot of {#>LEE} would you like to set?Hacking
What would you like in the Hacking slot of {#>LEE}?The Grateful Dead[A string is read. ]
```

Now we have given Lee a new **hacking** *slot, and the value should have replaced the old one. We ask for Lee's description:*

```
Editing {#>LEE} >>Describe [Describes the unit.. ]
Description of the ARLO unit {#>LEE}:
Description:              The description of LEE was not provided.
Prototype:                {#>PERSON}
Prototype Of:
My Creator:               Ken Haase
My File Of Definition:    ARLO: SOURCES; INQUIR * >
My Time Of Creation:      Saturday the twenty-eighth of July, 1984; 12:02:01 am
Full Name:                Pat Lee
Hacking:                  The Grateful Dead  [The new value.. ]
My Name:                  #>LEE
My To Describe Self:      #'LOOK-AT-UNIT
My To Print Self:         #'DEFAULT-UNIT-PRINTER
Personal Name:            Pat
Supervisor:               {#>CALVIN}
```

If everything worked, our change in Lee's **Hacking** *slot should have invalidated the default which ARLO computed earlier for Kyle. We finish editing* #>LEE *and return to editing* #>KYLE:

```
Editing {#>LEE} >>Quit
Finished editing {#>LEE}
Editing {#>KYLE} >>Describe [Describes the unit.. ]
Description of the ARLO unit {#>KYLE}:
Description:              The description of KYLE was not provided.
Prototype:                {#>PERSON}
Prototype Of:
My Creator:               Ken Haase
My File Of Definition:    ARLO: SOURCES; INQUIR * >
My Time Of Creation:      Saturday the twenty-eighth of July, 1984; 12:02:01 am
Full Name:                Kyle O'Shea
                          [The hacking slot — here before — has disappeared... ]
Last Name:                O'Shea
My Name:                  #>KYLE
My To Describe Self:      #'LOOK-AT-UNIT
My To Print Self:         #'DEFAULT-UNIT-PRINTER
Personal Name:            Kyle
Supervisor:               {#>LEE}
```

Now we ask for the **Hacking** *slot again, and it will be defaulted as before, except that this time Kyle's* **Supervisor** *slot is already known and doesn't have to be asked for.*

```
Editing {#>KYLE} >>G -- Describe Slot Value
Which slot of {#>KYLE} would you like to see?Hacking
The Hacking slot of {#>KYLE} is:  The Grateful Dead
This is justified by:
The Hacking slot of {#>LEE} is:  The Grateful Dead
The To Get Value slot of {#>HACKING} is:  #'TYPED-DEFAULTING-GET
The Supervisor slot of {#>KYLE} is:  {#>LEE}
The To Get Value slot of {#>SUPERVISOR} is:  #'TYPED-DEFAULTING-GET
The To Default Value slot of {#>HACKING} is:
#>INHERIT-THROUGH-SUPERVISOR-OR-ARLO:ASK-USER-FOR-SLOT-OR-ELSE
The To Get Value slot of {#>TO-DEFAULT-VALUE} is:  #'TYPED-DEFAULTING-GET
Editing {#>KYLE} >>Describe [Describes the unit.. ]
Description of the ARLO unit {#>KYLE}:
Description:            The description of KYLE was not provided.
Prototype:              {#>PERSON}
Prototype Of:
My Creator:             Ken Haase
My File Of Definition:  ARLO: SOURCES; INQUIR * >
My Time Of Creation:    Saturday the twenty-eighth of July, 1984; 12:02:01 am
Full Name:              Kyle O'Shea
Hacking:                The Grateful Dead [The new default is now cached.. ]
Last Name:              O'Shea
My Name:                #>KYLE
My To Describe Self:    #'LOOK-AT-UNIT
My To Print Self:       #'DEFAULT-UNIT-PRINTER
Personal Name:          Kyle
Supervisor:             {#>LEE}
```

And now let us set the world back to normal, changing Lee's Hacking *slot once more and reinstating the old value on Kyle.*

```
Editing {#>KYLE} >>Edit
Which slot of {#>KYLE} would you like to edit?Lee
``Lee'' isn't the name of a defined slot.
```

We accidently referred to the unit to edit, instead of the slot of Kyle we wished to edit. Fortunately, the unit editor was clever enough to warn us of our mistake, but not clever enough to see through it.

```
Did you make a mistake?(Y or N) Yes
Which slot of {#>KYLE} would you like to edit?Supervisor
Editing {#>LEE} >>Set
Which slot of {#>LEE} would you like to set?Hacking
What would you like in the Hacking slot of {#>LEE}?Engineering Design
Editing {#>LEE} >>Quit
Finished editing {#>LEE}
```

The world should be back to normal now...

```
Editing {#>KYLE} >>G -- Describe Slot Value
Which slot of {#>KYLE} would you like to see?Hacking
The Hacking slot of {#>KYLE} is: Engineering Design
```

And indeed it is....

```
This is justified by:
The Hacking slot of {#>LEE} is: Engineering Design
The To Get Value slot of {#>HACKING} is: #'TYPED-DEFAULTING-GET
The Supervisor slot of {#>KYLE} is: {#>LEE}
The To Get Value slot of {#>SUPERVISOR} is: #'TYPED-DEFAULTING-GET
The To Default Value slot of {#>HACKING} is:
#>INHERIT-THROUGH-SUPERVISOR-OR-ARLO:ASK-USER-FOR-SLOT-OR-ELSE
The To Get Value slot of {#>TO-DEFAULT-VALUE} is: #'TYPED-DEFAULTING-GET
```

3.6.3 Other slots

```
Editing {#>KYLE} >>New unit
What unit would you like to edit?Alice
Description of the ARLO unit {#>ALICE}:
Description:            The description of ALICE was not provided.
Prototype:              {#>PERSON}
Prototype Of:
My Creator:             Ken Haase
My File Of Definition:  ARLO: SOURCES; INQUIR * >
My Time Of Creation:    Saturday the twenty-eighth of July, 1984; 12:02:02 am
Full Name:              Alice Adams
Last Name:              Adams
My Name:                #>ALICE
My To Describe Self:    #'LOOK-AT-UNIT
My To Print Self:       #'DEFAULT-UNIT-PRINTER
Personal Name:          Alice
Editing {#>ALICE} >>G -- Describe Slot Value
Which slot of {#>ALICE} would you like to see?Working In Field
```

The **Working-In-Field** *slot – as defined on Page 3.3 – is the slot composition of the* **Hacking** *slot of ones* **Supervisor** *slot. As before though, it needs to know her supervisor slot in order to default a value.*

```
Who is Alice hacking for?Rodgers
The Working In Field slot of {#>ALICE} is:  Emotional Analouge Robots
```

And again, ARLO provides the dependencies of the computation:

```
This is justified by:
The Hacking slot of {#>RODGERS} is:  Emotional Analouge Robots
The To Get Value slot of {#>HACKING} is:  #'TYPED-DEFAULTING-GET
The Supervisor slot of {#>ALICE} is:  {#>RODGERS}
The To Get Value slot of {#>SUPERVISOR} is:  #'TYPED-DEFAULTING-GET
The To Default Value slot of {#>WORKING-IN-FIELD} is:
#>THE-VALUE-OF-THE-HACKING-OF-THE-SUPERVISOR-OF
The To Get Value slot of {#>TO-DEFAULT-VALUE} is:  #'TYPED-DEFAULTING-GET
```

Finally, we ask for Alice's description again, to see that the appropriate slot has been cached on the unit descripition.

```
Editing {#>ALICE} >>Describe [Describes the unit.. ]
Description of the ARLO unit {#>ALICE}:
Description:              The description of ALICE was not provided.
Prototype:                {#>PERSON}
Prototype Of:
My Creator:               Ken Haase
My File Of Definition:    ARLO: SOURCES; INQUIR * >
My Time Of Creation:      Saturday the twenty-eighth of July, 1984; 12:02:02 am
Full Name:                Alice Adams
Last Name:                Adams
My Name:                  #>ALICE
My To Describe Self:      #'LOOK-AT-UNIT
My To Print Self:         #'DEFAULT-UNIT-PRINTER
Personal Name:            Alice
Supervisor:               {#>RODGERS}
Working In Field:         Emotional Analouge Robots [The value has been cached.. ]
Editing {#>ALICE} >>Quit
Finished editing {#>ALICE}
```

3.6.4 Errors

```
Back to editing {#>KYLE}
Editing {#>KYLE} >>G -- Describe Slot Value
Which slot of {#>KYLE} would you like to see?Wedging
```

If you remember the definition on page 31, this slot has a defintion which "defaults" by referring to itself again. Attempting to get this slot from a unit will recurse fatally if a value isn't already available. (In which case that value would simply be returned.) Let's watch sparks fly.

```
>>ARLO-Error: I seem to be fatally recursing on getting the Wedging slot of {#>KYLE}
              (While getting the Wedging slot of {#>KYLE})
```

A description of the current slot operation being attempted is always provided to the user when she is asked to handle an ARLO condition.

```
GET-VALUE:
   Arg 0 (IN-UNIT): {#>KYLE}
   Arg 1 (OF-SLOT): {#>WEDGING}
s-A, [RESUME]:     Perform the operation using subprimitives.
s-B, c-?:            Print out the current state of ARLO's computations.
s-C, [ABORT]:      Return to the examining the unit {#>KYLE}
s-D:                 Return to Dribbling Lisp Listener
s-E:                 Return to Lisp Top Level in Lisp Listener 1
->c-? Print out the current state of ARLO's computations.
ARLO is currently:
3:   getting the Wedging slot of {#>KYLE}
2:    while trying to compute a default for the Wedging slot of {#>KYLE}
1:     while getting the Wedging slot of {#>KYLE}
```

This is the trace produced by the WHERE *function.*

```
-> [RESUME] Perform the operation using subprimitives.
```

The subprimitives, unfortunately, merely return NIL if a slot doesn't exists. Since Kyle has no Wedging *slot, the value NIL is computed as one. But the* Wedging *slot requires – as it is defined on page 31 – a string and ARLO complains about this inconsistency.*

```
>>ARLO-Condition: The Wedging slot cannot accept the value NIL
                  (it isn't of type STRING-TYPE)
  While caching NIL on the Wedging slot of {#>KYLE}
#>SLOT-VERIFIER-FOR-WEDGING:
   Arg 0 (UNIT): {#>KYLE}
   Arg 1 (SLOT): {#>WEDGING}
   Arg 2 (VALUE): NIL
s-A, [RESUME]:     Accept the value anyway
s-B, c-?:            Print out the current state of ARLO's computations.
s-C, [ABORT]:      Return to the examining the unit {#>KYLE}
s-D:                 Return to Dribbling Lisp Listener
s-E:                 Return to Lisp Top Level in Lisp Listener 1
->c-? Print out the current state of ARLO's computations.
ARLO is currently:
2:   caching NIL on the Wedging slot of {#>KYLE}
1:    while getting the Wedging slot of {#>KYLE}
-> [RESUME] Accept the value anyway
```

And finally we get a final result, after all of our running around in the error system.

```
The Wedging slot of {#>KYLE} is:  NIL
This is justified by:
The Functional Value slot of {#>DESCRIPTION-OF-WEDGE} is:
#<DTP-COMPILED-FUNCTION WEDGE 21016762>
The To Default Value slot of {#>WEDGING} is:  #'WEDGE
```

3.6.5 Shadowing Definitions

```
Editing {#>KYLE} >>New unit
What unit would you like to edit? Brian
Description of the ARLO unit {#>BRIAN}:
Description:              The description of BRIAN was not provided.
Prototype:                {#>WINNER}
Prototype Of:
My Creator:               Ken Haase
My File Of Definition:    ARLO: SOURCES; INQUIR * >
My Time Of Creation:      Saturday the twenty-eighth of July, 1984; 12:02:02 am
Full Name:                Brian Walking-Song
Last Name:                Walking-Song
My Name:                  #>BRIAN
My To Describe Self:      #'LOOK-AT-UNIT
My To Print Self:         #'DEFAULT-UNIT-PRINTER
Personal Name:            Brian
Supervisor:               {#>CHARO}
```

Here we ask for the hacking slot of Brian, *whose prototype is* Winner. *As defined initially, the* Winner *prototype provides a different definition of* Hacking *from the default. Precisely, it asks the user for the hacking slot directly, rather than first trying to inherit it through the* Supervisor *relation.*

```
Editing {#>BRIAN} >>G -- Describe Slot Value
Which slot of {#>BRIAN} would you like to see?Hacking
What is Brian hacking?Intelligent Mystic Systems
The Hacking slot of {#>BRIAN} is:  Intelligent Mystic Systems
This is justified by:  Ken Haase said so.
```

This citation - referring to myself, the person using the program - is recorded by a dependency record which is a SLOT-CITATION-RECORD, *documented in Section 2.2.1.*

```
Editing {#>BRIAN} >>Describe [Describes the unit.. ]
Description of the ARLO unit {#>BRIAN}:
Description:            The description of BRIAN was not provided.
Prototype:              {#>WINNER}
Prototype Of:
My Creator:             Ken Haase
My File Of Definition:  ARLO: SOURCES; INQUIR * >
My Time Of Creation:    Saturday the twenty-eighth of July, 1984; 12:02:02 am
Full Name:              Brian Walking-Song
Hacking:                Intelligent Movie Systems [The value has been cached.. ]
Last Name:              Walking-Song
My Name:                #>BRIAN
My To Describe Self:    #'LOOK-AT-UNIT
My To Print Self:       #'DEFAULT-UNIT-PRINTER
Personal Name:          Brian
Supervisor:             {#>CHARO}
```

Let's look at where Brian's description got its replacement hacking definition – which asked us for a value directly – from. The shadowed definition of hacking you remember – from Page 32 – looks on the prototypes of the unit it is accessing. So we edit the prototype of #>BRIAN...

```
Editing {#>BRIAN} >>Edit
Which slot of {#>BRIAN} would you like to edit?Prototype
Description of the ARLO unit {#>WINNER}:
Description:                Somone who doesn't always follow their supervisor.
Prototype:                  {#>PERSON}
Prototype Of:               {#>BRIAN}
My Creator:                 Ken Haase
My File Of Definition:      ARLO: SOURCES; INQUIR * >
My Time Of Creation:        Saturday the twenty-eighth of July, 1984; 12:02:00 am
My Name:                    #>WINNER
My Specific Type:           {#>WINNER-TYPE}
My To Describe Self:        #'LOOK-AT-UNIT
My To Print Self:           #'DEFAULT-UNIT-PRINTER
Shadowed Hacking Definition: {#>HACKING-0}
                            [And here is the shadowed definition of Hacking. ]
```

We can look deeper into this new definition of hacking by editing its description. Every ARLO slot, since it is explicitly described in ARLO, is accessible in this way.

```
Editing {#>BRIAN} >>Edit
Which slot of {#>BRIAN} would you like to edit? Shadowed Hacking Definition
Description of the ARLO slot {#>HACKING-0}:
Description:              The description of HACKING-0 was not provided.
Prototype:                {#>HACKING}
Prototype Of:
To Default Value:         ASK-USER-FOR-SLOT
Makes Sense For:          {#>WINNER-TYPE}
Data Type:                {#>STRING-TYPE}
My Creator:               Ken Haase
My File Of Definition:    ARLO: KBases; INQUIR.BIN.NEWEST
My Time Of Creation:      Saturday the sixth of April, 1985; 9:11:58 am
Actual Get Value:         #'CHECK-VALUE
My Name:                  #>HACKING
My To Describe Self:      #'LOOK-AT-SLOT
My To Print Self:         #'DEFAULT-UNIT-PRINTER
To Cache Value:           #'TYPED-CACHE
To Describe Value:        (LAMBDA (IGNORE) IGNORE)
To Get Value:             #'TYPED-DEFAULTING-GET
To Help Find Value:       #'EVAL-READ-AS-ESCAPE
To Print Value:           #'GPRINTC
To Prompt For Value:      #'CUTE-PROMPT-FOR-VALUE
To Read Value:            #'READLINE
To Verify Type:
#'CORE:INQUIR:SLOT-VERIFIER-FOR-HACKING-0
Editing {#>HACKING-0} >>Quit
Finished editing {#>HACKING-0}
Editing {#>WINNER} >>Quit
Finished editing {#>WINNER}
Editing {#>BRIAN} >>Quit
Finished editing {#>BRIAN}
Back to editing {#>KYLE}
```

3.6.6 Modifying our language

```
Editing {#>KYLE} >>New unit
What unit would you like to edit?Alice
Description of the ARLO unit {#>ALICE}:
Description:              The description of ALICE was not provided.
Prototype:                {#>PERSON}
Prototype Of:
My Creator:               Ken Haase
My File Of Definition:    ARLO: SOURCES; INQUIR * >
My Time Of Creation:
Saturday the twenty-eighth of July, 1984; 12:02:02 am
Editing {#>ALICE} >>Describe [Describes the unit.. ]
Description of the ARLO unit {#>ALICE}:
Description:              The description of ALICE was not provided.
Prototype:                {#>PERSON}
Prototype Of:
My Creator:               Ken Haase
My File Of Definition:    ARLO: SOURCES; INQUIR * >
My Time Of Creation:      Saturday the twenty-eighth of July, 1984; 12:02:02 am
Full Name:                Alice Adams
Last Name:                Adams
My Name:                  #>ALICE
My To Describe Self:      #'LOOK-AT-UNIT
My To Print Self:         #'DEFAULT-UNIT-PRINTER
Personal Name:            Alice
Supervisor:               {#>RODGERS}
Working In Field:         Emotional Analouge Robots
                          [This is the value defaulted earlier in the example (Section 3.6.3, Page 43). ]
```

Now we will change the definition of how defaults for `Working-In-Field` *should be computed, and this modification will make previous derivations – based on a different definition – invalid.*

```
Editing {#>ALICE} >>New unit
What unit would you like to edit?Working in Field
Description of the ARLO slot {#>WORKING-IN-FIELD}:
Description:              This is the field a person is working in.
Prototype:                {#>PERSON-SLOT}
Prototype Of:
To Default Value:
#>THE-VALUE-OF-THE-HACKING-OF-THE-SUPERVISOR-OF
Makes Sense For:          {#>PERSON-TYPE}
Data Type:                {#>STRING-TYPE}
My Creator:               Ken Haase
My File Of Definition:    ARLO: SOURCES; INQUIR * >
My Time Of Creation:      Saturday the twenty-eighth of July, 1984; 12:01:58 am
Actual Get Value:         #'CHECK-VALUE
High Level Definition:    #'CORE:INQUIR:THE-VALUE-OF-THE-HACKING-OF-THE-SUPERVISOR-OF
My Name:                  #>WORKING-IN-FIELD
My To Describe Self:      #'LOOK-AT-SLOT
My To Print Self:         #'DEFAULT-UNIT-PRINTER
To Cache Value:           #'TYPED-CACHE
To Describe Value:        (LAMBDA (IGNORE) IGNORE)
To Get Value:             #'TYPED-DEFAULTING-GET
To Print Value:           #'GPRINTC
To Process Slot:          #'DEFAULT-PROCESS-SLOT
To Verify Type:           #'CORE:INQUIR:SLOT-VERIFIER-FOR-WORKING-IN-FIELD
```

We define the new defaulting method by using the automatic coder SLOT-COMPOSITION. *Just as we defined* #>HACKING-0, *we define the new* #>WORKING-IN-FIELD *to inherit from the* #>HACKING *slot two supervisors away.*

```
Editing {#>WORKING-IN-FIELD} >>Set Slot Value
Which slot of {#>WORKING-IN-FIELD} would you like to edit? To Default Value
What value would you like in the {#>To-Default-Value} slot of {#>WORKING-IN-FIELD}:
(Slot-Composition (list #>Hacking #>Supervisor #>Supervisor))
Editing {#>WORKING-IN-FIELD} >>Describe
Description of the ARLO slot {#>WORKING-IN-FIELD}:
Description:              This is the field a person is working in.
Prototype:                {#>PERSON-SLOT}
Prototype Of:
To Default Value:
#>THE-VALUE-OF-THE-HACKING-OF-THE-SUPERVISOR-OF-THE-SUPERVISOR-OF
Makes Sense For:          {#>PERSON-TYPE}
Data Type:                {#>STRING-TYPE}
My Creator:               Ken Haase
My File Of Definition:    ARLO: SOURCES; INQUIR * >
My Time Of Creation:      Saturday the twenty-eighth of July, 1984; 12:01:58 am
Actual Get Value:         #'CHECK-VALUE
High Level Definition:
#'CORE:INQUIR:THE-VALUE-OF-THE-HACKING-OF-THE-SUPERVISOR-OF-THE-SUPERVISOR-OF
[The new high level definition, all compiled..]
My Name:                  #>WORKING-IN-FIELD
My To Describe Self:      #'LOOK-AT-SLOT
My To Print Self:         #'DEFAULT-UNIT-PRINTER
To Cache Value:           #'TYPED-CACHE
To Decache Value:         #'REMOVE-VALUE
To Describe Value:        (LAMBDA (IGNORE) IGNORE)
To Get Value:             #'TYPED-DEFAULTING-GET
To Print Value:           #'GPRINTC
To Process Slot:          #'DEFAULT-PROCESS-SLOT
To Verify Type:
#'CORE:INQUIR:SLOT-VERIFIER-FOR-WORKING-IN-FIELD
Editing {#>WORKING-IN-FIELD} >>Quit
Finished editing {#>WORKING-IN-FIELD}
Back to editing {#>ALICE}
```

Since we changed the way Working-In-Field *is defined, any values which were defaulted in the old way should be invalidated. Let's look back to Alice's description to see if this is indeed the case.*

```
Editing {#>ALICE} >>Describe
Description of the ARLO unit {#>ALICE}:
Description:              The description of ALICE was not provided.
Prototype:                {#>PERSON}
Prototype Of:
My Creator:               Ken Haase
My File Of Definition:    ARLO: SOURCES; INQUIR * >
My Time Of Creation:      Saturday the twenty-eighth of July, 1984; 12:02:02 am
Full Name:                Alice Adams
Last Name:                Adams
My Name:                  #>ALICE
My To Describe Self:      #'LOOK-AT-UNIT
My To Print Self:         #'DEFAULT-UNIT-PRINTER
Personal Name:            Alice
Supervisor:               {#>RODGERS}
                          [And the cached Working-In-Field has indeed disappeared. ]
```

Let's regenerate it.

```
Editing {#>ALICE} >>G -- Describe Slot Value
Which slot of {#>ALICE} would you like to see? Working In Field
The Working In Field slot of {#>ALICE} is: Robots
```

You can see from the justifications of the value that it did the right thing, looking at Rodger's supervisor and getting her Hacking *slot.*

```
This is justified by:
The Hacking slot of {#>CALVIN} is: Robots
The To Get Value slot of {#>HACKING} is: #'TYPED-DEFAULTING-GET
The Supervisor slot of {#>RODGERS} is: {#>CALVIN}
The Supervisor slot of {#>ALICE} is: {#>RODGERS}
The To Get Value slot of {#>SUPERVISOR} is: #'TYPED-DEFAULTING-GET
The To Default Value slot of {#>WORKING-IN-FIELD} is:
#>THE-VALUE-OF-THE-HACKING-OF-THE-SUPERVISOR-OF-THE-SUPERVISOR-OF
The To Get Value slot of {#>TO-DEFAULT-VALUE} is: #'TYPED-DEFAULTING-GET
```

And finally, we check that the value we have generated has been appropriately cached....

```
Editing {#>ALICE} >>Describe
Description of the ARLO unit {#>ALICE}:
Description:              The description of ALICE was not provided.
Prototype:                {#>PERSON}
Prototype Of:
My Creator:               Ken Haase
My File Of Definition:    ARLO: SOURCES; INQUIR * >
My Time Of Creation:      Saturday the twenty-eighth of July, 1984; 12:02:02 am
Full Name:                Alice Adams
Last Name:                Adams
My Name:                  #>ALICE
My Ip Describe Self:      #'LOOK-AT-UNIT
My To Print Self:         #'DEFAULT-UNIT-PRINTER
Personal Name:            Alice
Supervisor:               {#>RODGERS}
Working In Field:         Robots
                          [And, of course, the value is cached again. ]
```


Chapter 4
An Example Introspection

Chapter 4
An Example: Introspection

This chapter describes an automatic explanation system — implemented in and for ARLO — that examines a collection of ARLO units and generates a structured English explanation of them. These units would typically describe some particular domain or embedded representation language, and be organized to aid users or programmers introducing themselves to the domain or language. The system analyzes a collection of units by trying to extract their salient features as an organizational focus for its explanation. Unfortunately, since the text it generates is primitively template driven (currently), the system does not — at this time — use these extracted features as the focus for discourse or individual explanations.

This is an example of the sort of general self-referential facility which users may implement in ARLO. With something comparable to this explanation system, a user need merely point at some collection of units and ask "Explain this" to acquire an organized explanation capturing whatever special "observable" structure the units possessed.

4.1 Explanation Structures

The explanation system takes the collection of units handed to it and generates another set of units called an *explanation structure* describing them. This structure is a hierarchy of explanations, each level of which partitions the set of units over one of a number of possible relationships. These possible relationships are the possible *structural slots* of a given explanation, and defaults to the union of a collection of system defaults and the slot descriptions in the set of units being explained.

The explanation process takes the set of units being explained and generates a partion of it for each structural slot. The resulting partitions — one over each structural slot/relation — are then compared, and the slot whose partition contains the largest subgroups is selected as the focus of the explanation. The intuition this supports is that the organizational focus for an explanation of some collection of units should be the relation which organizes those units into the biggst "chunks". If a user doesn't like the partition chose at one level though, the explanation structure can be directly altered to focus on another divisive relation.

For each of the subgroups in the partition selected, a *sub explanation* is generated, whose relevant units are the elements of the subgroup, and whose structural slots are inherited from the original explanation, modulo the slot partitioned over. The explanation mechanism then recurs on these sub explanations, stopping when the *section size* — the number of units being explained by a given chunk of structure — drops below some explanation-wide threshold for specialization of sections.

The explanation structure produced by this process may then be passed to a text-generator, a graphical exploration environment, or even a theory-making mechanism trying to classify regularities among generated or accumulated ARLO structures.

4.2 Textual Generation

Textual generation from the explanation structure currently produces organized and formatted [15] output, appropriately sectionized and structured so as to produce readable, structured output. On both the level of describing individual units and organizing explanations into sections, the documentation process is data-directed by reference to descriptions in ARLO.

For individual units, their english explanation is provided by calling a LISP function on the unit's `#>My-Scribe-To-Document-Self` slot, which is inherited (by default) over the `#>Prototype` relation. (Of course this inheritance mechanism may be *shadowed* arbitrarily.) These inherited description functions will produce useful — for human consumption — descriptive text. Slot definitions, automatically coded LISP functions, ARLO *coders*, and user defined functions are all described in different ways so as to provide appropriate information to the user. In a more advanced form, the documentation system might take into account interests of the user, information already related, and "trivial" aspects of the description (for instance, expected colors, planets, languages, etc).

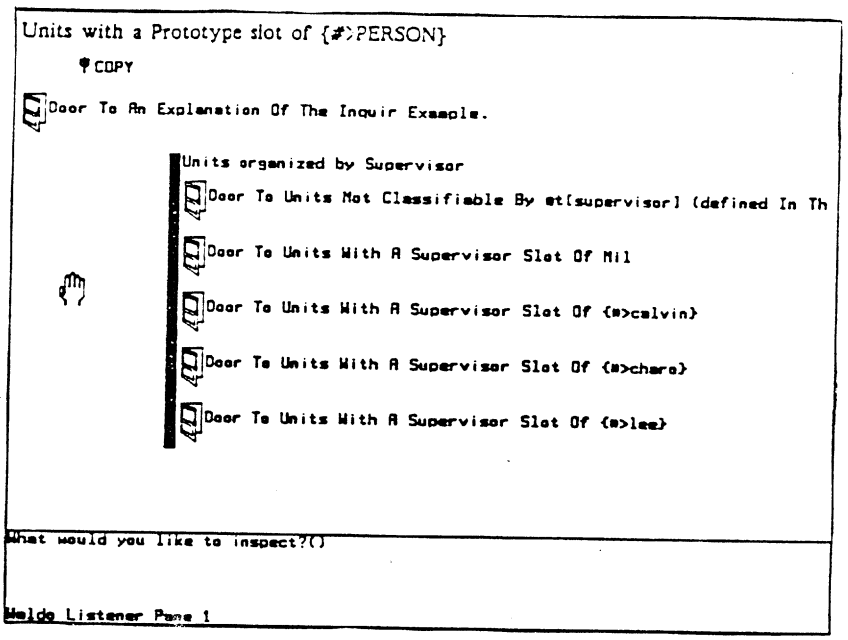
For every node in the explanation structure which has a relational focus — which *partitions* a set of units over some particular slot — the manner of sectionization (determining section titles, order of sections, discourse restrictions of sub-sections, etc) is determined by the slot being partitioned over (taken as the organizational focus of the explanation). For instance, relations which are posited by the user as *hierarchical* [16] are ordered into sections by a breadth first enumeration of the hierarchy they define. Other slots may organize their documented partitions on ages, execution speeds, size, or frequencies of appearance of their associated values.

4.3 Graphical Presentation

The explanation structure generated by the system can also be hooked up to a graphical interface for examining nested structures. Particularly, ARLO's generated explanation structure has been hooked up to the `Information-Waldo`, a gestural interface for manipulating abstract objects in an *information space*. This information space is constructed of interconnected rooms containing objects with various properties

[15] The text produced is either formatted for the terminal or (if going to a file) for some appropriate text formatting program.

[16] We could imagine the discovery of such relational properties (like being hierarchical) being made by an intelligent program generalizing from examples. [Cha83] describes a system which does just this sort of relational generalization from examples in the "world".



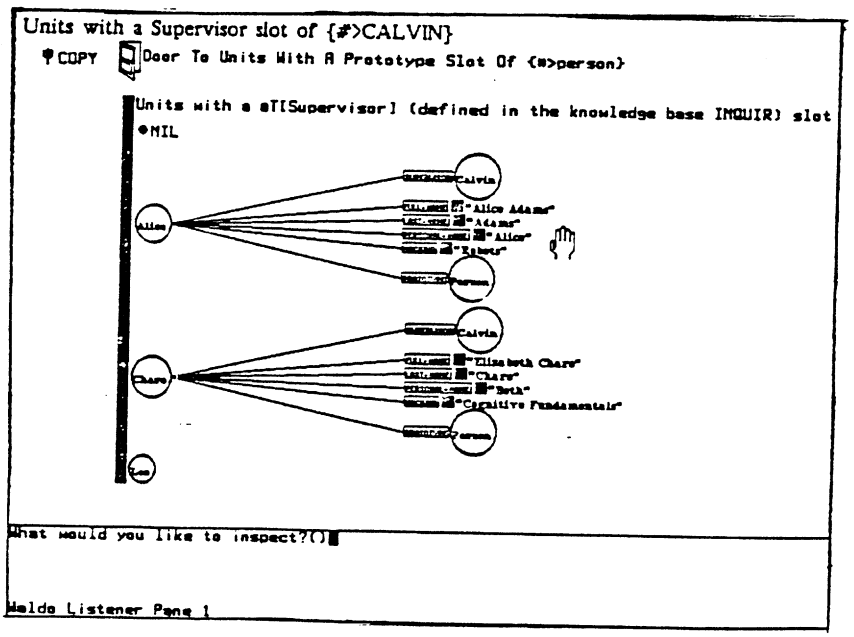


Figure 4-1. Using the Information Waldo in the INQUIR museum structure.

and powers. A user wields the information waldo to explore this network of rooms and manipulate their contents, moving from place to place and description to description by physically immediate gesture and action.

A user moves a hand shaped grip across a flat surface to move a hand on the screen — and "information waldo" existing in an abstract "information space" — from location to location. Stylized gestures of grasping, pointing, or squeezing are detected by the grip and cause the hand on the screen to manipulate the objects it is moving among. To examine an ARLO description with the information waldo, you merely pick up the roll-shaped description and squeeze it: its relations leap out from its body; to retract a relation, you rub out the label attaching it to the description; to move from one room to another, simply put your hand through an open door, and the new room opens itself up on the display. This gestural interface to ARLO is used as the basis of an explanation-based browser for ARLO structures. Structured explanations of collections of ARLO units are used in the design and construction of multi-room *museums* portraying and describing them.

The explanation structure produced for ARLO descriptions can generate a museum of the units explained; this museum consists of a network of rooms reflecting the connections and groupings of the explanation structure. A user exploring some particular implementation or representation with this facility can use spatial metaphors to organize her understanding. In a more advanced form, a sophisticated interface would design the museum with the explicit goal of providing such metaphors and mnemonic arrangements. Figure 4-1 shows the museum interface being used to explore the INQUIR knowledge base of the previous example.

4.4 An Explanation of the INQUIR system

The following is an automatically generated explanation for the INQUIR example of the previous chapter. It was produced by applying the above explanation system to the in-core implementation of the INQUIR system (determined by all of the units in the INQUIR knowledge base).

These units are best organized by the **Prototype** relation.

4.4.1 Units without any prototype.

Person is a protypical person description in the "INQUIR" knowledge base. This is the prototypical person.

4.4.2 Units with a prototype of **Hacking**

`Hacking` (as defined by `HACKING-0`) is a slot which accepts values of type *String Type* and makes sense for units of type *Winner Type*. The description of HACKING-0 was not provided. Its value defaults by the function ARLO:QUESTION-6, which:

Ask the user a question by:
(FORMAT QUERY-IO "What is a hacking on?" (GET-VALUE UNIT `#>PERSONAL-NAME`*)*

4.4.3 Units with a prototype of **Hand Coded Function**

These units are best organized by the **Prototype** relation.

4.4.4 Units without any prototype.

Person is a protypical person description in the "INQUIR" knowledge base. This is the prototypical person.

4.4.5 Units with a prototype of **Hacking**

`Hacking` (as defined by `HACKING-0`) is a slot which accepts values of type *String Type* and makes sense for units of type *Winner Type*. The description of HACKING-0 was not provided. Its value defaults by the function ARLO:QUESTION-6, which:

Ask the user a question by:
(FORMAT QUERY-IO "What is a hacking on?" (GET-VALUE UNIT `#>PERSONAL-NAME`*)*

4.4.6 Units with a prototype of **Hand Coded Function**

`DATA-TYPE-GENERATOR` is a user defined lisp function which has an argument list of (UNIT SLOT), and is documented as: *"Looks through the prototypes of a slot for its data-type".*

`DEFAULT-DESCRIPTION-GENERATOR` is a user defined lisp function which has an argument list of (IN-UNIT IGNORE), and is documented as: *"This generates a description excuse.".*

`FIND-HACKING-SLOT` is a user defined lisp function which has an argument list of (UNIT IN-SLOT), and is documented as: *"Looks for a replacement hacking definition in a persons prototypes.".*

`GENERATE-EXPLANATION-TITLE` is a user defined lisp function which has an argument list of (EXPLANATION IGNORE), and is documented as: *"Generates an title for a given explanation.".*

`TO-GENERATE-LAST-NAME` is a user defined lisp function which has an argument list of (UNIT IGNORE), and is documented as: *"Extracts a person's last name from her full name.".*

`TO-GENERATE-PERSONAL-NAME` is a user defined lisp function which has an argument list of (UNIT IGNORE), and is documented as: *"Extracts a person's first name from her full name.".*

`WEDGE` is a user defined lisp function which has an argument list of (UN SL), and is documented as: *"Recurses infinitely.".*

4.4.7 Units with a prototype of **Person Slot**

`Full-Name` is a slot which accepts values of type *String Type* and makes sense for units of type *Person Type*. This is the full, formal name of a person. Its value defaults by the function ARLO:QUESTION-3, which:

Ask the user a question by:
(FORMAT QUERY-IO " &What is the full name of the person described by a?" UNIT)

`Hacking` is a slot which accepts values of type *String Type* and makes sense for units of type *Person Type*. This is what a person is hacking on. Its value defaults by the function ARLO:TRY-AND-TRY-AGAIN-1, which:

Tries to compute a value by two distinct methods:
Searches through the CORE:INQUIR:SUPERVISOR slots of a unit for a value.

Ask the user a question by:
(FORMAT QUERY-IO "What is a hacking?" (GET-VALUE UNIT `#>PERSONAL-NAME`*)*

`Last-Name` is a slot which accepts values of type *String Type* and makes sense for units of type *Person Type*. This is the last name of a person. Its value defaults by the function ARLO:TO-GENERATE-LAST-NAME, which:

Extracts a person's last name from her full name.

`Personal-Name` is a slot which accepts values of type *String Type* and makes sense for units of type *Person Type*. This is the informal name of a person. Its value defaults by the function ARLO:TO-GENERATE-PERSONAL-NAME, which:

Extracts a person's first name from her full name.

`Supervisor` is a slot which accepts values of type *Person Type* and makes sense for units of type *Person Type*. This is the supervisor of a person. Its value defaults by the function ARLO:QUESTION-4, which:

Ask the user a question by:
(FORMAT QUERY-IO " &Who is a hacking for?" (GET-VALUE UNIT `#>PERSONAL-NAME`*)*

`Wedging` is a slot which accepts values of type *String Type* and makes sense for units of type *Person Type*. This breaks. Its value defaults by the function ARLO:WEDGE, which:

Recurses infinitely.

`Working-In-Field` is a slot which accepts values of type *String Type* and makes sense for units of type *Person Type*. This is the field a person is working in. Its value defaults by the function THE-VALUE-OF-THE-HACKING-OF-THE-SUPERVISOR-OF, which:

Gets the CORE:INQUIR:HACKING of the CORE:INQUIR:SUPERVISOR of some unit.

4.4.8 Units with a prototype of **Person**

These units are best organized by the **Supervisor** relation.

4.4.8.1 People without any supervisor

Susan Calvin is working on **Robots**.

4.4.8.2 People working for Susan Calvin

Alice Adams is working on **Robots** for *Susan Calvin.*
Elizabeth Charo is working on **Cognitive Fundamentals** for *Susan Calvin.*
Pat Lee is working on **Engineering Design** for *Susan Calvin.*
Robert Rodgers is working on **Emotional Analouge Robots** for *Susan Calvin.*

4.4.8.3 People working for Elizabeth Charo

Arthur Pendragon is working on **Fantasy Games** for *Elizabeth Charo.*

4.4.8.4 People working for Pat Lee

Kyle O'Shea is working on **Engineering Design** for *Pat Lee.*

4.4.8.5 Units not classifiable by `Supervisor`

Winner is a protypical person description in the "INQUIR" knowledge base. Somone who doesn't always follow their supervisor.

4.4.9 Units with a prototype of **Slot**

`Person-Slot` is a slot which accepts values of type *Any Type* and makes sense for units of type *Person Type*. This is the prototypical slot which attaches to people.

4.4.10 Units with a prototype of **Shadow Slot**

`Shadowed-Hacking-Definition` is a slot which accepts values of type *Slot Type* and makes sense for units of type *Slot Type*. This is a shadowed definition for hacking. Its value defaults by the function ARLO:FIND-HACKING-SLOT, which:

Looks for a replacement hacking definition in a persons prototypes.

4.4.11 Units with a prototype of **Type**

`Person-Type` specifies a class of LISP objects which are classified by `Unit-Type` and which additionally satisfy the predicate `TEST-4` (documented as *"An arbitarily hairy test."*). This is a type satisifed by any unit inheriting from Person.

`Winner-Type` specifies a class of LISP objects which are classified by `Unit-Type` and which additionally satisfy the predicate `PROTOTYPE-OF-WINNER?` (documented as *"Checks to see if a unit inherits from CORE:INQUIR:WINNER via CORE:PROTOTYPE."*). This is a type satisifed by units inheriting (via the Prototype relation) from the unit Winner.

4.4.12 Units with a prototype of **Winner**

Brian Walking-Song is working on **Intelligent Mystic Systems** for *Elizabeth Charo*.

Chapter 5
Conclusion

The preceding chapters may have seemed like an attempt to 'sell' ARLO as a panacea for all one's representation problems. Unfortunately, when pushed to the limit, ARLO broke down for fairly fundamental reasons. This conclusion examines those reasons and presents arguments for which of ARLO's ideas are worth keeping in new implementations, and which caused basic problems.

The version of ARLO described here was developed largely in the summer of 1983 and the spring of 1984. In the fall of 1984, a discovery program implemented in ARLO (Cyrano-0) acheived about half of the results of AM and Eurisko in elementary mathematics, discovering the notion of number and synthesizing operations such as multiplication over numbers. Due to an insufficent theory for the representation of inverses, the step to factorization and AM's subsequent discoveries in elementary number theory were not acheived. However, this work did reveal some fundamental properties of discovery programs, which are described in [Haa86b].

At the same time that the initial development of Cyrano-0 was proceeding, Dave McDonald and his students at UMASS-Amherst were using ARLO as the representational backbone for generating English text (using McDonald's MUMBLE [McD83]) for an 'intelligent encyclopedia. This work is described in [MP84].

Implementing Cyrano-0 in ARLO revealed a variety of cumbersome properties of ARLO; in the late winter and early spring of 1985, an effort to reimplement ARLO was undertaken. The key points of this implementation (in particular its differences with respect to the ARLO described here) are presented below. A manual for this version of ARLO is available as [Haa86a]. Work with this new ARLO, however, revealed deep problems (for purposes of automated discovery programs) in the 'frame-slot' orientation of ARLO. These problems, broached in detail in [Haa86c], are also sketched below.

Despite these criticims, many of the ideas behind ARLO are still neccessary constituents of AI languages. The ability to refer to abstract descriptions of properties allows programs to easily use meta-knowledge in describing their own constructions. In particular, knowledge about the semantic restrictions on properties allows a program to understand its own representation in a general way.

5.1 Flaws in ARLO

In developing Cyrano-0, ARLO was found cumbersome for a variety of reasons. Some of the reasons are endemic to RLL's in general and will be described in Section 5.2; others are particular to the implementation described in the preceding chapters. These problems are the topic of this section.

Most of the problems in using ARLO were not real problems of expressiveness; since a user could encode arbitrary patterns of activity into LISP procedures. ARLO was arbitrarily expressive in a weak way. The problems were rather problems of perspicuity; in order to say certain things that one wished to say, it was neccessary to descend into LISP. The magic grab-bag of LISP extensions became a cloak over the operation of the system, requiring that each modification and analysis module have special properties for special casing various opaque extensions of ARLO.

This problem revealed itself in two particular components of ARLO: the dependency network and the accretion of slot behaviours. In each of these, the usefulness and extensibility of the module was hampered by the lack of sufficiently explicit representations of ARLO's implementation; the module had to be extensively special-cased to handle opaquely distinct representational constructs.

5.1.1 Flaws in the Dependency Network

The dependency network, implemented in LISP Machine flavors, suffered from a variety of flaws. Most had to do with the opaqueness of the dependency implementation; user interface utilities, debuggers, and special network updating code had to deal with the vagaries of message passing in LISP as well as ARLO's unit-slot representation. There was also the familiar crossbar problem of introducing new sorts of dependencies; in order to introduce a new type of dependency, it was neccessary to determine the interaction of the new dependency type with all existing dependency types and tools. The standard protocol for invalidation helps this process, but managing details is still difficult. In particular, a user interface must special case its presentations for each different sort of dependency.

The general result of these opacities in the dependency network is the same as opacity anywhere; a significant increase in the amount of LISP code and programming required rather than a modest increase in the amount of specified representation. We would like to be able to extend and use the dependency network in much the same way as we use ARLO units. Unfortunately, dependency records are not units but are special purpose LISP data structures encumbered with methods and procedural semantics couched in LISP Machine LISP.

The obvious solution to this, implemented in [Haa86a], is to make dependency records into units. In [Haa86a] the values of slots may actually be 'value descriptions' which go through another level of interpretation to get 'actual values', but which provide useful information about the status of the value (where it came from, how reasonable it is, etc). These values are similar to the 'active values' of Loops [BS83] CYC [LSP85]; the are annotated values about which arbitrary properties may be stated or inferred.

5.1.2 Flaws in Combining Slot Actions

The flaws described in this section arise from ARLO's answer to the question: "How do we add new behaviours to a slot or type of slot" In ARLO, the way to add behaviours is to write LISP code which will execute the behaviours. The way to modify behaviours (much simpler) is to simply use one function instead

of another as one slot of the abstract slot description being modified. This is made possible by the use of reflexive operators. For adding behaviours, the presence of reflexive operators makes writing general code simpler; we may simply say "do the inversion side-effects of the slot" rather than having to specify whatever particular function implements "do the inversion side-effects of the MOTHER slot." However the problem is that new behaviours — specified in LISP — the become largely opaque to the other behaviours and functions of the system.

The one point where this problem became most obvious in ARLO was in attempting to maintain a distinction between 'syntactic' and 'semantic' information about slots. For instance, to implement many-to-many relations with slots, the values of slots must be interpreted as multiple values; the content of a slot is then (say) a list. But the semantic restrictions placed on a slot (properties like Makes-Sense-For and Data-Type) should apply to the individual elements of the list, rather than the list itself. This distinction (neccessary due to the focus of ARLO on single-valued slots[17]) is impossible to patch by using prototype inheritance for abstraction, for we wish to speak of semantic AND syntactic inheritance. Thus we can say that the Children slot is syntactically a set and semantically only accepts human beings on both ends (as attachement and value). We wish these properties to inherit differently. In ARLO, however, this was impossible.

The solution to this particular problem in [Haa86a] is to simply have two different inheritance relations and two distinct levels of operation for fetching slots: an implementation level of *accessing* a slot and an interpretation level of *getting* slots. The first level is a 'syntactic' level; the second level is 'semantic.'

This solution is effective but introduces some problems of its own. In particular, though we would like 'syntax' and 'semantics' to be orthogonal, they turn out not to be. When a new syntactic or semantic primitive is introduced into the language, provision must often be made in the 'other half' of the implementation. This is better than in the implementation described in this document (where adding a non-primitive construction involves combining LISP code from several places) but still not ideal. An argument that this problem is endemic to RLLs is offered in Section 5.2.

5.2 Why RLL's are no good

All of the problems described in the previous section arise from the opacity of extensions to the RLL. These opacities result from the inclusion of arbitrary LISP code in the specification of slot behaviours. In each case, in [Haa86a] the problem was resolved by factoring out the LISP code into primitives in the representation. Thus the methods for handling dependency propogation were assigned to properties of value descriptions and the discinction between syntactic specification and semantic specification moved from implicit specification in LISP code to a distinction between hierarchies in the representation. We might hope that — given enough such migrations — that the right 'primitives' would be found to avoid any need to escape to LISP.

Unfortunately, we already know — in some sense — what this 'right' set of primitives should be: it's called a programming language. Users of RLLs are forced into LISP (and therefore weaken the utility of the RLL) when they need to do something which the RLL (as given) cannot adequately express. *A sufficently powerful RLL is a full-fledged programming language.* It must be — however — a programming language

[17]ARLO might be criticized for this basic assumption, but the problem is that *any* basic assumption of the language may be 'short circuited' only by descending into the murky opaqueness of LISP code.

which has a manipulable and perspicuous representation of itself. 'Limited RLLs,' like ARLO and the language described in [Haa86a], are useful for particular applications but eventually lose generality when users require the full power of a programming language. For instance, slots defining individual slot actions are fine until one wishes to compose new actions to existing ones. At this point, since the notion of a slot is a weakened and limited version of the notion of a function, to define the composition of slot executions, the user must escape to LISP where she can use the full notion of functional composition and sequencing.

The solution to this problem, as I suggest in [Haa86c], is to develop a programming language with the self-descriptive capacity of RLLs. In brief, this language is a higher order language similar to FP [Bac78] with inferred typing of functions (much as in ML [Mil78] and the addition of a special class of functions — called mutable mappings — which replace the functionality of slots and properties. The function MAKE-MUTABLE constructs a mutable function which is simply a pairwise mapping of objects. The function MUTATOR returns a procedure for storing mappings for the mutable function. For example, the following uses mutable operations to define the COLOR function and set the color of a few objects.

```
(define color (make-mutable))
COLOR
(color 'apple)
<UNKNOWN> ; Indicates a value with no mapping.
(define define-color! (mutator color))
DEFINE-COLOR
(define-color! 'apple 'red)
<UNKNOWN> ; the previous return value.
(define-color! 'orange 'orange)
<UNKNOWN>
(color 'apple)
RED
(color 'orange)
ORANGE
```

These mutatable functions can be combined with higher order operators, like COMPOSE or RESTRICT-RANGE. Here we defined a special subset of colors and compose this with a class of fruits:

```
(define real-colors (set-of '(red green blue yellow orange pink)))
REAL-COLORS  ; The value of this is a type.
(define real-color (restrict-range color real-colors))
REAL-COLOR
(define fruit (make-mutable))
FRUIT
(define fruit-color (compose fruit real-color))
FRUIT-COLOR
```

So defined, we can set and access the color of fruits by using the procedures we have defined and their

associated mutators.

```
((mutator fruit) 'apple-tree 'apple)
<UNKNOWN>
(fruit-color 'apple-tree)
RED
```

Knowledge about procedures can be accessed by other procedures, in particular, DOMAIN and RANGE.

```
(domain color)
#[ANYTHING]
(range color)
#[ANYTHING]
(range real-color)
#[One of RED GREEN BLUE YELLOW ORANGE PINK]
(range fruit-color)
#[One of RED GREEN BLUE YELLOW ORANGE PINK]
```

By defining all of ones representational constructs in this way, the expressive power of our representation language is nearly equal to that of LISP-like languages while still giving us the power of an RLL.

5.3 Why RLL's Aren't So Bad

In the previous section, an argument was introduced for a new sort of representation language language, criticizing fundamental flaws in most representation language languages to date. An important point to make however, is that the criticism applies primarily to programs which must learn by accquiring new representations and definitions. For implementing any given AI program — capturing a given domain's expertise — an RLL provides a powerful toolkit for building a specially tailored representation. Only when new tools must be built do traditional RLLs falter or fail.

In conclusion, the reasons for wanting to have an RLL are sustained; self-debugging, self-explanation, and self-modification are greatly enhanced by having a representation of the representation being used. Unfortunately, these reasons are countervailed as the expressive demands on the language require escape to a 'real' programming language. The solution — it then seems — must be to make an RLL which is a 'real' programming language.

Chapter A-1
An ARLO 'Explanation'

These units are best organized by the **My File Of Definition** relation.

A-1.1 Units defined in Arlo: SOURCES; BOOT

These units are best organized by the **Makes Sense For** relation.

A-1.1.1 Units with a Makes Sense For slot of `Any-Type`

The unit **Defaulting Slot** is defined in the knowledge base `Core`. This is the prototype for slots which default their values.

The unit **Generic Slot** is defined in the knowledge base `Core`. This is a prototypical "generic" slot which looks for local slot definitions on each unit.

The unit **Primitive Slot** is defined in the knowledge base `Core`. This is the simplest prototype slot.

The unit **Prototype** is defined in the knowledge base `Core`. This is a unit's prototype.

A-1.1.2 Units with a Makes Sense For slot of `Slot-Type`

These units all have PROTOTYPE slots of `Slot`.

These units are best organized by the **Data Type** relation.

Units with a Data Type slot of `Function-Type`

These units are best organized by the **To Default Value** relation.

Units with a To Default Value slot of #'DECACHE-FINDER

`To-Decache-Value` is a slot which accepts values of type *Function Type* and makes sense for units of type *Slot Type*. This is a slot's function for invalidating it's value on a unit. Its value defaults by the function ARLO:DECACHE-FINDER, which:

This finds the deaching function for a unit by looking through its prototypes.

Units with a To Default Value slot of #'DONT-DEFAULT-SLOT

`To-Default-Value` is a slot which accepts values of type *Function Type* and makes sense for units of type *Slot Type*. This is the function for computing the value of a slot at need. Its value defaults by the function ARLO:DONT-DEFAULT-SLOT, which:

Signals an error if called to default a value.

Units with a To Default Value slot of #'FIND-VALUE

`Actual-Put-Value` is a slot which accepts values of type *Function Type* and makes sense for units of type *Slot Type*. This is a slot's function for "physically" depositing its value. Its value defaults by the function ARLO:FIND-VALUE, which:

Look through the prototoypes of a unit for a particular slot.

`To-Cache-Value` is a slot which accepts values of type *Function Type* and makes sense for units of type *Slot Type*. This is a slot's function for caching its value. Its value defaults by the function ARLO:FIND-VALUE, which:

Look through the prototoypes of a unit for a particular slot.

`To-Get-Value` is a slot which accepts values of type *Function Type* and makes sense for units of type *Slot Type*. This is a slot's procedure for fetching its value. Its value defaults by the function ARLO:FIND-VALUE, which:

Look through the prototoypes of a unit for a particular slot.

`To-Process-Slot` is a slot which accepts values of type *Function Type* and makes sense for units of type *Slot Type*. This is a slot's function for transforming its description into "print-queue" form. Its value defaults by the function ARLO:FIND-VALUE, which:

Look through the prototoypes of a unit for a particular slot.

`To-Put-Value` is a slot which accepts values of type *Function Type* and makes sense for units of type *Slot Type*. This is a slot's procedure for storing a value. Its value defaults by the function ARLO:FIND-VALUE, which:

Look through the prototoypes of a unit for a particular slot.

`To-Retract-Value` is a slot which accepts values of type *Function Type* and makes sense for units of type *Slot Type*. This is a slots procedure for removing its value. Its value defaults by the function ARLO:FIND-VALUE, which:

Look through the prototoypes of a unit for a particular slot.

Units with a To Default Value slot of #'TO-GENERATE-SLOT-DESCRIBER

`To-Describe-Value` is a slot which accepts values of type *Function Type* and makes sense for units of type *Slot Type*. This is a slot's function for describing its value. Its value defaults by the function ARLO:TO-GENERATE-SLOT-DESCRIBER, which:

Generates a function for describing a slot's value.

Units with a To Default Value slot of #'TO-GENERATE-SLOT-PRINTER

`To-Print-Value` is a slot which accepts values of type *Function Type* and makes sense for units of type *Slot Type*. This is the function for printing the value of this kind of slot. Its value defaults by the function ARLO:TO-GENERATE-SLOT-PRINTER, which:

Gets the function for printing a slot's value.

Units with a To Default Value slot of #'TO-GENERATE-SLOT-READER

`To-Read-Value` is a slot which accepts values of type *Function Type* and makes sense for units of type *Slot Type*. This is a slot's function for reading in its value. Its value defaults by the function ARLO:TO-GENERATE-SLOT-READER, which:

Gets the function for reading in a slot's value.

Units with a To Default Value slot of #'TO-GENERATE-TO-VERIFY-TYPE

`To-Verify-Type` is a slot which accepts values of type *Function Type* and makes sense for units of type *Slot Type*. This is the function which verifies the suitability of a slot's attachment. Its value defaults by the function ARLO:TO-GENERATE-TO-VERIFY-TYPE, which:

Compute a slot's type checker with the Type-Checker coder.

Units not classifiable by `To-Default-Value`

`Actual-Get-Value` is a slot which accepts values of type *Function Type* and makes sense for units of type *Slot Type*. This is a slot's function for "physically" extracting its value.

Units with a Data Type slot of `Slot-Type`

`Shadow-Slot` is a slot which accepts values of type *Slot Type* and makes sense for units of type *Slot Type*. This is the prototype for all slots which shadow other slots.

Units with a Data Type slot of `Type-Type`

`Data-Type` is a slot which accepts values of type *Type Type* and makes sense for units of type *Slot Type*. This is a slot's description of its acceptable values– its range. Its value defaults by the function ARLO:DATA-TYPE-GENERATOR, which:

Looks through the prototypes of a slot for its data-type

`Makes-Sense-For` is a slot which accepts values of type *Type Type* and makes sense for units of type *Slot Type*. This describes the sorts of units a slot may attach to– its domain. Its value defaults by the function ARLO:MAKES-SENSE-FOR-GENERATOR, which:

Looks through the prototypes of a slot for its attachment type.

A-1.1.3 Units with a Makes Sense For slot of `Unit-Type`

These units are best organized by the **Prototype** relation.

Units with a prototype of Generic Slot

The unit **Typed Slot** is defined in the knowledge base `Core`. This is the prototype for slots which perform type checking.

Units with a prototype of Defaulting Slot

`Slot` is a slot which accepts values of type *Any Type* and makes sense for units of type *Unit Type*. This is the prototype for slots which both default and type check their values.

Units with a prototype of Slot

These units are best organized by the **Data Type** relation.

Units with a Data Type slot of `Any-Type`

`My-File-Of-Definition` is a slot which accepts values of type *Any Type* and makes sense for units of type *Unit Type*. This is the file in which a unit was defined. Its value defaults by the function ARLO:GET-TIME, which:

Gets the current universal time.

`My-Name` is a slot which accepts values of type *Any Type* and makes sense for units of type *Unit Type*. This is a unit's name. Its value defaults by the function ARLO:GENERATE-UNIT-NAME, which:

Generates a unit name. (Never really called?)

`Shadow-Slot-Slot` is a slot which accepts values of type *Any Type* and makes sense for units of type *Unit Type*. This stores the slot referring to ways to find a slot.

Units with a Data Type slot of `Function-Type`

`My-To-Describe-Self` is a slot which accepts values of type *Function Type* and makes sense for units of type *Unit Type*. This is a unit's function for describing itself. Its value defaults by the function ARLO:UNIT-DESCRIBER-GENERATOR, which:

Looks through the prootypes of a unit for a description function.

`My-To-Print-Self` is a slot which accepts values of type *Function Type* and makes sense for units of type *Unit Type*. This is a unit's function for printing itself. Its value defaults by the function ARLO:UNIT-PRINTER-GENERATOR, which:

Looks through the prootypes of a unit for a printer function.

Units with a Data Type slot of `List-Type`

`High-Level-Definition` is a slot which accepts values of type *List Type* and makes sense for units of type *Unit Type*. This is a definition for some function in a high level language. Its value defaults by the function ARLO:ASK-USER-FOR-SLOT, which:

Asks user for a slot on a window that's big enough.

Units with a Data Type slot of `String-Type`

`Description` is a slot which accepts values of type *String Type* and makes sense for units of type *Unit Type*. This is a string describing what this unit is. Its value defaults by the function ARLO:DEFAULT-DESCRIPTION-GENERATOR, which:

This generates a description excuse.

`My-Creator` is a slot which accepts values of type *String Type* and makes sense for units of type *Unit Type*. This is the user who created (actually, compiled) a unit. Its value defaults by the function ARLO:GET-HACKER, which:

Returns the full name of the current user, as a string.

EXPECTING is an ARLO coder. This defines a try and try again function which expects certain errors.. The functions it generates are specified by two parameters: `Errors-Expected` and `Possible-Methods`. It's body is generated by the function GENERATE-EXPECTING.

INHERIT-THROUGH is an ARLO coder. This defines functions which search for values along some relation.. The functions it generates are specified by one parameter: `Slot-To-Inherit-Through` . It's body is generated by the function GENERATE-INHERIT-THROUGH.

INHERITS? is an ARLO coder. This implements a function for confirming inheritance along some relation.. The functions it generates are specified by two parameters: `From-Unit` and `Slot-To-Search-Through`. It's body is generated by the function GENERATE-INHERITS?.

METHODS is an ARLO coder. This builds a try and try again function.. The functions it generates are specified by one parameter: `Possible-Methods` . It's body is generated by the function GENERATE-METHODS.

SLOT-COMPOSITION is an ARLO coder. This generates a slot composition function. The functions it generates are specified by one parameter: `Slots-To-Combine` . It's body is generated by the function GENERATE-SLOT-COMPOSITION.

TEST is an ARLO coder. This defines a complicated conjunction of many predicates.. The functions it generates are specified by one parameter: `Test-Criterion` . It's body is generated by the function GENERATE-TEST.

Units with a prototype of Hand Coded Function

`GENERATE-METHOD-DESCRIPTIONS` is a user defined lisp function which has an argument list of (UNIT IGNORE), and is documented as: *"Generates descriptions for each method in a try-and-try-again function."*.

A-1.3 Units defined in Arlo: SOURCES; CODING

These units are best organized by the **Makes Sense For** relation.

A-1.3.1 Units with a Makes Sense For slot of `Coded-Function-Type`

`Coded-By` is a slot which accepts values of type *Coder Type* and makes sense for units of type *Coded Function Type*. This is the unit describing the implementation of this function. Its value defaults by the function ARLO:DONT-DEFAULT-SLOT, which:

Signals an error if called to default a value.

`Internal-Name` is a slot which accepts values of type *Symbol Type* and makes sense for units of type *Coded Function Type*. This is the unit describing the implementation of this function. Its value defaults by the function ARLO:GENERATE-INTERNAL-FUNCTION-NAME, which:

This conses an ugly internal function name for a description.

A-1.3.2 Units with a Makes Sense For slot of `Coder-Type`

`Coder-Slot` is a slot which accepts values of type *Any Type* and makes sense for units of type *Coder Type*. This is the prototype for all parts of coder descriptions. Its value defaults by the function ARLO:ASK-

USER-FOR-SLOT, which:

Asks user for a slot on a window that's big enough.

`Description-Parameters` is a slot which accepts values of type *List Type* and makes sense for units of type *Coder Type*. These are the specifications from which the function is generated.

`Documentor` is a slot which accepts values of type *Function Type* and makes sense for units of type *Coder Type*. This is the function which documents this sort of function.

`Implementor` is a slot which accepts values of type *Function Type* and makes sense for units of type *Coder Type*. This is the function which codes up this sort of function.

`Name-Generator` is a slot which accepts values of type *Function Type* and makes sense for units of type *Coder Type*. This is the function which names this sort of function. Its value defaults by the function ARLO:TO-DEFAULT-NAME-GENERATOR, which:

This generates a function which generates function name generators.

A-1.3.3 Units with a Makes Sense For slot of @T[Function-Type]

`Function-Debugging-Info` is a slot which accepts values of type *List Type* and makes sense for units of type *Function Type*. This is random debugging information for a function. (Generated by the compiler) Its value defaults by the function ARLO:TO-DEFAULT-FUNCTION-DEBUGGING-INFO, which:

This finds the internal debugging information for a function.

`Function-Max-Args` is a slot which accepts values of type *Integer Type* and makes sense for units of type *Function Type*. This is the maximum number of arguments a function may take. Its value defaults by the function ARLO:TO-DEFAULT-MAX-ARGS, which:

This returns the maximnum number of args a function may take.

`Function-Min-Args` is a slot which accepts values of type *Integer Type* and makes sense for units of type *Function Type*. This is the minimum number of args a function requires. Its value defaults by the function ARLO:TO-DEFAULT-MIN-ARGS, which:

This returns the minimum number of args a function takes.

`Macros-Used` is a slot which accepts values of type *List Type* and makes sense for units of type *Function Type*. This is the macros used in defining a function. Its value defaults by the function ARLO:TO-DEFAULT-MACROS-USED, which:

This determines what macros were expanded for a given function.

`Magic-Argument-Descriptor` is a slot which accepts values of type *Integer Type* and makes sense for units of type *Function Type*. This is a magic number describing a functions arguments (generated by the compiler) Its value defaults by the function ARLO:TO-DEFAULT-MAGIC-ARGUMENT-DESCRIPTOR, which:

This returns a magical argument descriptor for a function.

A-1.3.4 Units with a Makes Sense For slot of `Implemented-Function-Type`

These units are best organized by the **Prototype** relation.

Units with a prototype of Slot

`Function-Descriptor` is a slot which accepts values of type *Any Type* and makes sense for units of type *Implemented Function Type*. This the prototype for attributes describing functions.

Units with a prototype of Function Descriptor

These units are best organized by the **Data Type** relation.

Units with a Data Type slot of `Lisp-Function-Type`

`Functional-Value` is a slot which accepts values of type *Lisp Function Type* and makes sense for units of type *Implemented Function Type*. This is a version of the function acceptable to APPLY. Its value defaults by the function ARLO:TO-DEFAULT-FUNCTIONAL-VALUE, which:

Gets the functional value - compiled or interpreted - of a function.

Units with a Data Type slot of `List-Type`

`Arglist` is a slot which accepts values of type *List Type* and makes sense for units of type *Implemented Function Type*. This is the argument list for a function. Its value defaults by the function ARLO:TO-DEFAULT-ARGLIST, which:

Defaults the arglist of a function.

`Lambda-Body` is a slot which accepts values of type *List Type* and makes sense for units of type *Implemented Function Type*. This is the body of the function. Its value defaults by the function ARLO:TO-DEFAULT-LAMBDA-BODY, which:

Finds or generates a lambda body for a function.

`Lambda-Definition` is a slot which accepts values of type *List Type* and makes sense for units of type *Implemented Function Type*. This is the lambda definition of a function. Its value defaults by the function ARLO:TO-DEFAULT-LAMBDA-DEFINITION, which:

This tries to compute a lambda definition for a slot.

Units with a Data Type slot of `String-Type`

`Documentation` is a slot which accepts values of type *String Type* and makes sense for units of type *Implemented Function Type*. This is the documentation for a function. Its value defaults by the function ARLO:TO-DEFAULT-DOCUMENTATION, which:

Finds the documentation for a function.

Units with a Data Type slot of `Subr-Type`

`Compiled-Definition` is a slot which accepts values of type *Subr Type* and makes sense for units of type *Implemented Function Type*. This is the compiled definition of a function. Its value defaults by the function ARLO:TO-DEFAULT-COMPILED-DEFINITION, which:

Compiles the definition of a function.

Units with a Data Type slot of `Valid-Function-Name-Type`

`Function-Name` is a slot which accepts values of type *Valid Function Name Type* and makes sense for units of type *Implemented Function Type*. This is the function spec for the function described by a unit. Its value defaults by the function ARLO:TO-DEFAULT-FUNCTION-NAME, which:

Computes a function name by looking on a coder slot.

A-1.3.5 Units not classifiable by `Makes-Sense-For`

The unit **Coder** is defined in the knowledge base `Core`. This is the prototype for all ARLO's automatic coders.

The unit **Hand Coded Function** is defined in the knowledge base `Core`. This is the prototype for functions defined by DEFINE.

The unit **Implemented Function** is defined in the knowledge base `Core`. This is the prototype for implemented LISP function descriptions.

A-1.4 Units defined in Arlo: SOURCES; LISP

These units all have PROTOTYPE slots of `Hand-Coded-Function`. `FIND-VALUE` is a user defined lisp function which has an argument list of (UNIT SLOT), and is documented as: *"Look through the prototoypes of a unit for a particular slot."*.

`GENERATE-INTERNAL-FUNCTION-NAME` is a user defined lisp function which has an argument list of (UNIT IGNORE), and is documented as: *"This conses an ugly internal function name for a description."*.

`MAKES-SENSE-FOR-GENERATOR` is a user defined lisp function which has an argument list of (UNIT SLOT), and is documented as: *"Looks through the prototypes of a slot for its attachment type."*.

`TO-DEFAULT-COMPILED-DEFINITION` is a user defined lisp function which has an argument list of (UNIT IGNORE), and is documented as: *"Compiles the definition of a function."*.

`TO-DEFAULT-DOCUMENTATION` is a user defined lisp function which has an argument list of (UNIT IGNORE), and is documented as: *"Finds the documentation for a function."*.

`TO-DEFAULT-FUNCTION-NAME` is a user defined lisp function which has an argument list of (UNIT IGNORE), and is documented as: *"Computes a function name by looking on a coder slot."*.

`TO-DEFAULT-FUNCTIONAL-VALUE` is a user defined lisp function which has an argument list of (UNIT IGNORE), and is documented as: *"Gets the functional value - compiled or interpreted - of a function."*.

`TO-DEFAULT-LAMBDA-BODY` is a user defined lisp function which has an argument list of (UNIT IGNORE), and is documented as: *"Finds or generates a lambda body for a function."*.

`TO-DEFAULT-LAMBDA-DEFINITION` is a user defined lisp function which has an argument list of (UNIT IGNORE), and is documented as: *"This tries to compute a lambda definition for a slot."*.

`TO-GENERATE-TO-VERIFY-TYPE` is a user defined lisp function which has an argument list of (SLOT IGNORE), and is documented as: *"Compute a slot's type checker with the Type-Checker coder."*.

`UNIT-PRINTER-GENERATOR` is a user defined lisp function which has an argument list of (UNIT SLOT), and is documented as: *"Looks through the prootypes of a unit for a printer function."*.

A-1.5 Units defined in Arlo: SOURCES; TYPES

These units are best organized by the **Prototype** relation.

A-1.5.1 Units without any prototype.

The unit **Type** is defined in the knowledge base `Core`. This is the prototype for all types. It accepts anything.

A-1.5.2 Units with a prototype of **Coder**

TYPE-CHECKER is an ARLO coder. Generates a type checking function for a slot.. The functions it generates are specified by one parameter: `Relevant-Slot` . It's body is generated by the function GENERATE-TYPE-CHECKER.

A-1.5.3 Units with a prototype of **Function Descriptor**

`Relevant-Slot` is a slot which accepts values of type *Any Type* and makes sense for units of type *Implemented Function Type*. A descriptor for the TYPE-CHECKER coder.

A-1.5.4 Units with a prototype of **Hand Coded Function**

`TO-GENERATE-SLOT-DESCRIBER` is a user defined lisp function which has an argument list of (UNIT IGNORE), and is documented as: *"Generates a function for describing a slot's value."*.

`TO-GENERATE-SLOT-PRINTER` is a user defined lisp function which has an argument list of (UNIT IGNORE), and is documented as: *"Gets the function for printing a slot's value."*.

`TO-GENERATE-SLOT-READER` is a user defined lisp function which has an argument list of (UNIT IGNORE), and is documented as: *"Gets the function for reading in a slot's value."*.

`TO-GENERATE-TYPE-CHECKER` is a user defined lisp function which has an argument list of (UNIT IGNORE), and is documented as: *"Generates the type checking function for a type."*.

A-1.5.5 Units with a prototype of **Slot**

These units are best organized by the **Data Type** relation.

Units with a Data Type slot of `Function-Type`

`Function-To-Describe` is a slot which accepts values of type *Function Type* and makes sense for units of type *Type Type*. This is the function for describing a value of a particular type. Its value defaults by the function INHERIT-THROUGH-GENERALIZATION, which:

Searches through the CORE:GENERALIZATION slots of a unit for a value.

`Function-To-Print` is a slot which accepts values of type *Function Type* and makes sense for units of type *Type Type*. This is the function for printing a value of a particular type. Its value defaults by the function INHERIT-THROUGH-GENERALIZATION, which:

Searches through the CORE:GENERALIZATION slots of a unit for a value.

`Function-To-Read` is a slot which accepts values of type *Function Type* and makes sense for units of type *Type Type*. This is the function for reading a value of a particular type. Its value defaults by the function INHERIT-THROUGH-GENERALIZATION, which:

Searches through the CORE:GENERALIZATION slots of a unit for a value.

`Specification` is a slot which accepts values of type *Function Type* and makes sense for units of type *Type Type*. This is the function which specializes this type. Its value defaults by the function ARLO:QUESTION-2, which:

Ask the user a question by:

(FORMAT QUERY-IO
" &What predicate specifies a from a?"
UNIT
(GET-VALUE UNIT `#>GENERALIZATION`*))*

`Type-Checking-Function` is a slot which accepts values of type *Function Type* and makes sense for units of type *Type Type*. This is the predicate for a type. Its value defaults by the function ARLO:TO-GENERATE-TYPE-CHECKER, which:

Generates the type checking function for a type.

Units with a Data Type slot of `Type-Type`

`Generalization` is a slot which accepts values of type *Type Type* and makes sense for units of type *Type Type*. This is the type upon which a given type is built. Its value defaults by the function ARLO:QUESTION-1, which:

Ask the user a question by:
(FORMAT QUERY-IO " &What is a a specialization of?" UNIT)

`My-Specific-Type` is a slot which accepts values of type *Type Type* and makes sense for units of type *Unit Type*. This is how to tell if a unit inherits from this unit.

A-1.5.6 Units with a prototype of **Type**

These units are best organized by the **Generalization** relation.

Types without any generalizations.

`Any-Type` specifies the class of LISP objects which satisfy the predicate ANYTHINGP (documented as "*A unparticular type predicate.*"). This is the top of the type hierarchy.

Types which are specializations of Any Type

`Function-Type` specifies a class of lisp objects which are classified by `Any-Type` and which additionally satisfy the predicate CALLABLEP (documented as "*Determines if an object is either a function or a function-describing unit*"). This is a type satisifed by any callable object (including function descriptions).

`Integer-Type` specifies a class of lisp objects which are classified by `Any-Type` and which additionally satisfy the predicate FIXP. This is a type requiring a LISP integer. (a fixnum or a bignum)

`List-Type` specifies a class of lisp objects which are classified by `Any-Type` and which additionally satisfy the predicate LIST-OR-NIL-P (documented as "*A predicate which accepts conses and NIL.*"). This is a type satisfied by any list (including NIL).

`Pathname-Type` specifies a class of lisp objects which are classified by `Any-Type` and which additionally satisfy the predicate PATHNAMEP. This is a type which is satisfied by any pathname

`String-Type` specifies a class of lisp objects which are classified by `Any-Type` and which additionally satisfy the predicate STRINGP. This is a type satisifed by any string.

`Symbol-Type` specifies a class of lisp objects which are classified by `Any-Type` and which additionally satisfy the predicate SYMBOLP. This is a type satisified by any LISP symbol.

`Unit-Type` specifies a class of lisp objects which are classified by `Any-Type` and which additionally satisfy the predicate UNITP (documented as "*Function determining if something is a unit- used by TYPEP*"). This is a type describing any ARLO unit.

Types which are specializations of Function Type

`Implemented-Function-Type` specifies a class of lisp objects which are classified by `Function-Type` and which additionally satisfy the predicate IMPLEMENTED-FUNCTION?. This is a type satisifed by any lisp function.

`Lisp-Function-Type` specifies a class of lisp objects which are classified by `Function-Type` and which additionally satisfy the predicate FUNCTIONP. This is a type satisifed by any lisp function.

`Subr-Type` specifies a class of lisp objects which are classified by `Function-Type` and which additionally satisfy the predicate SUBRP. This is a type satisfied by any LISP callable object (i.e. APPLIcable)

`Valid-Function-Name-Type` specifies a class of lisp objects which are classified by `Function-Type` and which additionally satisfy the predicate VALIDATE-FUNCTION-SPEC. This is a type satisifed by any lisp function spec.

Types which are specializations of Implemented Function Type

`Coded-Function-Type` specifies a class of lisp objects which are classified by `Implemented-Function-Type` and which additionally satisfy the predicate PROTOTYPE-OF-CODED-FUNCTION? (documented as "*Checks to see if a unit inherits from CORE:CODED-FUNCTION via CORE:PROTOTYPE.*"). This is a type satisifed by any lisp function.

Types which are specializations of Integer Type

`Time-Type` specifies a class of lisp objects which are classified by `Integer-Type` and which additionally satisfy the predicate FIXP. This is a type requiring an integer indicating seconds past the turn of the century.

Types which are specializations of Unit Type

`Coder-Type` specifies a class of lisp objects which are classified by `Unit-Type` and which additionally satisfy the predicate CODER?. This is a type describing any ARLO slot.

`Slot-Type` specifies a class of lisp objects which are classified by `Unit-Type` and which additionally satisfy the predicate SLOT? (documented as "*Determines if a unit is a slot- (i.e. has PRIMITIVE-SLOT as a prototype)*"). This is a type describing any ARLO slot.

`Type-Type` specifies a class of lisp objects which are classified by `Unit-Type` and which additionally satisfy the predicate IS-IT-A-TYPE-P (documented as "*Determines if something is a unit inheriting from TYPE.*"). This is a type which is satisified by any type describing ARLO unit.

A-1.6 Units defined in Arlo: SOURCES; WHISTLES

`(PROPERTY ARLO-UNIT NAMED-STRUCTURE-INVOKE)` is a user defined lisp function which has an argument list of (OP UNIT &REST MISC-ARGS), and is documented as: *"Data directed pretty printing and describing for units."*.

ASK-USER-FOR-SLOT is a user defined lisp function which has an argument list of (IN-UNIT A-SLOT &OPTIONAL (STREAM QUERY-IO) &REST FORMAT-ARGS), and is documented as: *"Asks user for a slot on a window that's big enough."*.

Chapter A-2
An Explanation 'Explanation'

These units are best organized by the **My File Of Definition** relation.

A-2.1 Units defined in Arlo: AI; DOCUMENT

These units are best organized by the **Prototype** relation.

A-2.1.1 Units with a prototype of **Explanation Slot**

`Positional-Assumptions` is a slot which accepts values of type *Any Type* and makes sense for units of type *Explanation Type*. These are The slots distinguished by this explanations superiors. Its value defaults by the function ARLO:TO-DEFAULT-POSITIONAL-ASSUMPTIONS, which:

Adds a units superiors primary division to its positional assumptions.

`Scribe-Documentor` is a slot which accepts values of type *Function Type* and makes sense for units of type *Explanation Type*. This is the function SCRIBE documentation for an explanation. Its value defaults by the function ARLO:FIND-VALUE, which:

Look through the prototoypes of a unit for a particular slot.

`Scribe-Explanation-Title` is a slot which accepts values of type *Any Type* and makes sense for units of type *Explanation Type*. This is the section title SCRIBE should use for this explanation. Its value defaults by the function ARLO:GENERATE-SCRIBE-EXPLANATION-TITLE, which:

Attempts to generate an appropriate scribe-style heading for a section.

A-2.1.2 Units with a prototype of **Hand Coded Function**

`DOCUMENT-FILE` is a user defined lisp function which has an argument list of (PATHNAME KB TITLE), and is documented as: *"Documents all the units in a given KB coming from a given file."*.

`DOIT-DEFAULT-SLOT` is a user defined lisp function which has an argument list of (UNIT SLOT), and is documented as: *"Signals an error if called to default a value."*.

EXPLANATION-PRECEDENCE is a user defined lisp function which has an argument list of (EXPLANATION1 EXPLANATION2), and is documented as: *"Establishes an order on a hierarchy of explanations."*.

GENERATE-SCRIBE-DOCUMENTATION-FOR-COMPLEX-EXPLANATION is a user defined lisp function which has an argument list of (EXPLANATION STREAM), and is documented as: *"Produces scribe documentation for an explanation of a set of units."*.

GENERATE-SCRIBE-DOCUMENTATION-FOR-UNIT-EXPLANATION is a user defined lisp function which has an argument list of (EXPLANATION STREAM), and is documented as: *"Documents a unit by looking for a scribe documentor on its prototypes."*.

GENERATE-SCRIBE-EXPLANATION-TITLE is a user defined lisp function which has an argument list of (EXPLANATION IGNORE), and is documented as: *"Attempts to generate an appropriate scribe-style heading for a section."*.

INHERITING? is a user defined lisp function which has an argument list of (SUPER UNIT BY-RELATION), and is documented as: *"Determines if some unit inherits another by some relation."*.

PRINT-UNIT-FOR-SCRIBE is a user defined lisp function which has an argument list of (UNIT STREAM), and is documented as: *"Prints a unit for SCRIBE, being cute about knowledge bases."*.

RUN-SCRIBE-DOCUMENTOR is a user defined lisp function which has an argument list of (ON-EXPLANATION TO-BUFFER), and is documented as: *"Runs the documentor on some explanation."*.

SAY-SLOT-VALUE is a user defined lisp function which has an argument list of (UNIT SLOT STREAM), and is documented as: *"Produces a psuedo-english description of some slot value."*.

SCRIBE-ALPHABETIZE-EXPLANATIONS is a user defined lisp function which has an argument list of (EXPLANATIONS), and is documented as: *"Sorts a set of explanations alphabetically by SCRIBE-EXPLANATION-TITLE"*.

SCRIBE-DOCUMENT-EXPLANATION is a user defined lisp function which has an argument list of (EXPLANATION TO-STREAM), and is documented as: *"Generates scribe documentation for an explanation."*.

SCRIBE-DOCUMENT-PERSON-EXPLANATION is a user defined lisp function which has an argument list of (EXPLANATION STREAM), and is documented as: *"Produces SCRIBE documentation for a person description."*.

SCRIBE-DOCUMENT-RANDOM-COMPLEX-EXPLANATION is a user defined lisp function which has an argument list of (EXPLANATION STREAM), and is documented as: *"Documents an indistinctive collection of units."*.

SCRIBE-DOCUMENTOR-FOR-CODED-FUNCTIONS is a user defined lisp function which has an argument list of (EXPLANATION STREAM), and is documented as: *"Produces SCRIBE documentation for an automatically coded function."*.

SCRIBE-DOCUMENTOR-FOR-CODERS is a user defined lisp function which has an argument list of (EXPLANATION STREAM), and is documented as: *"Produces SCRIBE documentation for an ARLO coder."*.

SCRIBE-DOCUMENTOR-FOR-RANDOM-UNIT-EXPLANATIONS is a user defined lisp function which has an argument list of (EXPLANATION STREAM), and is documented as: *"Generates a scribe explanation for a unit explanation."*.

SCRIBE-DOCUMENTOR-FOR-SLOT-EXPLANATIONS is a user defined lisp function which has an argument list of (EXPLANATION STREAM), and is documented as: *"Generates a scribe explanation for some slot."*.

SCRIBE-DOCUMENTOR-FOR-TYPE-EXPLANATIONS is a user defined lisp function which has an argument list of (EXPLANATION STREAM), and is documented as: *"Generates a scribe explanation for some slot."*.

SCRIBE-DOCUMENTOR-FOR-USER-FUNCTIONS is a user defined lisp function which has an argument list of (EXPLANATION STREAM), and is documented as: *"Produces SCRIBE documentation for an explanation of a user function."*.

SECTIONIZE-BY-HIERARCHICAL-SLOT is a user defined lisp function which has an argument list of (EXPLANATION STREAM), and is documented as: *"Documents a collection of units organized by a hierarchical relation."*.

SECTIONIZE-FILE-OF-DEFINITION-SLOT is a user defined lisp function which has an argument list of (EXPLANATION STREAM), and is documented as: *"Sets sectionization determined by file of definition."*.

SECTIONIZE-GENERALIZATION-SLOT is a user defined lisp function which has an argument list of (EXPLANATION STREAM), and is documented as: *"Sectionizes based on the GENERALIZATION slot."*.

SECTIONIZE-PROTOTYPE-SLOT is a user defined lisp function which has an argument list of (EXPLANATION STREAM), and is documented as: *"Sectionizes based on the PROTOTYPE slot."*.

SECTIONIZE-SUPERVISOR-SLOT is a user defined lisp function which has an argument list of (EXPLANATION STREAM), and is documented as: *"Sectionizes based on the INQUIR:SUPERVISOR slot."*.

TO-DEFAULT-MY-TO-SCRIBE-DOCUMENT-SELF is a user defined lisp function which has an argument list of (UNIT SLOT), and is documented as: *"Looks on ones prototypes for a function and otherwise returns a default."*.

TO-DEFAULT-POSITIONAL-ASSUMPTIONS is a user defined lisp function which has an argument list of (EXPLANATION IGNORE), and is documented as: *"Adds a units superiors primary division to its positional assumptions."*.

TO-DEFAULT-TO-SECTIONIZE-BY is a user defined lisp function which has an argument list of (UNIT SLOT), and is documented as: *"Looks on ones prototypes for a function and otherwise returns a default."*.

A-2.1.3 Units with a prototype of Slot

My-To-Scribe-Document-Self is a slot which accepts values of type *Function Type* and makes sense for units of type *Unit Type*. This is the function for writing SCRIBE documentation for a unit. Its value defaults by the function ARLO:TO-DEFAULT-MY-TO-SCRIBE-DOCUMENT-SELF, which:

Looks on ones prototypes for a function and otherwise returns a default.

To-Sectionize-By is a slot which accepts values of type *Function Type* and makes sense for units of type *Slot Type*. This is the function for sectionizing a description focussed on this slot. Its value defaults by the function ARLO:TO-DEFAULT-TO-SECTIONIZE-BY, which:

Looks on ones prototypes for a function and otherwise returns a default.

To-Speak-Value is a slot which accepts values of type *Function Type* and makes sense for units of type *Slot Type*. This describes how to say this slot in English (sort of).

A-2.2 Units defined in Arlo: AI; EXPLAIN

These units are best organized by the **Prototype** relation.

A-2.2.1 Units without any prototype.

The unit **Explanation** is defined in the knowledge base Explain. This is the prototypical explanation.

A-2.2.2 Units with a prototype of **Explanation Slot**

Explanation-Kb is a slot which accepts values of type *Any Type* and makes sense for units of type *Explanation Type*. This is the knowledge base in which this explanation is consed up. Its value defaults by the function ARLO:GET-ORIGINAL-KB, which:

Extracts the knowledge base a unit was originally in.

Explanation-Title is a slot which accepts values of type *String Type* and makes sense for units of type *Explanation Type*. This is a string describing this explanation.

Relevant-Slots is a slot which accepts values of type *Any Type* and makes sense for units of type *Explanation Type*. This is a list of the slots relevant to this explanation. Its value defaults by the function INHERIT-THROUGH-SUPER-EXPLANATION, which:

Searches through the CORE:EXPLAIN:SUPER-EXPLANATION slots of a unit for a value.

Super-Explanation is a slot which accepts values of type *Any Type* and makes sense for units of type *Explanation Type*. This is the explanation this explanation is a component of.

A-2.2.3 Units with a prototype of **Explanation**

The unit **Unit Explanation** is defined in the knowledge base Explain. This is the prototypical explanation of an individual unit.

The unit **Unit Set Explanation** is defined in the knowledge base Explain. This is the prototypical explanation of a set of units.

A-2.2.4 Units with a prototype of **Hand Coded Function**

COMPUTE-CHUNK-SIZE is a user defined lisp function which has an argument list of (PARTITION), and is documented as: *"Computes the average size of classified chunks in this partition."*.

CONSTRUCT-EXPLANATION is a user defined lisp function which has an argument list of (TITLE SYMBOLIC-DIVISION IN-EXPLANATION UNITS STRUCTURE), and is documented as: *"Constructs an explanation for a set of units."*.

EXTEND-PARTITION is a user defined lisp function which has an argument list of (PARTITION ELEMENT GROUP), and is documented as: *"This adds an element - and its associated group - to a partition."*.

EXTRACT-SIMPLEST-PARTITION is a user defined lisp function which has an argument list of (PARTITIONS), and is documented as: *"Selects the partition with the largest 'chunks' from a list of partitions."*.

GENERATE-EXCUSES is a user defined lisp function which has an argument list of (EXPLANATION), and is documented as: *"Generates an explanation for the 'misfits' of an explanation."*.

GENERATE-SET-PARTITIONS is a user defined lisp function which has an argument list of (FOR-EXPLANATION), and is documented as: *"Computes or reduces (from its super-explanation) the partitions for an explanation."*.

GENERATE-SUB-EXPLANATIONS is a user defined lisp function which has an argument list of (EXPLANATION), and is documented as: *"Generates sub explanations from the partition of an explanation."*.

GENERATE-UNIT-EXPLANATION is a user defined lisp function which has an argument list of (UNIT SUPER-EXPLANATION), and is documented as: *"This generates an explanation object for a particular unit."*.

GET-ORIGINAL-KB is a user defined lisp function which has an argument list of (UNIT IGNORE), and is documented as: *"Extracts the knowledge base a unit was originally in."*.

PARTITION-UNITS is a user defined lisp function which has an argument list of (UNITS BY-SLOT), and is documented as: *"This takes some units and returns the partition defined over them by some slot."*.

REDUCE-PARTITION is a user defined lisp function which has an argument list of (PARTITION OVER-UNITS), and is documented as: *"This takes the subset of a partition determined by some set of units."*.

REDUCE-PARTITION-SET is a user defined lisp function which has an argument list of (PARTITION-SET OVER-UNITS OVER-SLOTS), and is documented as: *"This takes a set of partitions and reduces each one."*.

TO-DEFAULT-SET-PARTITIONS is a user defined lisp function which has an argument list of (FOR-EXPLANATION IGNORE), and is documented as: *"Computes or reduces (from its super-explanation) the partitions for an explanation."*.

TO-DEFAULT-SUB-DIVISIONS is a user defined lisp function which has an argument list of (EXPLANATION IGNORE), and is documented as: *"Selects the partition with the largest 'chunks' from a list of partitions."*.

TO-DEFAULT-SUB-EXPLANATIONS is a user defined lisp function which has an argument list of (EXPLANATION IGNORE), and is documented as: *"Generates sub explanations from the partition of an explanation."*.

TO-DEFAULT-UNEXPLAINED-UNITS is a user defined lisp function which has an argument list of (EXPLANATION IGNORE), and is documented as: *"Generates an explanation for the 'misfits' of an explanation."*.

A-2.2.5 Units with a prototype of **Slot**

Explanation-Slot is a slot which accepts values of type *Any Type* and makes sense for units of type *Explanation Type*. This is the prototypical slot referring to explanations.

To-Partition-By is a slot which accepts values of type *Any Type* and makes sense for units of type *Slot Type*. This tells how to partition by a particular slot. Its value defaults by the function INHERIT-THROUGH-PROTOTYPE, which:

Searches through the CORE:PROTOTYPE slots of a unit for a value.

Unit-Explanation-Slot is a slot which accepts values of type *Any Type* and makes sense for units of type *Unit Explanation Type*. This is the prototypical slot referring to unit explanations.

Unit-Set-Explanation-Slot is a slot which accepts values of type *Any Type* and makes sense for units of type *Unit Set Explanation Type*. This is the prototypical slot referring to unit set explanations.

A-2.2.6 Units with a prototype of **Type**

Explanation-Type specifies a class of LISP objects which are classified by Unit-Type and which additionally satisfy the predicate PROTOTYPE-OF-EXPLANATION? (documented as *"Checks to see if a unit inherits from CORE:EXPLAIN:EXPLANATION via CORE:PROTOTYPE."*). This is a type satisifed by units inheriting (via the Prototype relation) from the unit Explanation.

Unit-Explanation-Type specifies a class of LISP objects which are classified by Unit-Type and which additionally satisfy the predicate PROTOTYPE-OF-UNIT-EXPLANATION? (documented as *"Checks to see if a unit in-*

herits from CORE:EXPLAIN:UNIT-EXPLANATION via CORE:PROTOTYPE."). This is a type satisifed by units inheriting (via the Prototype relation) from the unit Unit Explanation.

`Unit-Set-Explanation-Type` specifies a class of LISP objects which are classified by `Unit-Type` and which additionally satisfy the predicate `PROTOTYPE-OF-UNIT-SET-EXPLANATION?` (documented as *"Checks to see if a unit inherits from CORE:EXPLAIN:UNIT-SET-EXPLANATION via CORE:PROTOTYPE."*). This is a type satisifed by units inheriting (via the Prototype relation) from the unit Unit Set Explanation.

A-2.2.7 Units with a prototype of **Unit Explanation Slot**

`Unit-To-Explain` is a slot which accepts values of type *Any Type* and makes sense for units of type *Unit Explanation Type*. This is the individual unit this explanation is about.

A-2.2.8 Units with a prototype of **Unit Set Explanation Slot**

These units all have MAKES-SENSE-FOR slots of `Unit-Set-Explanation-Type`.

These units are best organized by the **Data Type** relation.

Units with a Data Type slot of `Any-Type`

`Sub-Explanations` is a slot which accepts values of type *Any Type* and makes sense for units of type *Unit Set Explanation Type*. These are the explanations which are component to this explanation. Its value defaults by the function ARLO:TO-DEFAULT-SUB-EXPLANATIONS, which:

Generates sub explanations from the partition of an explanation.

`Symbolic-Division` is a slot which accepts values of type *Any Type* and makes sense for units of type *Unit Set Explanation Type*. This is a symbolic description of the focus of this explanation.

`Unexplained-Units` is a slot which accepts values of type *Any Type* and makes sense for units of type *Unit Set Explanation Type*. This is an explanation of the units not covered in this explanation. Its value defaults by the function ARLO:TO-DEFAULT-UNEXPLAINED-UNITS, which:

Generates an explanation for the 'misfits' of an explanation.

Units with a Data Type slot of `Integer-Type`

`Section-Size` is a slot which accepts values of type *Integer Type* and makes sense for units of type *Unit Set Explanation Type*. This is the threshold when explanation sectionization is attempted.

Units with a Data Type slot of `List-Type`

`Relevant-Units` is a slot which accepts values of type *List Type* and makes sense for units of type *Unit Set Explanation Type*. This is a list of the units this explanation refers to.

`Set-Partitions` is a slot which accepts values of type *List Type* and makes sense for units of type *Unit Set Explanation Type*. This is a list of the possible partitions of this set of units. Its value defaults by the function ARLO:TO-DEFAULT-SET-PARTITIONS, which:

Computes or reduces (from its super-explanation) the partitions for an explanation.

`Structural-Slots` is a slot which accepts values of type *List Type* and makes sense for units of type *Unit Set Explanation Type*. This is a list of the slots which may sectionize this explanation.

Sub-Divisions is a slot which accepts values of type *List Type* and makes sense for units of type *Unit Set Explanation Type*. This is the most relevant partition of the set of units. Its value defaults by the function ARLO:TO-DEFAULT-SUB-DIVISIONS, which:

Selects the partition with the largest 'chunks' from a list of partitions.

A-2.3 Units defined in Arlo: SOURCES; LISP

DECACHE-FINDER is a user defined lisp function which has an argument list of (UNIT SLOT), and is documented as: *"This finds the deaching function for a unit by looking through its prototypes."*.

Chapter A-3
References

[Bac78] John Backus. Can programming be liberated from the von neumann style? a functional style and its algebra of programs. *Communications of the ACM*, 21(8), August 1978.

[BS83] Daniel Bobrow and Mark Stefik. *The Loops Manual.* Xerox Corporation, 1983.

[Can83] Howard Cannon. *Programming with Flavors.* Symbolics, Inc., Cambridge, Massacussets, 1983.

[Cha83] David Chapman. *Naive Mathematics and Naive Problem Solving.* Working Paper 249, Artifical Intelligence Laboratory, MIT, June 1983.

[Doy77] Jon Doyle. *A Truth Maintenance System.* Master's thesis, Massachussets Institute of Technology, 1977. Available as MIT AI Technical Report.

[GR84] Adele Goldberg and David Robson. *Smalltalk-80: The Language an its Implementation.* Addison-Wesley, Reading, Massachusetts, 1984.

[Gre80] Russel Greiner. *RLL-1: A Representation Langauge Language.* Working Paper 80-9, Stanford Heuristic Programming Project, October 1980.

[Haa86a] Ken Haase. *ARLO': Describing Representations.* Memo 955, Artificial Intelligence Laboratory, MIT, 1986.

[Haa86b] Ken Haase. Discovery systems. In *ECAI '86 Proceedings*, ECAI, August 1986. Also available as MIT AI Memo 899.

[Haa86c] Ken Haase. *Why Representation Language Languages are No Good.* AI Memo 943, Artificial Intelligence Laboratory, MIT, October 1986.

[Hil85] Danny Hillis. *The Connection Machine.* MIT Press, 1985.

[Hof80] Douglas Hofstader. *Godel, Escher, Bach: An Eternal Golden Braid.* Basic Books, 1980.

[Len82] Douglas B. Lenat. Am: discovery in mathematics as heuristic search. In *Knowledge-Based Systems in Artificial Intelligence*, McGraw Hill, 1982.

[Len83] Doug Lenat. Eurisko: a program that learns new heuristics and domain concepts. *The AI Journal*, March 1983. This is the last in a series of articles on heuristics about heuristics.

[LSP85] Doug Lenat, Mary Shepherd, and Mayank Prakash. Cyc: a large common-sense knowledge base. *AI Magazine*, June 1985.

[McA78] David McAllester. *A three-valued truth maintenance system.* AI Memo 473, Artificial Intelligence Laboratory, MIT, 1978.

[McD83] David McDonald. Mumble: a natural language generation system. In *Computational Theories of Discourse*, MIT Press, 1983.

[Mil78] Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Science 17*, 1978.

[MP84] David McDonald and Robert Putojevsky. Generating text for an intelligent encyclopedia. In *AAAI-84*, AAAI, August 1984.

[RG77] Bruce Roberts and Ira Goldstein. *FRL Users Manual.* AI Memo 408, Artificial Intelligence Laboratory, MIT, 1977.

[Ric80] Charles Rich. *Inspection Methods In Programming.* PhD thesis, Massachussets Institute of Technology, 1980. Also available as MIT AI Lab Technical Report 604.

[SR76] Howard Shrobe and Charles Rich. *Initial Report on a LISP Programmers Apprentice.* Master's thesis, Massachussets Institute of Technology, 1976. Also available as MIT AI Lab Technical Report 354.

[Ste79] Mark Stefik. An examination of a frame-structured representation system. In *Proceedings of the Sixth International Joint Conference on Artificial Intelligence*, IJCAI, August 1979.

[Ste84] Guy L. et al Steele. *Common Lisp Reference Manual.* Digital Equipment Corporation, Maynard, Massacussets, 1984.

[Wat78] Richard Waters. *Automatic Analysis of the Logical Structure Of Programs.* PhD thesis, Massachussets Institute of Technology, 1978. Also available as MIT AI Lab Technical Report 492.

[Wei83] Daniel Weinreb. *Signalling and Handling Conditions.* Symbolics, Inc., Cambridge, Massacussets, 1983.

[WM82] Daniel Weinreb and David Moon. *Lisp Machine Manual.* Symbolics, Inc., Cambridge, Massacussets, 1982.

This blank page was inserted to preserve pagination.

CS-TR Scanning Project
Document Control Form

Date : 10 / 26 / 95

Report # AI-TR-901

Each of the following should be identified by a checkmark:
Originating Department:

- [x] Artificial Intellegence Laboratory (AI)
- [ ] Laboratory for Computer Science (LCS)

Document Type:

- [x] Technical Report (TR)
- [ ] Technical Memo (TM)
- [ ] Other: _____

Document Information

Number of pages: 99 (106-IMAGES)
Not to include DOD forms, printer intstructions, etc... original pages only.

Originals are:

- [x] Single-sided or
- [ ] Double-sided

Intended to be printed as :

- [ ] Single-sided or
- [x] Double-sided

Print type:

- [ ] Typewriter
- [ ] Offset Press
- [ ] Laser Print
- [ ] InkJet Printer
- [ ] Unknown
- [x] Other: COPY ?

Check each if included with document:

- [x] DOD Form (2)
- [ ] Funding Agent Form
- [x] Cover Page
- [x] Spine (ON COVER)
- [ ] Printers Notes
- [ ] Photo negatives
- [ ] Other: _____

Page Data:

Blank Pages (by page number): _____

Photographs/Tonal Material (by page number): _____

Other (note description/page number):

Description : Page Number:

IMAGE MAP: (1 - 12) UN#'ED TITLE, UN# BLANK, UN# ACK, UN# BLK, ii, UN# BLK, iii-vii, UN# BLK,
(13 - 99) PAGES #'ED 1 - 87
(100 - 106) SCANCONTROL, COVER (WITH SPINE), DOD(2), TRGT'S(3)

Scanning Agent Signoff:

Date Received: 10 / 26 / 95 Date Scanned: 11 / 13 / 95 Date Returned: 11 / 16 / 95

Scanning Agent Signature: Michael N. Cook

Rev 9/94 DS/LCS Document Control Form cstrform.vsd

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER AI-TR-901	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) ARLO Another Representation Language Offer		5. TYPE OF REPORT & PERIOD COVERED Technical Report 6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) Kenneth W. Haase Jr.		8. CONTRACT OR GRANT NUMBER(s) N00014-85-K-0124
9. PERFORMING ORGANIZATION NAME AND ADDRESS Artificial Inteligence Laboratory 545 Technology Square Cambridge, MA 02139		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS Advanced Research Projects Agency 1400 Wilson Blvd. Arlington, VA 22209		12. REPORT DATE October, 1986 13. NUMBER OF PAGES 95
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office) Office of Naval Research Information Systems Arlington, VA 22217		15. SECURITY CLASS. (of this report) Unclassified 15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Distribution is unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

None

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Knowledge Representation, Representation Languages, Meta-Representation, Reflection, Artificial Intelligence, AI Languages, RLL, Representation Language Languages

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

This paper describes ARLO, a representation language language loosely modelled after Greiner and Lenat's RLL-1. ARLO is a structure-based representation language for describing structure-based representation languages, including itself. A given representation language is specified in ARLO by a collection of structures describing how its descriptions are interpreted, defaulted, and verified. This high level description is compiled into LISP code and ARLO structures whose interpretation fullfills

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE
S/N 0102-014-6601

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

the specified semantics of the language. In addition, ARLO itself --- as a language for expressing and compiling partial and complete language specifications --- is described and interpted in the same manner as the languages it describes and implements. This self description can be extended or modified to expand or alter the expressive power of ARLO's initial configuration. Languages which describe themselves --- like ARLO --- provide powerful mediums for systems which perform automatic self-modification, optimization, debugging, or documentation. AI systems implemented in such a self-descriptive language can reflect on their own capabilities, applying general problem solving and learning strategies to enlarge or correct them.

Scanning Agent Identification Target

Scanning of this document was supported in part by the **Corporation for National Research Initiatives,** using funds from the **Advanced Research Projects Agency** of the **United states Government** under Grant: **MDA972-92-J1029.**

The scanning agent for this project was the **Document Services** department of the **M.I.T Libraries.** Technical support for this project was also provided by the **M.I.T. Laboratory for Computer Sciences.**

Scanned

Date: 11/13/1995

M.I.T. Libraries
Document Services