A Representation Language Language

by

Russell Greiner and Douglas B. Lenat
Computer Science Deptartment
Stanford University

ABSTRACT

The field of AI is strewn with knowledge representation languages. The language designer typically has one particular application domain in mind; as subsequent types of applications are tried, what had originally been useful *features* become undesirable limitations, and the language is overhauled or scrapped. One remedy to this bleak cycle might be to construct a representational scheme whose domain is the field of representational languages itself. Toward this end, we designed and implemented RLL, a frame-based Representation Languange Language. The components of representation languages in general (such as slots and inheritance mechanisms) and of RLL itself are encoded declaratively as frames. Modifying these frames can change the semantics of RLL, by radically altering the character of the RLL environment.

1. MOTIVATION

"One ring to rule them all... and in the darkness bind them."

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

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

One remedy to this bleak cycle might be to construct a representation scheme whose domain is the field of representational languages itself, a program which could then be tailored to suit many specific applications. Toward this end, we are designing and implementing RLL, an object-centered[1] Representation Languange Language.[2] This paper reports on the current state of our ideas and our implementation.

[1] This "*object-centering*" does not represent a loss in generality. We will soon see that each part of the full system, including procedural information, is reified as a unit.

[2] As RLL is itself a completely self-descriptive representation language, there is no need for an RLLL.

2. INTRODUCTION

RLL explicitly represents (i.e. contains units[3] for) the components of representation languages in general and of itself in particular. The programming language LISP derives its flexibility in a similar manner: it, too, encodes many of its constructs within its own formalisms. Representation languages aim at easy, natural interfacing to users; therefore their primitive building blocks are larger, more abstract, and more complex than the primitives of programming languages.

Building blocks of a representation language include such things as control regimes (ExhaustiveBackwardChaining, Agendas), methods of associating procedures with relevant knowledge (Footnotes, Demons), fundamental access functions (Put/Get, Assert/Match), automatic inference mechanisms (InheritFromEvery2ndGeneration, InheritBut-PermitExceptions), and even specifications of the intended semantics and epistemology of the components (ConsistencyConstraint, EmpiricalHeuristic).

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

3. HOW IS A REPRESENTATION LANGUAGE LIKE AN ORGAN?

When the user starts RLL, he finds himself in an environment very much like the Units package [*Stefik*], with one major difference. If he desires a new type of inheritance mechanism, he need only create a new Inheritance-type of unit, initialize it with that desired property; and that new mode of inheritance will automatically be enabled. This he can do using the same editor and accessing functions he uses for entering and codifying his domain knowledge (say, poultry inspection); only here the information pertains to the actual Knowledge Base system itself, not turkeys.

The Units package has Get and Put as its fundamental storage and retrieval functions; therefore RLL also begins in that state. But there is nothing sacred about even these two "primitives". Get and Put are encoded as modifiable units; if they are altered, the nature of accessing a slot's value will change correspondingly. In short, by issuing a small number of commands he can radically alter the character of the RLL environment, molding it to his personal

[3] RLL is a frame-based system [*Minsky*], whose building blocks are called *Units* [*Stefik*], [*Bobrow & Winograd*]. Each unit consists of a set of *Slots* with their respective values.

preferences and to the specific needs of his application. RLL is responsible for performing the necessary "truth maintainence" operations, (e.g. retroactive updates) to preserve the overall correctness of the system as a whole. For example, Get and Put can be transformed into units which, when run as functions, resemble Assert (store proposition) and Match (retrieve propositions), and the user need never again mention "slots" at all.

RLL is more like a stop organ than a piano. Each stop corresponds to a "pre-fabricated" representational part (e.g. a slot, inheritance, format, control regime, etc.), which resides in the overall RLL system. The initial RLL is simply one configuration of this organ, with certain stops "pulled out" to mimic the Units package. These particular stops reflect our intuitions of what constitutes a general, powerful system. Some of the units initially "pulled out" (activated) define more or less standard inheritance regimes, such as Inherit-Along-*IS-A*-Links, which enables Fido to gather default information from AnyDog.

We chose to include a large assortment of common slots. One hundred and six types of slots, including *IS-A*, *SuperClass*, *BroaderHeuristics*, and *TypicalExamples*, are used to hierarchically organize the units. That number grows daily, as we refine the organizing relationships which were originally "smeared" together into just one or two kinds of slots (e.g. *A-Kind-Of*). An additional fifteen types of slots, including *ToGetValue*, *ToPutValue*, *ToKillUnit*, and *ToAddValue*, collectively define the accessing/updating functions.

This bootstrapping system (the initial configuration of "organ stops",) does *not* span the scope of RLL's capabilities: many of its stops are initially in the dormant position. Just as a competent musician can produce a radically different sound by manipulating an organ's stops, so a sophisticated RLL user can define his own representation by turning off some features and activating others. For instance, an FRL devotee may notice -- and choose to use exclusively -- the kind of slot called *A-Kind-Of*, which is the smearing together of *Is-A*, *SuperSet*, *Abstraction*, *TypicalExampleOf*, *PartOf*, etc. He may then deactivate those more specialized units from his system permanently. A user who does not want to see his system as a hierarchy at all can simply deactivate the A-Kind-Of unit and its progeny. The user need not worry about the various immediate, and indirect, consequences of this alteration (e.g., deleting the Inherit-Along-*IS-A*-Links unit); RLL will take care of them. By selectively pushing and pulling, he should be able to construct a system resembling almost any currently used representational language, such as KRL, OWL and KLONE;[4] after all, an organ *can* be made to sound like a piano.

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

---

[4] This particular task, of actually simulating various existing Representation Languages, has not yet been done. It is high on our agenda of things to do. We anticipate it will require the addition of many new components (and types of components) to RLL, many representing orthogonal decompositions of the space of knowledge representation.

4. EXAMPLES

The following examples convey the flavor of what can currently be done with the default settings of the RLL "organ stops".

4.1. EXAMPLE: Creating a New Slot

In the following example, the user wishes to define a *Father* slot, in a sexist geneological knowledge base which containes only the primitive slots *Mother* and *Spouse*. As RLL devotes a unit to store the necessary knowledge associated with each kind of slot, (see Figure 1,) defining a new kind of slot means creating and initializing one new unit. In our experience, the new desired slot is frequently quite similar to some other slot(s), with but a few distinguishing differences. We exploited this regularity in developing a high level "slot-defining" language, by which a new slot can be defined precisely and succinctly in a single declarative statement.

Name:	IS-A
Description:	Lists the classes I AM-A member of.
Format:	List-of-Entries
Datatype:	Each entry represents a class of objects.
Inverse:	Examples
IS-A:	(AnySlot)
UsedByInheritance:	Inherit-Along-*IS-A*-Links
MyTimeOfCreation:	1 April 1979, 12:01 AM
MyCreator:	D.B.Lenat

Figure # 1 - Unit devoted to the "*IS-A*" slot. There are many other slots which are appropriate for this unit; whose value will be deduced automatically (e.g. inherited from AnySlot) if requested.

Creating a new slot for *Father* is easy: we create a new unit called Father, and fill its *HighLevelDefn* slot with the value (Composition *Spouse Mother*). Composition is the name of a unit in our initial system, a so-called "slot-combiner" which knows how to compose two slots (regarding each slot as a function from one unit to another). We also fill the new unit's *Isa* slot, deriving the unit shown in Figure 2.

Name:	Father
IS-A:	(AnySlot)
HighLevelDefn:	(Composition *Spouse Mother*)

Figure # 2 - Slots filled in by hand when creating the unit devoted to the "*Father*" slot. Several other slots (e.g., the syntactic slots *MyCreator*, *MyTimeOfCreation*) are filled in automatically at this time.

The user now asks for KarlPhilippEmanuel's father, by typing

```
(GetValue 'KPE 'Father).
```

GetValue first tries a simple associative lookup (GET), but finds there is no *Father* property stored on KPE, the unit representing KarlPhilippEmanuel. GetValue then tries a more sophisticated approach: ask the Father unit how to compute the *Father* of any person. Thus the call becomes

```
[Apply* (GetValue 'Father 'ToCompute) 'KPE].
```

Notice this calls on GetValue recursively, and once again there is no value stored on the *ToCompute* slot of the unit called Father. The call now has expanded into

```
[Apply* (Apply* (GetValue 'ToCompute 'ToCompute)
                'Father) 'KPE].
```

Luckily, there *is* a value on the *ToCompute* slot of the unit ToCompute: it says to find the *HighLevelDefn*, find the slot-combiner it employs, find its *ToCompute*[5], and ask it. Our call is now expanded out into the following:

```
[Apply* (Apply* (GetValue 'Composition
 'ToCompute) 'Spouse 'Mother) 'KPE].
```

---

[5] Each unit which represents a function has a *ToCompute* slot, which holds the actual LISP function it encodes. Associating such a *ToCompute* slot with each slot reflects our view that each slot is a function, whose argument happens to be a unit, and whose computed value may be cached away.

The unit called Composition does indeed have a *ToCompute* slot; after applying it, we have:

```
[Apply* '(λ (x) (GetValue (GetValue x 'Mother)
                   'Spouse)) 'KPE].
```

This asks for the *Mother* slot of KPE, which is always physically stored in our knowledge base, and then asks for the value stored in her *Spouse* slot. The final result, JohannSebastian, is then returned. It is also cached (stored redundantly for future use) on the *Father* slot of the unit KPE. See [Lenat *et al.*, 1979] for details.

Several other slots (besides *ToCompute*) are deduced automatically by RLL from the *HighLevelDefn* of Father (see Figure 3) as they are called for. The *Format* of each *Father* slot must be a single entry, which is the name of a unit which represents a person. The only units which may have a *Father* slot are those which may legally have a *Mother* slot, *viz.*, any person. Also, since *Father* is defined in terms of both *Mother* and *Spouse*, using the slot combiner Composition, a value stored on KPE:*Father* must be erased if ever we change the value for KPE's *Mother* or AnnaMagdelena's *Spouse*, or the definition (that is, *ToCompute*) of Composition.

Name:	Father
IS-A:	(AnySlot)
HighLevelDefn:	(Composition *Spouse Mother*)
Description:	Value of unit's *Mother*'s *Spouse*.
Format:	SingleEntry
Datatype:	Each entry is unit representing a person.
MakesSenseFor:	AnyPerson
DefinedInTermsOf:	(Spouse Mother)
DefinedUsing:	Composition
ToCompute:	(λ (x) (GetValue (GetValue x *'Mother*) *'Spouse*)

Figure #3 - Later form of the Father unit, showing several slots filled in automatically.

Notice the the ease with which a user can currently "extend his representation", enlarging his vocabulary of new slots. A similar, though more extravagant example would be to define *FavoriteAunt* as (SingleMost (Unioning (Composition *Sister Parents)* (Composition *Spouse Brother Parents*)) *$$*). Note that "Unioning" and "SingleMost" are two of the slot combiners which come with RLL, whose definition and range can be inferred from this example.

It is usually no harder to create a new type of slot format (OrderedNonemptySet), slot combiner (TwoMost, Starring), or datatype (MustBePersonOver16), than it was to create a new slot type or inheritance mechanism. Explicitly encoding such information helps the user (and us) understand the precise function of each of the various components. We do not yet feel that we have a complete set of any of these components, but are encouraged by empirical results like the following: The first two hundred slots we defined required us to define thirteen slot combiners, yet the *last* two hundred slots required only five new slot combiners.

4.2. EXAMPLE: Creating a New Inheritance Mode

Suppose a geneticist wishes to have a type of inheritance which skips every second ancestor. He browses through the hierarchy of units descending from the general one called Inheritance, finds the closest existing unit, InheritSelectively, which he copies into a new unit, InheritFromEvery2ndGeneration. Editing this copy, he finds a high level description of the path taken during the inheritance, wherein he replaces the single occurrence of "*Parent*" by "*GrandParent*" (or by two occurrences of *Parent*, or by the phrase (Composition *Parent Parent*)). After exiting from the edit, the new type of inheritance will be active; RLL will have translated the slight change in the unit's high-level description into a multitude of low-level changes. If the geneticist now specifies that Organism#34 is an "InheritFromEvery2ndGeneration offspring" of Organism#20, this will mean the right thing. Note that the tools used (browser, editor, translator, etc.) are themselves encoded as units in RLL.

4.3. EXAMPLE: Epistemological Status

Epistemological Status: To represent the fact that John believes that Mary is 37 years old, RLL adds the ordered pair (SeeUnit AgeOfMary0001) to the the *Age* slot of the Mary unit. RLL creates a unit called AgeOfMary0001, fills its **value** slot with 37 and its *EpiStatus* slot with "John believes". See Figure 4. Note this mechanism suffices to represent belief about belief (just a second chained SeeUnit pointer), quoted belief ("John thinks he knows Mary's age", by omitting the **value** 37 slot in AgeOfMary0001), situational fluents, etc. This mechanism can also be used to represent arbitrary n-ary relations, escaping the associative triple (i.e. Unit/*Slot*/value) limitation.

Name:	Mary
IS-A:	(Person Female ContraryActor)
Description:	The grower of silver bells etc.
Age:	(39 (SeeUnit) (SeeUnit))
Mother:	Jane
Father:	((SeeUnit) ...)
:	:

Name	AgeOfMary0001
Isa	(UnitForASlotFiller)
LivesInUnit	Mary
LivesInSlot	Age
value	37
Epistatus	John believes
Teleology:	Epistemic

Name	AgeOfMary0002
Isa	(UnitForASlotFiller)
LivesInUnit	Mary
LivesInSlot	Age
value	21
Epistatus	During Wedding805
Teleology:	Historic

Figure 4 - Representing "John believes that Mary is 37, but she's really 39. When she was married, she was 21".

4.4. EXAMPLE: Enforcing Semantics

Suppose that Lee, a user of RLL, is constructing HearSayXXIV, a representation language which contains cooperating knowledge sources (KSs). He specifies that each unit representing a knowledge source should have some very precise applicability criteria (he defines a *FullRelevancy* slot) and also a much quicker, rougher check of its potential relevance (he defines a *PrePreConditions* slot). If HearSayXXIV users employ these two slots in just the way he intended, they will be rewarded with a very efficiently-running program.

But how can Lee be *sure* that users of HearSayXXIV will use these two new slots the way he intended? He also defines a new kind of slot called *Semantics*. The unit for each type of slot can have a *Semantics* slot, in which case it should contain criteria that the values stored in such slots are expected to satisfy.

Lee fills the *Semantics* slot of the unit called PrePreConditions with a piece of code that checks that the *PrePreConditions* slot of every KS unit is filled by a Lisp predicate, which is very quick to execute, which (empirically) correlates highly to the *FullRelevancy* predicate, and which rarely returns NIL when the latter would return T. This bundle of constraints captures what he "really means" by *PrePreConditions*.

A user of HearSayXXIV, say Bob, now builds and runs a speech understanding program containing a large collection of cooperating knowledge sources. As he does so, statistics are gathered empirically. Suppose Bob frequently circumvents the *PrePreConditions* slot entirely, by placing a pointer there to the same long, slow, complete criteria he has written for the *FullRelevancy* slot of that KS. This is empirically caught as a violation of one of the constraints which Lee recorded in the *Semantics* slot of the unit PrePreConditions. As a result, the *Semantics* slot of the Semantics unit will be consulted to find an appropriate reaction; the code therein might direct it to print a warning message to Bob: "The *PrePreConditions* slot of a KS is meant to run very quickly, compared with the *FullRelevancy* slot, but 70% of yours don't; please change your *PrePreConditions* slots, or your *FullRelevancy* slots, or (if you insist) the *Semantics* slot of the PrePreConditions unit".[6]

5. SPECIFICATIONS FOR ANY REPRESENTATION LANGUAGE LANGUAGE

The following are some of the core constraints around which RLL was designed. One can issue commands to RLL which effectively "turn off" some of these features, but in that case the user is left with an inflexible system we would no longer call a representation language language. Further details may be found in [*Lenat, Hayes-Roth, & Klahr*] and in [*Genesereth & Lenat*].

Self-description: No part of the RLL system is opaque; even the primitive Get and Put and Evaluate functions are represented by individual units describing their operation.[2] Current status: complete (to a base language level).

Self-modification: Changes in the high-level description of an RLL process automatically result in changes in the Lisp code for -- and hence behavior of -- RLL. Current status: this works for changes in definition, format, etc. of units representing slots and control processes. Much additional effort is required.

Codification of Representation Knowledge: Taxonomies of inheritance, function invocation, etc. Tools for manipulating and creating same. These correspond to the stops of the organ, illustrated above. Current status: this is some of the most exciting research we foresee; only a smattering of representation knowledge has yet been captured.

6. INITIAL "ORGAN STOPS"

The following characteristics pertain especially to the initial state of the current RLL system, wherein all "organ stops" are set at their default positions. Each RLL user will doubtless settle upon some different settings, more suited to the representation environment he wishes to be in while constructing his application program. For details, see [*Greiner*].

Cognitive economy: Decision-making about what intermediate values to cache away, when to recompute values, expectation-filtering. Current status: simple reasoning is done to determine each of these decisions; the hooks for more complex procedures exist, but they have not been used yet.

Syntactic *vs* Semantic slots: Clyde should inherit values for many slots from TypicalElephant, such as *Color, Diet, Size*; but *not* from slots which refer to TypicalElephant *qua* data structure, slots such as *NumerOfFilledInSlots* and *DateCreated*. Current status: RLL correctly treats these two classes of slots differently, e.g. when initializing a new unit.

Onion field of languages: RLL contains a collection of features (e.g., automatically adding inverse links) which can be individually enabled or disabled, rather than a strict linear sequence of higher and higher level languages. Thus it is more like an onion *field* than the standard "skins of an onion" layering. Current status: Done. Three of the most commonly used settings are bundled together as CORLL, ERLL, and BRLL.

----

[6] This work has led us to realize the impossibility of unambiguously stating semantics. Consider the case of the semantics of the Lisp function "OR". Suppose one person believes it evaluates its arguments left to right until a non-null value is found; a second person believes it evaluates right to left; a third person believes it evaluates all simultaneously. They go to the *Semantics* slot of the unit called OR, to settle the question. Therein they find this expression: (OR *(Evaluate the args left to right) (Evaluate the args right to left)*). Person #3 is convinced now that he was wrong, but persons 1 and 2 point to each other and exclaim in unison "See, I *told* you!" The point of all this is that even storing a Lisp predicate in the *Semantics* slots only specifies the meaning of a slot up to a set of fixed points. One approaches the description of the semantics with some preconceived ideas, and there may be more than one set of such hypotheses which are consistent with everything stored therein. See [*Genesereth & Lenat*].

Economy via Appropriate Placement: Each fact, heuristic, comment, etc. is placed on the unit (or set of units) which are as general and abstract as possible. Frequently, new units are created just to facilitiate such appropriate placement. In the long run, this reduces the need for duplication of information. One example of this is the use of of appropriate conceptual units:

Clarity of Conceptual Units: RLL can distinguish (i.e. devote a separate unit to each of) the following concepts: TheSetOfAllElephants, (whose associated properties describe this as a set -- such as *#OfMembers* or *SubCategories*), TypicalElephant, (on which we might store *Expected-TuskLength* or *DefaultColor* slots), ElephantSpecies, (which *EvolvedAsASpecies* some 60 million years ago and is *CloselyRelatedTo* the HippopatamusSpecies,) ElephantConcept, (which *QualifiesAsA* BeastOfBurden and a TuskedPackyderm,) ArchetypicalElephant (which represents an elephant, in the real world which best exemplifies the notion of "Elephant-ness"). It is important for RLL to be able to represent them distinctly, yet still record the relations among them. On the other hand, to facilitate interactions with a human user, RLL can accept a vague term (Elephant) from the user or from another unit, and automatically refine it into a precise term. This is vital, since a term which is regarded as precise today may be regarded as a vague catchall tomorrow. Current status: distinct representations pose no problem; but only an exhaustive solution to the problem of automatic disambiguation has been implemented.

7. CONCLUSION

"...in Mordor, where the Shadow lies."

The system is currently usable, and only through use will directions for future effort be revealed. Requests for documentation and access to RLL are encouraged. There are still many areas for further development of RLL. Some require merely a large amount of work (e.g., incorporating other researchers' representational schemes and conventions); others require new ideas (e.g., handling intensional objects). To provide evidence for our arguments, we should exhibit a large collection of distinct representation languages which were built out of RLL; this we cannot yet do. Several specific applications systems live in (or are proposed to live in) RLL; these include EURISKO (discovery of heuristic rules), E&E (combat gaming), FUNNEL (taxonomy of Lisp objects, with an aim toward automatic programming), ROGET (Jim Bennett: guiding a medical expert to directly construct a knowledge based system), VLSI (Mark Stefik and Harold Brown: a foray of AI into the VLSI layout area), and WHEEZE (Jan Clayton and Dave Smith: diagnosis of pulminary function disorders, reported in [*Smith & Clayton*]).

Experience in AI research has repeatedly shown the need for a flexible and extensible language -- one in which the very vocabulary can be easily and usefully augmented. Our representation language language addresses this challenge. We leave the pieces of a representation in an explicit and modifiable state. By performing simple modifications to these representational parts (using specially-designed manipulation tools), new representation languages can be quickly created, debugged, modified, and combined. This should ultimately obviate the need for dozens of similar yet incompatible representation languages, each usable for but a narrow spectrum of task.

ACKNOWLEDGEMENTS

The work reported here represents a snapshot of the current state of an on-going research effort conducted at Stanford University. Researchers from SAIL and HPP are examining a variety of issues concerning representational schemes in general, and their construction in particular (*viz.*, [*Nii & Aiello*]). Mark Stefik and Michael Genesereth provided frequent insights into many of the underlying issues. We thank Terry Winograd for critiquing a draft of this paper. He, Danny Bobrow, and Rich Fikes conveyed enough of the good and bad aspects of KRL to guide us along the path to RLL. Greg Harris implemented an early system which performed the task described in Section 4.1. Others who have directly or indirectly influenced this work include Bob Balzer, John Brown, Cordell Green, Johan deKleer, and Rick Hayes-Roth. To sidestep InterLisp's space limitation, Dave Smith implemented a demand unit swapping package (see [*Smith*]). The research is supported by NSF Grant #MCS-79-01954 and ONR Contract #N00014-80-C-0609.

BIBLIOGRAPHY

Aikins, Jan, "Prototypes and Production Rules: An Approach to Knowledge Representation from Hypothesis Formation", HPP Working Paper HPP-79-10, Computer Science Dept., Stanford University, July 1979.

Bobrow, D.G. and Winograd, T., "An Overview of KRL; a Knowledge Representation Language", 5IJCAI, August 1977.

Brachman, Ron, "What's in a Concept, Structural Foundations for Semantic Networks", BBN report 3433, October 1976.

Findler, Nicholas V. (ed.), *Associative Networks*, NY, Academic Press, 1979.

Genesereth, Michael, and Lenat, Douglas B., *Self-Description and -Modification in a Knowledge Representation Language*, HPP Working Paper HPP-80-10, June, 1980.

Greiner, Russell, "A Representation Language Language", HPP Working Paper HPP-80-9, Computer Science Dept., Stanford University, June 1980.

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

Lenat, D. B., Hayes-Roth, F. and Klahr, P., "Cognitive Economy", Stanford HPP Report HPP-79-15, Computer Science Dept., Stanford University, June 1979.

Minsky, Marvin, "A Framework for Representing Knowledge", in *The Psychology of Computer Vision*, P. Winston (ed.), McGraw-Hill, New York, 1975.

Nii, H. Penny, and Aiello, N., "AGE (Attempt to Generalize): A Knowledge-Based Program for Building Knowledge-Based Program", 6IJCAI, August 1979.

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

Smith, David and Clayton, Jan, "A Frame-based Production System Architecture", AAAI Conference, 1980.

Smith, David, "CORLL: A Demand Paging System for Units", HPP Working Paper HPP-80-8, June 1980.

Stefik, Mark J., "An Examination of a Frame-Structured Representation System", 5IJCAI, August 1977.

Szolovits, Peter, Hawkinson, Lowell B., and Martin, William A., "An Overview of OWL, A Language for Knowledge Representation", MIT/LCS/TM-86, Massachusetts Institute of Technology, June 1977.

Woods, W. A., "What's in a Link, Foundations for Semantic Networks", in D. G. Bobrow & A. M. Collins (eds.), *Representation and Understanding*, Academic Press, 1975.