Stanford Artificial intelligence Laboratory
Memo AIM-293

November 19 7 6

Computer Science Department
Report No. STAN-E-76-58 1

AN OVERVIEW OF KRL, A KNOWLEDGE REPRESENTATION LANGUAGE

by

Daniel G. Bobrow
Terry Winograd

Research sponsored by

Advanced Research Projects Agency
ARPA Order No. 2494

COMPUTER SCIENCE DEPARTMENT
Stanford University

Stanford Artificial Intelligence Laboratory **November 19 76**
Memo AIM-293

Computer Science. Department
Report No. STAN-G-7 6-58 1

AN OVERVIEW OF KRL, A KNOWLEDGE REPRESENTATION LANGUAGE

by

Daniel G. Bobrow
Terry Winograd

ABSTRACT

This paper describes KRL, a Knowledge Representation Language designed for use in understander systems. It outlines both the general concepts which underlie our research and the details of K RL-0, an experimental implementation of some of these concepts. **KRL** is an attempt to integrate procedural knowledge with a broad base of declarative forms. These forms provide a variety of ways to express the logical structure of the knowledge, in order to give flexibility in associating procedures (for memory and reasoning) with specific pieces of knowledge, and to control the relative accessibility of different facts and descriptions. The formalism for declarative knowledge is based on *structured conceptual objects* with associated *descriptions.* These objects form a network of *memory units* with several different sorts of linkages, each having well-specified implications for the retrieval process. Procedures can be associated directly with the internal structure of a conceptual object. This *procedural attachment* allows the steps for a particular operation to be determined by characteristics of the specific entities involved.

The control structure of KRL is based on the belief that the next generation of intelligent programs will integrate data-directed and goal-directed processing by using multi-processing. It provides for a priority-ordered multi-process agenda with explicit (user-provided) strategies for scheduling and resource allocation. It provides *procedure directories* which operate along with process *frameworks* to allow procedural parameteritation of the fundamental system processes for building, comparing, and retrieving memory structures. Future development of **KRL** will include integrating procedure definition with the descriptive formalism.

Daniel Bobrow is affiliated with Xerox Palo Alto Research Center, Palo Alto, California. *This document is also being issued as Xerox PARC report CSL-76-4, and will appear in the journal Cognitive Science, V. 1, No. I, 1977.*

Table of Contents

Acknowledgements:

This paper describes work being done jointly by the Understander Group at Xerox Palo Alto Research Center, and by a research group at the Artificial intelligence Laboratory at Stanford. The work done by Terry Winograd was partially supported by the Advanced Research Projects Agency of the Department of Defense under contract #MDA 903-76-C-0206 at Stanford University. Ronald Kaplan, Martin Kay, David Levy, Paul Martin, Donald A. Norman, and Henry Thompson have played significant roles in its development. We have profited from extensive and insightful comments on earlier versions of this paper by Drew McDermott, Donald A. Norman, Aaron Sloman, and Ben Wegbreit.

This paper is an introduction to KRL, a Knowledge Representation Language, whose construction is part of a long term program to build systems for language understanding, and through these to develop theories of human language use. What we describe is a formal computer language for representing knowledge. It has been shaped by our understanding of what is needed to build natural language understanders, and on analogies with human information processing, particularly in the areas of memory and attention.

Our ideas are in the course of active expansion and modification, and there will be many changes before KRL comes close to our goals. We have implemented a subset of our ideas in a system which we call KRL-0, and the facilities we describe here are those that existed on May 1, 1976. We have conducted some experiments in system building in KRL-0 to test its utility and habitability. Although we have planned a number of features which will be incorporated in its successors, we feel it useful to present our current ideas for discussion and evaluation.

1. Why we are doing it

There is currently no-suitable base on which to build sophisticated systems and theories of language understanding. A complete understander system demands the integration of a. number of complex components, each resting on ones below, as illustrated in Figure 1. Current systems, even the best ones, often resemble a house of cards. The researchers are interested in the higher levels, and try to build up the minimum of supporting props at the lower levels. The standardly available bases (such as LISP, QLISP, CONNIVER, production systems, etc.) are at a low enough level that many middle layers must be built up in an ad hoc way for each project. The result is an extremely fragile structure, which may reach impressive heights, but collapses immediately if swayed in the slightest from the specific domain (often even the specific examples) for which it was built.

TASK DOMAINS:
Travel Arrangements, Medical Diagnosis, Story Analysis, etc

LINGUISTIC DOMAINS:
Syntax and Parsing Strategies; Morphological and Lexical Analysis; Discourse Structures; Semantic Structures, etc.

COMMON SENSE DOMAINS:
Time, Events and States; Plans and Motivations; Actions and Causes; Knowledge and Belief Structures; Hypothetical Worlds

BASIC STRATEGIES:
Reasoning; Knowledge Representation; Search Strategies

UNDERLYING COMPUTER PROGRAMMING LANGUAGE AND ENVIRONMENT
Representation Language; Debugging Tools; Monitoring Tools

Figure 1. A layered view of a language understanding system

Much of the work in AI has involved fleshing in bits and pieces of human knowledge structures, and we would like to provide a systematic framework in which they can be assembled. Someone who wishes to build a system for a particular task, or who wishes to develop theories of specific linguistic phenomena should be able to build on a base which includes well thought out structures at all levels. Jn providing a framework, we impose a kind of uniformity (at least in style) which is based on our own intuitions about how knowledge is organized. We state our major intuitions here as a set of aphorisms, and provide justification and explanation in the body of the paper.

* Knowledge should be organized around conceptual entities with associated descriptions and procedures.
* A description must be able to represent partial knowledge about an entity and accommodate multiple descriptors which can describe the associated entity from different viewpoints.
* An important method of description is comparison with a known entity, with further specification of the described instance with respect to the prototype.
* Reasoning is dominated by a process of recognition in which new objects and events are compared to stored sets of expected prototypes, and in which specialized reasoning strategies are keyed to these prototypes.
* Intelligent programs will require multiple active processes with explicit user-provided scheduling and resource allocation heuristics.
* Information should be clustered to reflect use in processes whose results are affected by resource limitation and differences in information accessibility.
* A knowledge representation language must provide a flexible set of underlying tools, rather than embody specific commitments about either processing strategies or the representation of specific areas of knowledge.

2. Description as the basis for a declarative language

A natural organization for declarative knowledge is to center it around a set of *conceptual entities* with associated *descriptions.* Much of the detailed syntax and data structuring in KRL flows from a desire to explore the consequences of an object-centered factorization of knowledge, rather than the more common factorization in which knowledge is structured as a set of facts, each referring to one or more objects. Objects, relations, scenes and events are all examples of conceptual entities which can be associated with appropriate descriptions in KRL. A description is fundamentally *intensionaf,* where the structure of the description can be used in recognizing a conceptual entity and comparing it with others. The three underlying operations in the system are *augmenting* a description to incorporate new knowledge, *matching* two given descriptions to see if they are compatible for the current purposes, and *seeking* referents for entities which match a specified description. In this section we will describe the forms of description available in the system, the dimensions of matching as we see them, and the basic facilities for context dependent search and retrieval.

2a. Multiple descriptions of conceptual entities

A *description* is made up of one or more *descriptors.* For example, the description associated with a particular object in a scene might include descriptors corresponding to "the thing next to a table," "something made of wood," "something colored green," "something for sitting on," and "a chair". Some of these descriptors express facts which might be thought of as additional propositions about the objects, while others reflect different viewpoints for description by comparison.

The description of a complex event such as ***kissing*** involves one viewpoint from which it is a physical event, and should be described in terms of body parts, physical motion, contact, etc. The descriptors used from this viewpoint would have much in common with those used to describe other acts such as eating and testing someone's temperature with your lips. In the same description, we want to be able to describe kissing from a second viewpoint, as a social act involving relationships between the participants with particular combinations of motivations and emotions. Viewing ***kissing*** in this way, it would be described analogously to other social acts including hugging, caressing, and appropriate verbal communications. In general we believe that the description of a complex object or event cannot be broken- down into a single set of primitives, but must be expressed through multiple views.*

---

*MERLIN (Moore and Newell, 1973) was an early attempt to use multiple viewpoints of this sort.

In addition to containing descriptors corresponding to different viewpoints, a description can combine different modes of description. These include:

- *Assigning an object to membership in a *category* (such as "is a city");
- *Stating its role in a complex object or event (the "destination" of a particular trip);
- * Providing a unique identifier (this includes using a proper name like "Boston");
- *Stating a relationship in which the object is a participant (being "to the North of Providence");
- *Asserting a complex logical formula which is true of the object ("Either this person must be over 65, or a widow or widower with dependent children.");
- *Describing an object in terms of membership in a set, or a set in terms of the objects it contains ("One of the 50 Model Cities");
- * Combining these other descriptors into time-dependent or contingent descriptions ("The place you are today").

In creating a set of descriptor forms, we have been guided by our intuitions about how they will be used in reasoning processes. They represent an alternative to the standard, more uniform notations (such as predicate calculus) which were developed for the purposes of formal logic and mathematics. We believe that it is more useful and perspicuous to preserve in the notation many of the conceptual differences which are reflected in natural language, even though they could be reduced to a smaller basis set.

We expect this to ease the task of designing the strategies which guide the application of declarative knowledge.

We believe that multiple descriptions containing redundant information are used in the human representation system to trade off memory space for computation depth, and that computer systems can take advantage of the same techniques. The choice of where to put redundancy provides further structure for memory, and can be used to limit search and deduction. As a simple example, an understander system might know that every ***plumber*** is a ***person*** and that ***Mary*** is a ***plumber***. The memory unit for ***Mary*** would contain a descriptor stating that she is a ***plumber,*** and would very likely also contain an explicit descriptor stating that she is a ***person*** This is redundant, but without it the system would be continually re-deducing simple facts, since personhood is a basic property often used in reasoning about entities. Memory structure in KRL is organized in a way which makes it possible to include redundant information for immediacy while keeping the ability to derive information not explicitly stated.

2b. Descriptions based on comparison to other individuals and prototypes

In designing KRL we have emphasized the importance of describing an entity by comparing it to another entity described in the memory. The object being used as a basis for comparison (which we call the *prototype)* provides a *perspective* from which to view the object being described. The details of the comparison can be thought of as a *further specification* of the prototype. Viewed very abstractly, this is a commitment to a *wholistic* as opposed to *reductionistic* view of representation. It is quite possible (and we believe natural) for an object to be represented in a knowledge system only through a set of such comparisons. There would be no simple sense in which the system contained a '*definition" of the object, or a complete description in terms of its structure. However if the set of comparisons is large and varied enough, the system can have a functionally complete representation, since it could find the answer to any question about the object which was relevant to the reasoning processes. This represents a fundamental difference in spirit between the KRL notion of representation, and standard logical representations based on formulas built out of primitive predicates.

In describing an object by comparison, the standard for reference is often not a specific individual, but a stereotypical individual which represents the *typical* member of a class. Such a prototype has a description which may be true of no one member of the class, but combines the *default* knowledge applied to members of the class in the absence of specific information. This default knowledge can itself be in the form of intensional description (for example, the prototypical family has "two or three" children) and can be stated in terms of other prototypes.

A single object or event can be described with respect to several prototypes, with further specifications from the perspective of each. The fact that last week Rusty flew to San Francisco would be expressed by describing the event as a typical instance of ***Travel*** with the mode specified as ***Airplane,*** destination ***San Francisco,*** etc. It might also be described as a ***Visit*** with the actor being ***Rusty***, the friends a particular group of people, the interaction warm, etc.

The *further specifications* in a description by comparison can provide more detail to go along with less specific properties associated with the prototype, or can contradict the *default assumptions* which are assumed true in the absence of more specific information. The default for the destination of a 'trip simply specifies that it is some city, and in a

particular event is further specified to be Boston. The default for a trip also includes the fact that the traveller starts from and ends at home, which might be violated in a specific instance. A comparison can be based on an individual rather than an abstract prototype ("He's like Brian, but shorter and with red hair") again with the assumption that the properties of the prototype individual are assumed true of the individual being described unless explicitly counterindicated. It has been pointed out in many places how important it is to make heavy use of typical and expected properties in contexts where the reasoner has incomplete information about the world, and cannot prove logically that a particular individual has a desired property.* It is important to see this analysis as an intuition about how people structure descriptions, rather than as a specific technical device. Many of the mechanisms proposed for in the literature on memory representation (e.g., semantic networks, frames, etc.) can be used in a style compatible with this kind of *inheritance of properties?* We emphasize perspectives as a fundamental part of the notation. Other systems for simulation of human cognitive processing have used similar ideas*** with further specifications which must follow constraints specified in the prototype.

---

*The use of prototypes is the subject of much current research, in computer science (e.g., Minsky, 1975). psychology (e.g., Rosch and Mervis, 1975), and linguistics (e.g., Fillmore, 1975).

**gee Winograd (1975) for a discussion of property inheritance, and Woods (1975) for a discussion of the issues involved in building networks with sufficient intensional information.

● **gchank (1975a) uses conceptual dependency primitives as prototypes, with constraints on objects which can . fill various roles; Norman and Rumelhart (1975) allow constraints on arguments of processes whose prototypes are word definitions.

2c. The detailed structure of units and perspectives

Sections 2a and 2b present our overall notion of the structure of descriptions. In order to better discuss the ways in which descriptions are used, we will introduce in this section some specific notations of KRL. Through the remainder of the paper, terms such as "unit", "perspective", and "description" will be used in the narrower technical sense defined here.

The data structures of KRL are built-of *descriptions,* clustered together into structures called *units,* which serve as unique mental referents for entities and categories. Each unit has a unique *name,* is assigned to a *category type* (see below), and has one or more named *slots* containing descriptions of entities associated with the conceptual entity referred to by the unit as a whole. Slots are used among other things to describe those substructures of a: unit which are significant for comparison. Each slot has a *slotname* which is unique within the unit, and significant only with respect to that unit. One distinguished slot in each unit (named SELF) is used to describe the entity represented by the unit. Associated with each slot are a set of procedures which can be activated under certain conditions of use of the unit. The use of this *procedural attachment* is described further in section 3a. Our convention is to capitalize the initial letter of unit names, but not of slot names.

Each description is a list of *descriptors,* each with a set of associated *features* (discussed at the end of section 2d). There is a limited set of distinct descriptor types (twelve types in the May 1 version), each with a distinct syntactic form. The descriptor type used for description by comparison is called a *perspective.* An entity is further specified in a

perspective by further describing its slots. A perspective is expressed in KRL-0 notation:

(a *prototype* with *identifier*, = *fillerdescription*, . . . *identifier*, = *fillerdescription*,)

where *prototype* names a unit being used as the basis for the comparison, there are indefinitely many pairs of the form *identifier* = *fillerdescription*, and each *identifier* is either a slot-name naming a slot in the prototype unit, or a description which matches only one of the slot descriptions in the prototype unit. Thus the descriptor (a Person) when used to describe an object represents the fact that the object is one instance of the general class *Person* A descriptor (a Person with name = "Joe") implies that *name* is a slot associated with the unit for *Person*, and would be used to describe a particular individual. The descriptor:

```
(a Person with
     name = "Joe"
     (an Address) = " 1004 Main Street" )
```

includes both the name and address information, and assumes that there is only one slot in the unit for *Person* whose description could be matched by the descriptor (an Address). Thus a perspective combines classification with a set of bindings called *fillers*, which establish the correspondence between specific descriptions (often indicating individuals) and roles associated with the prototype unit in general, as indicated by the pairing of the identifiers and additional descriptions.

In Figure 2 we show some simple units which describe Rusty's trip to San Francisco. These illustrate some KRL-0 notation using a simplified example. The overall syntax is like that of LISP, using paired delimiters as an explicit representation of the tree structure.

```
[Travel UNIT Abstract                  ..Travel is the unit name.  Its category type is Abstract .
    <SELF (an Event) >                    . ..description of the Travel unit itself.
                                          Event, Plane, Auto and City are known units
    <mode (OR Plane Auto Bus)>            ..either Plane or Auto or Bus
                                              can fill the slot named mode.
    <destination (a City)>]

[Visit UNIT Specialization             ... a specific category of SocialInteraction
    <SELF (a SocialInteraction)>
    <visitor (a Person)>
    <visitees (SetOf (a Person))>]

[Event 13 7 UNIT Individual            . . ..a specific event described from two viewpoints
    <SELF {(a Visit with
       visitor = Rusty                 . ..The actor is the known unit Rusty
       visitees = (Items Danny Terry)) ..Items indicates at least Danny and Terry are
     (a Travel with                        set elements in this set
       mode = Plane
       destination= SanFrancisco)} >] . ..SanFrancisco is a known unit describing a City
```

Figure 2. KRL Representation of Rusty's Trip to San Francisco

Brackets [...] enclose each unit, and angles <...> enclose each slot in a unit. Each complex descriptor form is delimited by parentheses (...), and braces {...} are used to combine multiple descriptors into a single description. As in LISP, division of text onto separate lines and indentation are used to clarify the structure for human readers; they are not used to convey syntax information to the system. .

Categories of units: The *unit* is a formal data structure in the KRL language for descriptions. It is used for entities at a number of different levels of abstraction -- individuals, prototypes, relations, etc. It can be thought of as a mechanism for providing a larger structure which encompasses a set of descriptions, relating them to a set of procedures. Each unit has a category type, selected from: *Basic, Abstract, Specialization, Individual, Manifestation, Relation, and Proposition.* The category types determine certain modes of operation for the basic system procedures which manipulate descriptions.

Abstract, basic, and specialization: Units of these three types are used for categories such as ***Person, Integer, MakeReservation***, etc. These units are used principally as prototypes for perspectives; the distinction between them is used primarily by the matcher.

Basic categories represent a simple non-overlapping partition of the world into different kind of objects (such as ***dog bacteria, . ..).*** The matcher assumes that no individual is in two distinct basic categories. Therefore, quick tests of basic category match or conflict can be used in a many cases to decide whether a specific object fits a description. This use of simple disjoint categories corresponds to the use of selection restrictions as proposed in some linguistic and semantic theories* and data types in programming.

*This includes much of the work on semantics associated with transformational grammar (e.g., Katz and Fodor. 1964), AI formalisms such as conceptual dependency (Schank, 1975a) and most forms of case grammar (See Bruce, 1975 for a summary).

Specializations represent further distinctions within a basic category (such as ***Poodle,*** or ***E.Coli***). A specialized prototype will have descriptions and procedures associated with it which are more specific than those for a basic category. In general, they are more useful for their procedural attachment and described properties than for any uniform treatment by the matcher. The description of a specialized prototype can indicate a *primary* perspective which describes it as a subclass of some basic category or other specialization. The partial tree formed by these primary links* is used by the matcher in comparing individuals from two categories which are not explicitly comparable.

*This tree corresponds to a simple generalization hierarchy, as discussed in Winograd, (1975).

An abstract category (such as ***action*** or ***living thing)*** primarily serves as a way of chunking a set of descriptions and procedures to be inherited by any entity described by a perspective for which the abstract unit is a prototype. Very general problem solving information will often be attached at this level of abstraction. There is no commitment in KRL as to what sorts of concepts in the domain should be represented at what level of categorization. The specific three types are based on psychological studies (Rosch 1975,

Rosch & Mervis 1975) that human reasoning makes extensive use of a layered system of categories. The choice of whether a particular prototype (such as *Person* or *Visit*) should be basic, abstract, or a specialization depends on the way in which descriptions are built up and used in matching. What is provided is a mechanism by which the careful use of levels can result in achieving many of the efficiency benefits of semantic marker mechanisms and classification trees.

Individuals: The KRL matcher and other primitive mechanisms for building descriptions assumes that different individuals are different unique entities in the world being modelled. For example, no individual can match (in a simple sense) a different individual. An inconsistency is signalled whenever there is an attempt to use pointers to two different individuals as descriptors in a single description. However, there are no built-in assumptions about how individualhood should be assigned. The definition of what should constitute an individual within a domain is relative to a particular set of reasoning purposes. As a simple example, in the air travel domain, a particular flight (including date) could be an individual, with the flight number as a property (filling a slot), or, alternatively, each flight in the schedule (by number) could be treated as an individual, with a particular flight instance represented as a manifestation (see below).

Manifestations: Often it is useful to group together a set of descriptions which belong to some individual. There are three main cases in which we anticipate this need, and units with category type *manifestation* can be used for all of them:

. **Further specified individuals:** A manifestation can be used to provide a single memory unit (for purposes of retrieval and context dependent description) containing a set of descriptions belonging to an individual within one context. For example, we might separate out the physical properties of an object for which we also have functional or historical descriptions, or the description of some person as a scientist from the description of that person as a friend.

Contingent properties: An individual can be described using time-dependent descriptions without creating a separate manifestation. However, it is often useful to collect a set of descriptions which are true at some time (or in some hypothesized world) and treat them as *time-independent descriptions* of a manifestation which represents the individual at that particular time.

Ghosts: A representation must enable us to describe entities whose unique identity is not known. There are many cases in which we may know many properties of some object without knowing which of the known objects in our world it is. Such objects have at times been called "formal objects" (Sussman, 1973) and "ghosts" (Minsky, 1975). A standard detective story plot involves knowing that one of the people in a house is a murderer, knowing many properties of the murderer, and not knowing which individual it is. The unit used to represent the murder is a manifestation which has no associated individual.

Relations and propositions: An abstract relationship, such as the relative magnitude of two numbers can be described using the ideas of slots and description we have used so far. There is a unit, with category class *relation,* which represents the relationship (or predicate) as an abstract mapping; a *proposition* unit represents each instantiation of the relationship. The truth value of a proposition is specified explicitly rather than being determined as an implicit consequence of its existence in the data base.

2d. The family of descriptors

Each descriptor in a description is an independent characterization of the object associated with the description. The variety of descriptor types corresponds to the notion of natural description discussed above. Each descriptor type is intended to express a different mode, of describing conceptual objects. The syntax of descriptors depends on key words (such as *a, the, front, which)* based on analogy with simple English phrases. They are mnemonic indicators for a set of precisely defined structures within the formalism.

This set of descriptors was not designed with the goal of boiling everything down to the smallest possible set of mechanisms. On the contrary, it is based on an attempt to provide a simple and natural way of stating information conceptualized in different ways. There is a great deal of overlap. For example, the notion of "bachelor" might be represented in any of the following ways:

- There could be a prototype unit for *Bachelor*, with an individual described as

  (a Bachelor with...)

- Bachelorhood could be represented indirectly by having a prototype for *MalePerson* and *Adult,* and a predicate for *IsMarried*, and using the description:

  {(a MalePerson) (an Adult) (NOT (which IsMarried))}

- There could be a unit representing a *Marriage* with slots for the *malePartner* and *femalePartner*, and a description:

  {(a MalePerson) (an Adult)
  (NOT (the malePartner from (a Marriage)))}.

- There could be a one-place predicate, *IsBachelor*. The predicate definition might (but need not) include a special procedural test for bachelorhood. An individual would then be described using the *predication*

  (which IsBachelor).

These KRL forms are described in general in Figure 3. No one of these forms is automatically primary. All of them could coexist, and be defined in terms of each other. The system provides the necessary reasoning mechanisms to interrelate the different forms in which essentially equivalent information could appear, and the hope is that additional knowledge (especially procedural knowledge) which is best stated with respect to any one form can be represented directly. Our intuition leads us to believe that prototypes and perspectives will most often serve as the fundamental organizing representation, with the others serving to provide secondary information.

Some descriptor types in KRL-Q

Form of each entry

name of descriptor type: *BNF specification of its form*

Informal description of what this descriptor type is used for

examples in KRL notation

---

1) direct pointer: ***unitName*** **|** ***numberOrString*** **|** **(QUOTE** ***lispObject)***

A pointer to units, or to data directly in the description. Provides a unique identifier (this includes using a proper name like "Boston")

Block1 7, PaloAlto, 356, "a string", (QUOTE (A PIECE (OF LIST) STRUCTURE)))

---

2) perspective: (a *prototype* **with** *identifier,* **=** *filler, . . . identifier,* **=** *filler,)*

*Assigns an object to membership in a category (such as "city**). A comparison of the current object with the "prototype", with slots further specifying this object*

(a Trip with destination = Boston airline = TWA)

---

3) specification: (the ***slotSpecifier*** **from** *view target Description)*

*Specifies the current object in terms of its role in a perspective of prototype: "view". States a role in a complex object or event (e.g., the **destination'* of a particular trip);*

(the actor from Act {Event17 (a Chase with quarry = (Car22 (a Dodge)})})

---

4) predication: (which ***predicateName*** **.** ***predicateArgs)***

Describes a relationship in which the object is a participant (being "to the North of Providence"). Defined in terms of a specification. A way of specifying an object in terms of a relation and arguments; allows special procedural attachment.

[which Owns (a Dog)) (which IsBetween Block17 (a Pyramid))

---

5) logical boolean: (OR . ***booleanArgs)*** **|** **(XOR .** ***booleanArgs)*** **|** **(NOT** ***booleanArg)***

Simple logical connectives. A description is an implicit AND of descriptors, thus AND is not needed

(OR (a Dog) {(a Cat)(which hasColor Brown)})
(NOT (a Pet with owner = (a Student))

continued...

6) restriction: (theOne *restrictionDesc*)

Marks the **enclosed** *description as being sufficient to refer to a unique object in context*

(theOne {(a Mouse)(which Owns (a Dog)))

---

7) selection: (using *selectionDesc*
select From *selectionPattern* ~ *selection*
***selectionPattern* ~ *selection* ...**
otherwise *defaultSelection*)

This is a declarative form corresponding to CASE or SELECT statements in programming languages.

```
(using     (the age from Person ThisOne)
      selectFrom  (which isLessThan 2 ) ~ Infant
                  (which isAtLeast 12)  ~ Adult
      otherwise Child)
```

---

8) set specification: (SetOf *setElementDescription*) | (In . *setDescription*)
| (Items . *setElements*) | (NotItems . *setElements*) | (AllItems . *setElements*)
| (Sequence . *setElements*) | (ListOf . *setElements*)

These descriptors allow specification of partial information about sets, sequences and lists. Describes an object in terms of membership in a set, or a set in terms of the objects it contains

```
(SetOf {(an Integer)( which hasFactor 2)}) ...all elements are even numbers
(Items 2 4)                                . . . At least 2 and 4 are in this set
(AllItems 2 4 64 {(an Integer)( which hasFactor 3)}) . . . a four element set
(NotItems 51)                              . . . 51 is not in this set
(In {(SetOf (an Integer)) (Items 2 5 8) (NotItems 4)})
        ... describes an object in a set of integers containing at least 2 5 8, and not 4
```

---

9) contingency: (during *timeSpecification* then *contingentDescription*)

Specifies a time (or hypothetical world) dependent description.

(during State24 then (the topBlock from (a Stack with height = 3)))
... part of the description of an individual block

(during (a Dream with dreamer = Jacob) then (an Angel))
... part of the description of an individual person

Figure 3. Different descriptor types in KRL-0 (a partial list)

In order to demonstrate the different uses of these descriptors, we give here a more extended example. It is based on a hypothetical system that acts as a travel assistant, making reservations and computing costs of trips. As with the example above, this is greatly oversimplified and is not intended as a careful analysis of the travel domain. Figures 4 and 5 show a number of units with which we construct our examples. There is a basic unit for *Person*, and the different things we might know about a person are further grouped according to the ways they are used in this system. A person viewed as a *Customer* has a set of properties different from a person viewed as a Traveller. These specialized units can share information which is generally true of people. The decision of

```
[Person UNIT Basic
      <SELF >      ... There is no description for Person since it is not further
                   analyzed in the travel planning task
      <firstName (a String)>
      <lastName (a String) >
      <age (an Integer)>      . ..A person's age in years; distinct from age slot in Traveller.
      <hometown {(a City) PaloAlto ; DEFAULT}>
                  . ..semicolons are used to attach features to individual
                  descriptors. Unless otherwise specified, the hometown of a
                  Person will be assumed to be PaloAlto (which is in turn a unit
                  described as a City).
      <streetAddress (an Address)>]

[Traveller UNIT Specialization
      <SELF (a Person)>      . ..a specialization of the unit Person
      <age { (XOR Infant Child Adult)
                  . ..the age for any specific Treveller will be one of these units
            (using (the age from Person ThisOne)
               selectFrom (which isLessThan 2)      ~ Infant
                          (which isGreaterThan 11)  ~ Adult
               otherwise Child)}>
                  . ..the selection descriptor provides a way of determining which
                  case this is from the age field in the view of this Traveller
                  viewed as a Person
      <preferredAirport {(an Airport)
                  (In (the localAirports from City
                        (the hometown from Person ThisOne))); DEFAULT}>]
                  . ..the airport is found in the set of preferred airports found
                  from the City in the hometown slot from a view of this
                  Traveller viewed as a Person

[Customer UNIT Specialization
      <SELF (a Person)>
      <billingAddress {(an Address)
                  (the streetAddress from Person ThisOne) ; DEFAULT}>
      -<credit (a CreditCard)>]
```

Figure 4. Some units for an airline travel system

```
[Airport UNIT Basic
      <SELF >
      <location (a City)>]

[City UNIT Basic
      <SELF >
      <localAirports {(SetOf (an Airport with location = ThisOne));DEFAULT }>]

[PaloAl to UNIT individual
      <SELF (a City with localAirports = (items SJO SFO OAK))>]
            ...these airports are units too

[SJO UNIT individual
      <SELF (an Airport with location = SanJose )>]

[UniversalCharge UNIT--individual
      <SELF (a CreditCompany )>]

[Magnitude UNIT Relation
      <SELF (an ArithmeticRelation) TRIGGERS (ToTest some LISP code inserted here)>
      <greater (a Quantity)>
      <lesser (a Quantity)>]

[IsGreaterThan PREDICATE Magnitude greater lesser]
            ... defines the Predication IsGreaterThan with focus being the slot greater in Magnitude;
            ... the argument which follows the predicate is to fill the lesser slot
            ... used in the form (which IsGreater Tban 2) equivalent to the form
            ... (the greater from Magnitude (a Magnitude with lesser = 2))

[IsLessThan PREDICATE Magnitude lesser greater]
            ... a second predication based on Magnitude with focus on lesser
```

Figure 5. Some units and predicates used in travel information

how-to group the slots which make up units is up to the programmer and the purpose of the representation. in this example, *age* as associated with a traveller is one of *Infant, Child,* or *Adult,* the necessary distinctions for fare determination, while age for a *Person* is an integer.

Figure 6 shows a description of an individual traveller, GOO43, described from two perspectives using these basic units. These three figures use of a number of different types of descriptors, including specifications, set descriptors, predications, and units representing individuals. individuals are either LISP objects (such as the string *"Juan"* and the integer *3*) or units, such as *UniversalCharge.* Specifications provide a way to refer to fillers in perspectives associated with either the unit in which they appear, or other' units. The descriptor *(the age from Person* GO043) would refer to the age of the individual referred to by the unit named *G0043* when viewed as a *Person* (and thus is an integer),

while *(the age from Travelfer G0043)* refers *to* his age as a *Traveller*, and thus is one of *Infant, Child,* or *Adult.* Specifications can be nested. The special unit *ThisOne is* interpreted to refer to the entity being described when the description is used as a prototype. Thus, for example the descriptor:

(the preferredAirports from City (the homeTown from Person ThisOne))

which appears in the unit for *Travelfer* includes a nesting of one implicit and one explicit target. The descriptor *(the hometown from Person ThisOne)* is interpreted as referring to whatever fills the *hometown* slot in a perspective whose prototype is *Person* The descriptor *(the localAirports from City {...})* is based on the unit for *City,* which has a slot *localAirports.* This descriptor assumes that the object which is the *hometown* specified by the embedded specification can be viewed as a *City.*

There are a group of descriptors based on sets which have the obvious intuitive interpretations. In the example, the *localAirports* for a *City* are described as a set each of whose members is an *Airport.* The default for *preferredAirports* is described as an unspecified member of the set of local airports for the hometown. In the case where a town had only one airport, this could be used directly to find the departure airport.

The predication *(which IsLessThan* 2) is a descriptor using the predicate *IsLessThan* which relates pairs of numbers, and in turn is based on a unit *Magnitude* Predicates are defined with respect to a unit in which the arguments to the predicate are among the slots. Two different predicates are defined in terms of the *Magnitude* relation, as shown in Figure 5. This unit includes in its *SELF* slot a procedure to test for the truth of the relation.

```
[G0043 UNIT individual
      <SELF {(a Person with
               firstName = "Juan"
               lastName = {(a ForeignName)
                            (a String with firstCharacter = "M"})
... These descriptors give partial information.  A unique string need not be specified
               age = (which IsGreaterThan 2 1))
             (a Traveiier with
               preferredAirport = SJO
               age = Adult)
             (a Customer with
               credit = (a CreditCard with
                          company = UniversalCharge
                          number = "G45-7923-220"))}>]
```

Figure 6. A unit describing a specific traveller

A predication is always used in describing one specific argument, as opposed to a proposition relating several variables in formal logic, which states the relationship without focusing on any one argument. Figure 7 shows another example of multiple predicates based on a single unit. In some cases, the different predicates focus on different parts of

the total relationship, as in the case of *HusbandOf* and *IsMotherOf* which relate different slots in the *Family* unit. In other cases, they simply provide different points of view by choosing a different argument as the implicit primary argument, as in the difference between *IsHusbandOf* and *IsWifeOf*. Each predicate can have associated procedures for proving and matching both for the relationship as a whole, and the particular focus of its use. Except for this procedural attachment, and for economy of writing, predicates and predications can be replaced in a uniform way by specifications. For example, the following two descriptors are equivalent based on the definitions in Figure 7:

(which IsHusbandOf Mary)
(the maleParent from (a Family with femaleParent = Mary))

```
[Family UNIT Abstract
      <SELF {}>
      <femaleParent {(a Person)(a Female)}>
      <maleParent {(a Person)(a Male)}>
      <children (SetOf (a Person))>]

[IsWifeOf PREDICATE Family femaleParent maleParent
      (TRIGGERS (ToTest some LISP code))]   ...only this one has a special trigger

[IsHusbandOf PREDICATE Family maleParent femaleParent]

[IsMotherOf PREDICATE Family femaleParent (in children)]
```

Figure 7. Multiple predicates defined on the basis of one relation

---

Features and meta-descriptions

Often it is important to represent knowledge about knowledge. KRL allows any descriptor to have associated with it a *feature* (a lisp atom) or a *meta-description* (a full-fledged unit characterizing the descriptor viewed as a piece of knowledge). In the example of Figure 4, there are descriptors marked with the feature DEFAULT, indicating that they can be assumed valid in the absence of other information, but should be superseded by any other information, and should not be used in looking for contradictions. Two other features are used in standard ways by the matching, searching, and data adding routines: CRITERIAL, indicating the set of descriptors whose satisfaction can be counted as proving a match, even if there are other descriptors around; and PRIMARY indicating the primary perspective for inheritance of properties in a hierarchy.

The ability to associate a complete unit at this meta-level makes it possible to have representations which include facts about facts (e.g., justifications, histories of when things were learned or inferred, interdependencies between assumptions, etc.). We have not yet worked out a standard notation for use at this level. As we build different domain programs, we will develop a set of standard units for talking about descriptors and propositions, and a set of facilities for modifying and using them when appropriate.

2e. Description matching as a framework for reasoning

We believe that reasoning is dominated by a process of *recognition* in which new objects and events are compared to a stored set of expected prototypes.* The key part of a recognition process is a *description matcher* which serves as a *framework* for comparing descriptions. We have intentionally used the term *matching* for a range of functions which are broader than its standard use in AI languages.

---

*This basic approach has been advocated as "frame theory" by Minsky (1975) and Winograd (1974 - Lecture 1), as "scripts" by Schank (1975b), and as "Beta-structures" by Moore and Newell (1973). There are many related notions in the current AI literature.

First, we have separated the issue of indexed data base retrieval from that of matching two descriptions. Section 2g deals separately with the processes of indexing, context searching and retrieval. Our matcher takes two inputs: a pattern, and a specific object to be tested. Secondly, we use the abstract concept of matching in a very general sense, to include all sorts of reasoning processes which are used to decide whether a given entity fits a given description: Much of what is usually thought of as "deduction" comes under this heading, as do the notions which Moore and Newell (1973) have called "mapping".

We think of the matching process as a *framework* because the user has choices along several dimensions which determine how the matcher operates. In its simplest form, the matcher compares the exact forms of two given representational structures: at the other extreme it guides the overall processing of the system. The matcher may use the semantics of descriptors as well as their syntactic form to decide whether two descriptions match. The matcher may search for a referent of a description (in order to use its properties), invoke special match procedures associated with a descriptor type or a specific pattern, and invoke a general reasoning process to search for chains of implications.

In extending the notion of matching, we have adopted and extended ideas which have been implemented in a variety of systems. Our attempt has been to integrate them into a coherent framework which gives the user of KRL a choice of the strategies best suited to the specific task. The choices can be viewed as representing four interacting dimensions. We will first list these dimensions and give examples of the choices made in well known match systems, then describe the range of possibilities provided for each in KRL.

Subtasks: In all but the most trivial matching operations, the pattern and the *datum* to which it is compared are complex objects with an internal structure. The match is carried out by setting up a series of subtasks, each of which matches one piece of the structure of the pattern against a corresponding piece of the datum.. In the case of simple syntactic matchers (as in the AI languages) the division into subtasks is a direct reflection of the syntax of the structures -- if the pattern and datum are represented as lists, each subtask involves matching one element of the pattern list against the corresponding structural element of the datum list, using a recursive application of the same matching algorithm.

Terminals: In applying a match process recursively to a complex structure, there is a choice of where the recursion "bottoms out". In matchers operating on individual data structures (such as *assertions* in the AI languages, or *well-formed-formulas* in the unification algorithm of a logic-based system) there is a natural set of terminals provided by the underlying language. In a LISP-based system, pattern variables and atoms form the terminals -- the task of matching them is not done by recursively setting up match subtasks, but by calling the appropriate procedure for deciding the match for that kind of object directly. Even though atoms have an internal structure, they conventionally serve as terminals. In logic, the predicate, variable, constant, and function symbols form a set of terminals. In simple network matchers, the individual links (and their labels) usually serve as terminals.

Results: The simplest kind of result from a matching process is a binary answer -- MATCH or FAIL. Most matchers add to this some kind of mechanism for returning a set of associations between variables in the pattern and constants (or other variables) in the datum, either as an explicit output or implicitly through side effects.

Process: There are a number of control questions in deciding how the match process should proceed. These include deciding what subtasks to carry out in what sequence, and when the match as a whole should be stopped. In the simplest case, the subtasks are taken serially, usually in the sequence provided by the syntactic structure of the pattern. The process continues until ail subtasks have succeeded, at which point it returns the variable assignments, or until any subtask fails, at which point the entire match fails.

In a multiprocess system, there are additional strategy decisions in choosing when to carry out subtasks in serial or parallel. Further, since the match process as a whole may be only one of several competing processes (for example several patterns being compared for "best match") there are choices of when the match process should be suspended or resumed, and how it should compete for processing resources. At a still more sophisticated level, there can be sharing of overlapping subtasks between two match processes.

The Match framework in KRL

We provide a framework for carrying out a match process, and an appropriate set of building blocks from which a matching strategy can be constructed within this framework for-a specific user or domain or process. Section 3c describes in more detail how such "procedural parameterization" is done. We do not believe that any one combination of features will provide a universally applicable matcher. In the course of working with KRL, we plan to experiment with and develop a set of generally applicable strategies which can be used in building a user-tailored match process. The following list of extensions is intended to give some feeling for the scope and variety of issues we believe must be dealt with. At the moment, what exists in KRL-0 is the framework into which they will be integrated, and a set of simple strategies which handle straightforward cases. Figure 8 contains some simple data which will be used in illustrating some problems in matching.

Matching multiple descriptions: The pattern and datum in a KRL match are both descriptions which may contain any number of individual descriptors. In order for a match to be completed, all of the descriptors in the pattern should be satisfied in some way by the datum. But there is no simple sense in which the sequence of descriptors in one can be set into correspondence with the sequence of descriptors in the other. The matcher includes a set of strategies for *alignment* of descriptors. If, for example, both pattern and datum contain a descriptor which is a pointer to an individual, these two individuals will be compared and the match will succeed or fail depending on whether they are identical. If both pattern and datum contain perspectives with the same prototype, the two perspectives (including all of the filler pairs) will be compared in detail. If pattern and datum each contain a perspective whose prototype is a basic unit, those two prototypes will be compared. If the pattern contains a logical descriptor (such as a NOT) and its argument corresponds in one of these simple ways to a descriptor in the datum, the two will be compared.

```
[Cat UNIT Basic
      <SELF {(an Animal)(a Pet)}>]

[Dog UNIT Basic
      <SELF {(en Animal)(a Pet)}>]

[Pluto UNIT Individual
      <SELF (a Dog)>]

[Mickey UNIT Individual
      <SELF (which Owns Pluto)>)

[Minnie UNIT Individual
      <SELF (which Owns (a DogLicense))>]

[Ownership UNIT Specialization
      <SELF (a State)
          TRIGGERS (ToEstablish
              (AND (Match \(the possession) \(a Dog)) ...a LISP expr
                   (Match \( the owner)   ... \ prefixes a KRL expression
                              \(which Owns (a DogLicense with
                                        licensed = (the possession))))))>
      <owner (a Person)>
      <possession (a Thing)>]

[Owns PREDICATE Ownership owner possession]

[DogLicense UNIT Specialization
      <SELF>
      <licensed (a Dog)>]
```

Figure 8. Sample knowledge base used in matching

The algorithm for alignment makes decisions both about what subtasks will be attempted, and in what order they will be done. In the default algorithm built into the matcher, only those subtasks which can be set up simply (like those listed above) will be tried. The order in which these simple strategies will be tried can be determined by the user, and they can be intermixed with user defined strategies. The simple matching strategies handle a large number of the typically occurring cases, but do not account for all possible ways in which two descriptions might be matched. Whenever there is a descriptor in the pattern for which no simple alignment can be found, the system looks for a strategy program provided by the user for this specific match (either written for the special case, or chosen from the set of building blocks). The following examples illustrate possibilities for setting up an appropriate subtask for testing whether a pattern descriptor is matched.

Using properties of the datum elements: Consider matching the pattern descriptor *(which Owns* (a *Dug))* against a datum which explicitly includes a descriptor *(which Owns Pluto)* The SELF description in the memory unit for Pluto contains a perspective indicating that he is a dog. In a semantic sense, the match should succeed. It can do so only by further reference to the information about Pluto. *The KRL matcher can vary the level of what it considers to be terminals.* Faced with comparing a descriptor to an individual (which corresponds loosely to an atom or constant in other matchers) it can set up a subtask of matching the pattern descriptor against the description stored in the unit for that individual. This is done only when the alignment strategies cannot find an appropriate descriptor in the datum, but can find a pointer to an individual.

This extension is naturally recursive. In matching the descriptor *(which Owns (a Dog))* against the unit *Mickey* we first need to use the description within the *Mickey* unit to find *(which Owns Pluto),* then to use this we need to further look within the *Pluto* unit. The problem of finding the correct information in a general way has been called the "symbol mapping problem'* (Fahlman, 1975), and has been handled to some degree by matchers based on networks rather than propositions (for example, Nash-Webber 1975. Hendrix 1975). These matchers do not have a rigid notion of scope and terminals, since matching follows open-endedly along links, rather than operating within a specified formula.

Using deduced properties: The example above assumes that the desired properties are explicit in the data base, but are not local to the datum being matched. The matcher sets up a subtask which is a recursive call to the same matching process, with an expanded datum to work on. A further generalization of recursive subtasks in the matcher allows it to set up subtasks which are not primitive match operations or recursive calls to the match functions, but which require that a needed property be derived from known facts about the elements. This is the province of a theorem prover.

As a simple example, we might want to match *(which Owns* (a *Dog))* against *Minnie* having as our description of Minnie *(which Owns (a DogLicense))*, and having other information asserting that only dog owners own dog licenses. The appropriate subtask to set up is not one which can be done as a simple match operation, but one corresponding to the goal of proving that a dog license owner is a dog owner. This kind of subtask requires general capacities for reasoning and deduction. There is a search problem in deciding what alignments should be tried (which of the conclusions the system should try to prove on the basis of which parts of the available description). The KRL matching

framework does not include any automatic mechanisms for making these decisions, but it does provide a natural superstructure in which specific deductive goals arise and, in the overall flow of control, it is typical that deduction tasks arise as subtasks called by the matcher.

In our example of Figure 8 we have associated a' specialized procedure with the general unit for *Ownership*. In a realistic system, it would be associated with the combination of the concepts *Ownership* and *Dog*, using the index mechanism described in section 2g.

Specialized procedures for subtasks: The previous example involves setting up subtasks which are not simple recursive applications of the matcher. The match framework provides for a general capability for calling arbitrary procedures as subtasks of a match. This can including setting up non-serial control regimes for running subtasks in parallel.

We believe that the most important weapon for attacking the combinatorial problems which arise in matching and deduction is the ability to attach specialized matching procedures to descriptions and units. When the matcher is faced with a pattern and datum which cannot be simply matched, it looks in several places for specific procedural information telling what to do to decide what the result of the match should be:

- There can be procedures associated with general types of alignment -- for example, the user can provide a special procedure to be used when a perspective with its prototype in some specified set can be matched against another perspective with a prototype in that set.
- Procedures can be associated with a unit, to be used whenever a perspective having that unit as its prototype appears as a pattern descriptor. These procedures can either be general (to match the perspective as a whole) or associated with specific slots.* The **ToEstablish** procedure associated with *Ownership* in Figure 8 is an example of such a procedure.

---

*This is a generalization of the notion of active elements in patterns, as in the elements originally called *actors* by Hewitt (1972).

Interleaving a match with ongoing processes: In the standard notions of matching, a match is a unitary process with respect to the rest of the system. The matcher is called as a subroutine, and other processing continue's when it is done. In using a matcher as the basis for a general process of reasoning by recognition, this is not an acceptable strategy. The attempted match of a "frame" or "script" or "schema" begins when its presence is first conjectured, but may continue through a series of further inputs and other processing by the system. This is a natural extension of the ability for specific match situations to set up arbitrary subtasks as described above. Rather than calling a specific subtask and waiting for its answer, the specialized procedures can start up tasks (in a co-routine fashion) which will direct the processing of the system, while the match process remains in the background, waiting for the appropriate information to be found.

There are a number of ways the processing can be organized. In a simple case, the matcher can set up a demon waiting for each piece of information it needs (for example, a demon for each slot to be filled in a perspective) and simply resume the normal processing, waiting for the information to come in. In other cases, the specialized match procedures might start up information-seeking processes (such as asking a question of a user in a natural language system, or doing a visual scan in a vision system) in addition to the demons, so that the attempt to match. the pattern serves as a driving force for deriving new possibly relevant information.*

*Some of the possible methods are discussed in Minsky (1975), Kuipers (1975). and Bobrow et al. (1976).

Admission of ignorance and partial results: The KRL matcher is designed to distinguish among four possible results, both in reporting the result of the match as a whole and using the results of subtasks. In addition to a result of success, it separates two kinds of failures -- where the pattern demonstrably does not match the datum; and where there is insufficient evidence. The case of insufficient evidence is further distinguished according to whether the matcher has been limited by resources it has used in the match, or the matcher failed to decide after trying every strategy it knew.

The general multi-process capabilities of KRL can be used to suspend a match process and return a partial result. Thus, a match can be started, and after some amount of processing (using the resource limitation mechanisms described in section 3b) if no definite success or failure has occurred, it can return a result of "don't know yet". The process using the match can decide whether to accept this as sufficient for an assumed "yes", or to consider it a "no", or to abandon whatever it was doing for lack of sufficient information, or to resume the match process, giving it more resources.

Resource limitation and pinpointing of further problems: In a case where the processing so far has not produced a definite answer, the matcher should be able to return specific details in addition to the result of "don't know yet". Given the problem of matching *(which Owns (a Pet))* against *Mickey*, with sufficient resources (and the data of Figure 8), it could answer "Yes". With less resources, it could answer "Yes, if *(a Dog)* matches *(a Pet)*', and with still less, "Yes, if *Pluto* matches *(a Pet)*'. In general we want to limit the depth of the reasoning, and have the matcher return a list of yet unsolved problems if no definite answer has been found within the limitation.* We hope to integrate this mechanism with the means of returning bindings relating specific elements in the pattern and those which matched them in the datum. As of the current version, only the "hooks" for calling these mechanisms exist, and no details have been filled in.

• This is similar to the idea of residues proposed by Srinivasin (1976) in MDS. a language with many similarities to KRL in both goals and mechanisms.

Evaluation of the quality of a match: Many uses of the matching paradigm are not simple cases of matching a single pattern against a single datum, but involve finding the *best match* among a set of patterns. An example is the matching of a set of disease patterns against a specific symptom set in doing medical diagnosis. The matcher should have some way of assigning values on some scale to the individual parts of a match' and combining these to return a "goodness measure" chosen along a scale of values, rather than just "yes", "no", or "I don't know".

Along with a value for how good the match seems to be, there should be a separate value for how reliable the information is, depending on how much work has been done, and perhaps on reliability measures stored with the information used. This can be combined with progressive deepening' such that each time a match process is resumed, it can further evaluate the match, updating its factor for the goodness of fit, and increasing the factor for reliability of the knowledge.* As with the binding of specific elements, we have so far only provided a place in the framework where such measures could be used.

---

*A simple version of goodness evaluation is implemented in MYCIN (Shortliffe, 1976), and Rubin (1975). There has been some work (e.g., Colby, 1973) on the evaluation of reliability.

Interaction of multiple matching processes: In looking for a best match, it is useful not to think of the separate matching operations as independent, but to allow interactions between them. A simple level of interaction occurs in operating them in a progressive deepening mode, and using some sort of heuristic strategy (e.g., best first generation) to decide which alternative to pursue at each step. At a deeper level, there can be additional information associated with specific ways in which a pattern fails to match (an *error pointer*), which indicates a specific alternative or provides direct information on the goodness of other matches being simultaneously attempted.* This capability should include the potential of triggering new patterns as candidates for a match, on the basis of new data turned up in the match process.**

---

*See Minsky (1975) and Kuipers (1975) for an extended discussion of the ways in which systems of patterns can be linked into a network.

**See Rubin (1975) for a discussion of this kind of triggering in medical diagnosis.

Forced match: A matcher can be operated in a mode in which the question instead of "Does this match?" is "What would you have to believe in order to make this match?".* If asked to match *(which Owns (a Cat))* against *Mickey,* instead of responding with failure, it should return "You have to view a *Dog* as a *Cat*". This is a natural extension for the KRL matcher, since there is a general facility by which the user can provide procedures for alignment and the treatment of types of match (i.e. what to do when you try to match two different individuals).

---

*The general issue of forced matching (or mapping) is developed in Moore and Newell (1973).

Using individuals as patterns: Given the ability to do forced match and to return an indication of the differences, the matcher can be used in a mode where the pattern is an actual individual' rather than a general prototype. The result of matching two individuals would be a specification of the ways in which they differ. This needs to be combined with a further mechanism (associated with resource limitation) which heuristically guides the choice of which properties to follow up and which to ignore. We hope that this style of matching, together with a basic commitment to description by comparison will provide us with a strong base for doing more interesting sorts of reasoning by analogy.

2f. Chunking of knowledge, accessibility and redundancy

One of the fundamental problems in artificial intelligence is the "combinatorial explosion'*. A large knowledge base provides an exponentially expanding set of possible reasoning chains for finding desired information. We believe that the solution to this problem must be found by dealing with it directly through explicit concern with the *accessibility* of information. The representation language must provide the user with a set of facilities for controlling the way in which memory structures are stored, so that there will be a correspondence between "salience" or "relevance" and the information accessed by procedures for search and reasoning operating under processing resource limitations.

Much of the current research on memory structures in artificial intelligence deals with ways of organizing knowledge into *chunks* or clusters which are larger than single nodes 'or links in a semantic net or formulas in a logic-based system. This is particularly important in reasoning based on prototypes' using description by comparison, in which typical properties are assumed true of an object unless more specific information is *immediately available.* It is also necessary in recognition, where identification of an object is based on recognizing some set of *salient* properties, and in analogies, where only the *relevant* properties of one object are inherited by the other.*

---

*For a general discussion of these issues, see Bobrow and Norman (1975). A system which tries to distinguish the salience of different descriptors is described in Carbonell and Collins (1974).

The KRL data structures were designed to be used in processes which are subject to resource limitation and differences in accessibility. Two forms which are equivalent in a strict logical sense are not at all equivalent if used by a processor which takes different numbers of steps to come to the conclusion, and which may well be stopped or suspended before reaching completion. In KRL the unit is treated as a basic memory chunk, and processes such as the matcher operate differently when using information within a unit' and when retrieving information from a unit being pointed to or referenced by it. By making explicit decisions about how to divide information up between units, the user has structural dimension of control over the matching and reasoning processes.

Figures 9 and 10 illustrate the use of redundant information in building up descriptions to be used in matching. Figure 9 gives an example of a set of facts representing a particular event, presenting several different forms which are equivalent in abstract logical content, but different in their behavior with respect to memory chunking and accessibility. Figure 10 gives the units which are referred to by these alternatives.

The unit named *Event234* represents a specific event which is being remembered by a KRL program. In all of the versions, the recipient is represented by a pointer to the unit for the individual *Person1*, corresponding to a situation in which the identity of that individual was the salient fact. Also in every version, the object is represented only by a description *(a Pen),* indicating that the specific identity of the pen is not remembered. Of course, these are arbitrary choices representing the way one particular program (or person) would remember the event. Someone else remembering the same event might store a description containing the identity of the pen (for example' if it were a special memento) and only a description of the person who received it.

The four versions differ in how much detail about the other person is considered salient to the specific event. In Version 1, the giver's identity is all that is included. In version 2, the identity of his wife is included as well, and in version 3, so is her occupation. Version 4 differs from the others in omitting the specific identities altogether. It corresponds to the kind of incomplete memory which might be expressed as "Let's see, David got the pen from some guy who was married to a lawyer."

```
[Event234 UNIT Individual                Version 1
      <SELF (a Give with
            object = (a Pen)
            giver = Person2
            recipient = Person1 )>]

[Event234 UNIT Individual                Version 2
      <SELF (a Give with
            object = (a Pen)
            giver = {Person2 (which IsHusbandOf Person3)}
            recipient = Person1 )>]

[Event 2 3 4 UNIT Individual             Version 3
      <SELF (a Give with
            object = (a Pen)
            giver = {Person2 (which IsHusbandOf {Person3 (a Lawyer)})}
            recipient = Person1 )>]

[Event234 UNIT Individual                Version 4
      <SELF (a Give with
            object = (a Pen)
            giver = (which IsHusbandOf (a Lawyer))
            recipient = Person1 )>]
```

Figure 9. Alternative memory structures showing different choices of redundancy

```
[Give UNIT Specialization
      <SELF (an Event)>
      <object (a Thing)>
      <giver (a Person)>
      <recipient (a Person>]

[Lawyer UNIT Specialization
      <SELF {(a Person))>]

[Pen UNIT Basic
      <SELF {(a PhysicalObject)>]

[Person 1 UNIT Individual
      <SELF (a Person with firstName = "David")>]

[ Person2 UNIT Individual
      <SELF {(a Person with firstName = "Jonathan")
            (which IsHusbandOf Person3)}>]

[Person3 UNIT Individual
      <SELF {(a Person with firstName = "Ellen")
            (a Lawyer))>]
```

Figure 10. Units used in the alternative formulations shown in Figure 9

If we try to match *Event234* with the pattern *(a Give with giver = (which Is HusbandOf (a Lawyer)))*, the amount of processing needed to determine the answer differs for the different versions. The matcher must look into the contents of the units for Person2 and Person3 in the cases where there is less redundant information. The results for the various versions could differ as well. Matching *(a Give with giver = Person2)* against version 4, all we could say is that it is potentially compatible, while the other versions all provide a definite "Yes". In a system with competing parallel processes, these differences can have a significant effect on the results of reasoning.

It should be apparent that the kind of knowledge structuring involved in chunking facts into units is very different from the structuring of facts into truth *contexts,* as provided by the AI languages such as CONNIVER and QLISP. In those systems, each context contains a set of objects and assertions whose connection derives from being present and true within a particular state representing a hypothetical world, or time. This grouping is orthogonal to KRL's object-oriented chunking based on grouping a set of facts (or properties) about a particular conceptual object. In KRL there are descriptors (the *contingency* form) that are applicable only in some worlds, and facts that are true in some worlds, but this mechanism is separate from the clustering of relevant facts into a memory structure.

2g. Indexing and Retrieval

One of the fundamental problems in the use of memory is the *retrieval* of appropriate knowledge from a large data base. As mentioned in section 2e, KRL makes a distinction between the process of retrieval and the process of matching. In finding a desired object in memory, there are two steps. The first is a rough retrieval step, designed to produce a small set of units which potentially fit the specification for what is being sought. This is followed by a more thorough matching process in which each candidate is matched against the retrieval pattern, using the mechanisms discussed above.

This separation between retrieval and matching is carried over in the form of the memory. In most existing AI systems (and models of human memory) there is an underlying assumption that there is a single set of data linkages, used both for retrieval and for matching or deduction. The data structures must contain all of the logical form of what is being stored, and be usable in some uniform way for memory search. Many researchers have explored the problems which arise in trying to create structures which have desirable properties for retrieval processes while also being an adequate representation of the logical structure; for example, see Anderson and Bower (1973). Woods (1975), and Hendrix (1975). Many human memory experiments (beginning with Collins and Quillian (1968)) have been based on the same assumption that a single set of links must handle both tasks. The uniformity of logical and retrieval structure is also basic in systems which use complete indexing (as in CONNIVER, QLISP, etc.), and those which retrieve through a complex search process (as in the derivatives of Quillian's original network representation (Quillian, 1968)).

We believe that the presence of *associative links* for retrieval is an additional dimension of memory structure which is not derivable from the logical structure of the knowledge being associated. KRL has a separate independent mechanism for creating associative links between arbitrary combinations of units, and for retrieving those units on the basis of the associations. These associations would be closely related to the rest of the knowledge structure, but in a way determined by specific memory strategies. One of the major topics for research is the development of strategies for deciding when to put in associative links and when to look for them in retrieval.

KRL-0 has a simple indexing mechanism which allows the user to catalog any unit under an list-structured pattern of *keys*. There is a primitive to retrieve all units matching a key combination and another for matching ail units indexed under any subset of a given key pattern. For most of what has been typically stored in AI language data bases, no indexing at all is needed in KRL-0, since the way in which descriptions are combined into units explicitly gathers together the information which would be retrieved by a data base mechanism. In the future, we hope to explore other retrieval mechanisms, perhaps involving spreading activation models, and parallel computation structures.

In addition to the internal structure of units, and the associations represented by the index, there is a third type of structuring in which a set of units are collected in a context or *focus list.** These focus lists are primitive building blocks for use in experimenting with notions of attention and differential access to data at different "levels of consciousness". There are primitives for adding anddeleting items, and for finding all (or the most recent) units in a given focus list matching a given description. A focus list provides one mechanism by which to implement and test models of short-term memory.

*The use of focus lists in solving anaphora problems in English dialogs has been explored by Deutsch (1974).

One aspect of memory structure which we plan to explore in KRL is the use of context-dependent description.* The results of human reasoning are *context dependent;* the structure of memory includes not only the long-term storage organization (what do I know?) but also a current context (what is in focus at the moment?). We believe that this is an important feature of human thought, not an inconvenient limitation. It allows great simplifications in the form of descriptions by allowing them to be context dependent. A descriptor which is going to be interpreted in a context with other descriptions and objects around can implicitly describe its connections to them, rather than needing to make all of the links explicit. The descriptor form *(theOne ...)* is used to specify a unit by giving a description which will pick it out uniquely in a context. This context might be a stored focus list, or one dynamically created as part of a current process.

*For an extended discussion of this idea. see Bobrow and Norman (1975).

3. Extended control structures

In designing KRL-0 we chose to concentrate our efforts on the declarative side of the language. The control mechanisms and procedure specification formalism makes use of -LISP as much as possible, extending what was already there, rather than building from the ground up. There is no such thing in KRL-0 as a "KRL-procedure". When a procedure is to be specified, it is done as a LISP function which makes use of a set of primitives (LISP functions) provided for manipulating the KRL-0 data structures. Arguments and values are passed in the normal LISP way, and subroutine calling obeys the usual stack discipline rules, as provided by INTERLISP. This includes the use of generators and other coroutines made possible by the spaghetti stack.

The' underlying control structure has been extended in several directions: object-oriented process specification *(procedural attachment);* a general *signal* mechanism for error handling, notification, and dynamic procedural parameterization; organization of the basic system functions around *process frameworks;* and a multi-process executive, based on a *multi-level scheduling agenda* with resource and priority management facilities.

3a. Procedural attachment

One of the major current directions in programming language research involves factoring procedural knowledge orthogonally to the traditional programming formalisms. Each primitive program step can be viewed as applying some *operation to* one or more data objects each of which belongs to some ***class.*** The traditional way of organizing programs is to have a procedure keyed to each operation. The internal structure of this procedure takes into account the alternatives for the data objects on which it will operate. Languages such as SMALLTALK (Learning Research Group, 1976) and SIMULA (Dahl and Nygaard, 1966) and the various ACTOR formalisms (Hewitt, Bishop, and Steiger, 1973, Hewitt and Smith, 1974) group together the different procedures to be carried out on objects of a single class. The programmer defines classes of objects, and associates with each class a procedure whose internal structure takes into account the different operations which will be done. The ***to-fill*** and ***when-filled*** triggers discussed in the context of frame representations (Winograd, 1975, and Bobrow et al., 1976) are further examples of this *procedural attachment.* AI languages such as QLISP and the PLANNER family represent a different method of factoring the procedures, according *to* ***configurations*** defined in terms of patterns which are to be matched against goals and assertions. These configurations represent potentially arbitrarily overlapping classes, which is quite different from the use of classes in SMALLTALK and SIMULA.

We have extended the notion of object-oriented procedure definition in two important directions. The first is an integration of the ideas of object-associated procedures with those of frame structure and multiple description. In associating procedures with a class represented by a unit in KRL, procedures can be linked to the slots of a unit, and specifically associated with several different descriptor types. This makes the clustering of the procedures correspond better to the conceptual structuring of the domain. In addition, KRL provides through its use of perspectives a notion of *subclass* which allows objects to inherit procedural as well as declarative properties. Since each unit can contain multiple perspectives it can be a member of a number of subclasses.

KRL-0 provides facilities for procedural attachment which can be divided along two dimensions. The first is based on when the procedure is intended to be used -- whether it is a *servant* which provides the method to carry out some operation, or a *demon* which causes a secondary effect of some event. The second dimension corresponds to whether the procedure is associated with an individual data element or with a class (prototype).

A servant is invoked when the system has the goal of applying some specific operation to a data object (or set of objects), and needs a procedure for accomplishing the specified task-. The interpreter looks for servant procedures associated with the data object or its class, and if it finds one, executes it to carry out the operation. If there is more than one, a strategy procedure is called (using the signal mechanism described below) to choose. A typical use of servants is to attach a procedure describing how to match a descriptor involving a particular relation or prototype. In Figure *8,* there is a ***ToEstablish*** servant, associated with the unit for ***Ownership,*** which uses the LISP function AND and the KRL function Match. It provides a specific procedure for determining ownership in a special case.

A demon is invoked as a side effect of actions taken by the system. All of the primitive data-manipulating operations check for demons, whenever they use or add information. A demon can be associated with a description of the operation to be done, and the object it is done to. A unique object can be specified, or a demon can be invoked for any object of a specified class. The antecedent theorems or if-added methods of the AI languages are examples of demon-like mechanisms, in which the invoking events are asserting and erasing, and the classes of data objects are specified by patterns. Demons can be awakened when something is about to be done or has just been done (there are both types). A demon typically might be invoked when a u-nit representing an individual is filled in as part of the description in an instance. This could trigger further processing which requires knowing the individual.

The second dimension of procedural attachment is the distinction between procedures associated with individual data objects *(traps)* and those associated with classes *(triggers).* Triggers are class- based -- they apply to operations and events which take place on objects whose description includes a perspective whose prototype is the unit to which the trigger is attached. Traps are instance-based -- they apply to operations and events directly involving the unit to which they are attached. A list of triggers and a list of traps can be associated with each slot within a unit. Both triggers and traps can be either servants or demons.

In addition to the servants explicitly sought by the system (in the frameworks for the matching and searching processes described below), and the demons explicitly triggered (by the data manipulating functions) the user can also provide arbitrary demons and servants and check for them explicitly. The user can independently define a set of trap and trigger names, and use them for organizing a computation. There is a primitive used to *probe* for attached procedures, and if multiple traps or triggers are applicable, they can set up multiple processes (see below).

3b. Multiprocessing and variable depth of processing

The overall control structure of KRL is based on our belief that the next generation of intelligent programs will be built using multi-processing with explicit (user-provided) scheduling heuristics and resource allocation. This view is based partly on looking at current multi-process oriented systems*, and partly on looking at properties of human processing, such as *partial output* based on *variable processing depth.*** We want to provide ways to build a system whose components can run for varying amounts of effort, producing some initial results with a small effort, and improving the quality of their results (either in amount of data, or certainty) as the effort is increased. We also want to explore issues of *attention* or focus in choosing which of competing procedures to run.*** By beginning with a multi-process system whose control structure is explicit and visible, we hope to have a base from which to experiment with a variety of control and resource allocation strategies.

---

*Such as in syntactic analysis (Kaplan, 1973b) and in speech understanding systems, as described in Reddy and Erman (1975).

**See Norman and Bobrow, (1975) for a discussion of human processes which are limited by resources.

***See Hayes-Roth and Lesser, (1976) and Paxton and Robinson, (1975).

We expect one of the major research areas to be the integration of data-directed and goal-directed processing. In the course of running, there will be an explicit goal structure for the program as a whole, while new processes will continually be triggered through procedural attachment and data from external sources. Resource manipulation and priorities will be used to provide a global direction to the processing yet keeping flexibility to deal with unanticipated combinations.

Many of the specific facilities in KRL, such as procedural attachment and the extended notion of matching discussed earlier, presuppose some sort of multiple process system, in which a set of procedures can be set up and control can be passed between them in a systematic way. The central executive of KRL is based on an *agenda** of runnable processes, and a scheduler for running them in a systematic order. The agenda is a priority ordered list of queues, with all processes on a higher priority queue run before any on lower priority queues.

*See Kaplan, (1975) for the use of an agenda in a system for parsing natural language.

All scheduling is done by cooperation, not preemption. A process runs until it explicitly returns control to the scheduler. It can add any number of other calls to the agenda before it gives up control, including a call to continue itself. Whenever the scheduler runs, it scans down a series of *priority levels* on the agenda, and runs the first process in the highest priority non-empty queue. It removes that process from the agenda and starts it (if it is new) or resumes it (if it has been suspended).

The agenda levels can be used to achieve a variety of standard control disciplines. As one example, new inputs can be checked for periodically, and can put actions on the agenda at a high priority level, which will then be done before ongoing processing at lower levels, and may even remove such processes from the agenda. This makes it possible to write procedures (e.g., story understanders) whose depth of processing varies with the rate of the inputs. As another example, a part of the computation can be made *relatively continuous** (Fisher, 1970) with respect to another by causing all of its processes to be scheduled at a higher priority level. A higher priority process is relatively continuous with respect to a lower priority process in that it is guaranteed to run to completion between any successive actions in the lower priority process. Use of relatively continuous processes is especially useful for coordinating processes with intermediate steps which leave data in states which would be inconsistent for use in other parts of the computation.

'Part of the information associated with each process includes a pointer to a *resource pool,* which can be shared between any number of processes. There is no automatic assignment or checking of resources, but any process can check and increment or decrement its resource pool, and invoke a user-provided procedure if resources have run out. The agenda itself is an accessible data structure, and the scheduler looks for specialized strategy procedures (using the signal mechanism described below) in all of the places where resource and scheduling decisions are made. The user can design combinations of strategy procedures (called a *control idiom)* suited to the particular program or component. This will be described further below in discussing process frameworks.

3c. Procedure directories, modules, and process frameworks

For dealing with complex descriptions and matching processes, it is critical that the user be able to build up different mechanisms and strategies at a high conceptual level which do not demand detailed concern with all the ways different descriptor types may appear. There need to be processes built into the system which do the necessary bookkeeping and basic alignment for carrying out matching, description building, and searching operations. At the same time, there is no one way that things should be done. The strategy for carrying out a particular matching task can be selected along many dimensions, and detailed decisions at each step depend on the particular design choices.

Our solution to the desire for both generality and automatic handling of detail is to provide *process frameworks* for all of the basic operations, and a *procedure directory* which provides a mapping from a set of names (designating the procedure to be done) to proced u res. For example, rather than having a semantically complete definition of what happens in a match, the system provides a matching framework which contains processes for setting up the structures to compare two descriptions, doing the alignment of comparable descriptors, looking for procedures attached to specific patterns, and handling all of those cases where a simple syntactic match will work.

Whenever there is a "hard case" of any sort (i.e., something which cannot be resolved by simple syntactic properties), the system looks in the procedure directory for an entry corresponding to the unique name associated with that case. The directory is dynamically maintained -- entries can be added and removed either singly or in groups called *directory modules. The user can specify with each call to the matcher a directory module which has entries to take the appropriate actions for all of the hard cases he wants to handle.* There is a set of initial default entries in the directory which take some action (the best that can be done without further information) in the absence of a user-supplied entry. Typically, a program will include a set of alternative modules, with one of them being "plugged in" to the directory for each call to the matcher.

In some sense, this can be thought of as defining the system's basic functions (such as Match) using calls to sub-procedures which are to be provided by the user. The additional directory mechanisms make it possible for the user to provide alternative definitions in a much more flexible way than with the static lexical procedure-naming conventions used by most programming languages. The use of directory modules makes it easy for the user to bind and unbind clusters of "functional arguments" in a single operation.

As an example within the matcher, the user could provide a module with an entry telling what to do when matching two perspectives whose prototypes represent abstract categories such as *(a PhysicalObject)* and *(an Animal).* One possible entry would indicate that the match should be abandoned without further work, while a different one might call on a complex procedure which uses a classification hierarchy. We expect to build up a vocabulary of standard modules (which we call *match idioms)* which represent different combinations of the features described above. One module (used for quick checking to see if there is an easy match) returns "don't know" if the built-in syntactic checks don't

give an immediate yes or no answer. A module for quick disconfirmation checks only for those kinds of descriptors which give definite negative evidence easily (e.g., conflicting individuals or conflicting basic categories). One for forced match would do a complete mapping process when faced with two conflicting individuals, returning the places where their descriptions differ. The standard modules can be augmented with specialized individual entries for specific situations. The user can also control the way in which new entries take precedence over pre-existing ones.

Briefly summarizing, a process framework provides a basic structure which sets up an environment and divides the process up into a set of cases (subtasks) to be handled. Each case is given a unique name (as part of defining the framework). For each such name, there are three three different places to look for a detailed procedure:

* in the built-in mechanisms of the framework (e.g., the simple syntactic cases of matching);
* in procedures attached to the data on which it is working (e.g., To-Match triggers associated with a pattern);
* in the current procedure directory (e.g., an entry stating that when two conflicting individuals are to be matched, a given mapping process should be called).

There are several places other than matching where process frameworks are used in KRL-0. The primitive process for adding a new description to an already existing unit is a framework which allows for event-triggered side effects of each of its actions. The scheduler described in the previous section is a framework which allows for modules specifying different control strategies. As an example, a simple program has been written to search an AND-OR tree, along with two different modules which cause the search to be breadth-first or depth-first respectively. Other modules could specify a heuristically guided search according to different strategies. We expect to build up a vocabulary of generally useful control idioms, including complex ones involving resource allocation.

The mechanism used in KRL-0 to implement procedure directories is based on a notion of signals*. The system maintains a *signal path* associated with each independent process, which is a linked list of *signal tables,* each of which is an association list pairing *signal names* with actions. There is a primitive which, given a signal name, looks on the signal path for the first place where that signal name is found, then executes the action associated with it. In addition, there are a range of primitives for putting actions into a signal table, pushing one onto or popping one off of the signal path, resetting the current signal path, etc.

---

*The concept of signals has been adapted from MESA (Lampson, Mitchell and Satterthwaite, 1974).

The signal mechanism is used for two other purposes in addition to providing procedure directories for process frameworks.

Errors: Whenever an error condition arises, a signal is generated whose name specifies the error condition. The default action is to stop and interact with the user on line, but the user can specify in the current signal path any action whatsoever to be taken to handle the error. This could include patching things up so the computation can go on, or could involve aborting the process in which it occurred, or any complex computation which might be built in as part of a debugging system.

Notification: Whenever any one of a specified set of system operations occurs (e.g., adding a new process to the agenda) a signal is generated. The default is to do nothing, but the user can specify any action, which typically would include printing out monitoring or debugging information, taking special actions, or keeping statistics. This makes it easier to provide debugging and monitoring tools for use in a multi-process environment, which by its essential nature makes it difficult to keep track of just what is going on when. These notification signals can also be used directly to trigger event-driven processes. For example, a data base indexing mechanism might operate by catching the signal generated by the primitive which augments a description with a new descriptor and taking the appropriate indexing actions.

Communication between the procedure assigned to the signal (according to the current signal path) and the context in which it is invoked is handled through the use of free variables.

4. Where we are headed

4a. Experimental implementation and recycling

Our approach in building KRL has been guided by a philosophy of working from actual domains and problems towards a habitable representation system, rather than starting with an abstractly designed representation and trying to force the world into it. To some degree we have drawn on our collective experience with designing language understanders. However, we believe that complex systems are "Whorfian" in that the underlying structure and style of a representation language can have a strong effect on what people attempt to do with it. Therefore, we see a need for a feedback loop in which systems are built, used, and then redesigned on the basis of experience.

Our current research strategy includes a cycle of three steps, with a step time on the order of 6 months to a year:

1) Design a system based on our current understanding and experience.

2) Build an experimental implementation which captures as much of this as feasible.

3) Use the system in a number of test domains to understand its capabilities and push its limits.

We are currently entering the third step on our first major round of design on KRL as embodied in KRL-0. For our experiments with KRL-0, we have chosen the strategy of

implementing a set of already existing AI programs, each of which we hope will exercise different subsets of its facilities, and raise additional representation issues. Our current plan is for the Xerox understander group and several Stanford students to work on 5 to 10 programs*. Each of these benchmark programs makes use of well-understood AI -techniques, which push the current facilities of AI languages and systems, but which are clearly formulated in the already existing programs (or extended program descriptions which take the place of programs in several of the M.I.T. dissertations). In some cases (such as MYCIN) much of the actual program deals with issues of smooth user interface, and the accumulation of large bodies of knowledge. We will not try to imitate these aspects, but only try to duplicate the basic modes of operation and reasoning. For most of the systems, however, it seems quite feasible to duplicate the complete published performance.

---

*Candidates being considered (some in progress) are: a simple cryptarithmetic problem solver (See Newell and Simon, 1972 for a description of the task); SAM, the Yale story understander (Schank et al., (1975). and Lehnert, 1975): a learning program for recognizing a simple kind of ARCH (Winston, 1975); the Blocks world planning programs HACKER (Sussman. 1973) and NOAH (Sacerdoti, 1975); the Rutgers action understander. BELIEVER (Schmidt, 1975); MYCIN (Shortliffe, 1976), a simple medical advice system; a more complex medical reasoner, perhaps CASNET (Kulikowski, 1974) or a diagnosis program sketched at MIT (Rubin, 1975); a legal reasoner based on a recent MIT dissertation (Meldman, 1975); GSP, a general syntactic processor (Kaplan, 1973a): and travel assistant programs which are part of the series of GUS programs at Xerox PARC (Bobrow et al., 1976). We will not do all of the programs on this list, but include them as examples of the kinds of programs we would like to do.

By beginning with systems whose behavior and general outline are well defined, we can concentrate on the ways in which the KRL representation can be used to full advantage, and the places in which it does not meet the needs. We expect each of these systems to take on the order of l-2 person-months of work, and to provide part of the feedback cycle for the design of KRL.

4b. Goals for future versions of KRL

Although we cannot predict all the changes which our experience will force on us, we are aware of several major issues which we consciously avoided in designing KRL-0. Major areas of expansion in our future designs will include: a Lisp-independent specification of the. primitive data objects; a descriptive formalism for specifying procedures; integration of the different procedure-calling facilities and indexes; development of an integrated system for programming and debugging; and development of a more convenient syntax.

Procedure specification: We need a way to specify procedures other than by giving a LISP function or expression. We believe that a representation system should make it possible to describe processes with the same generality and flexibility as any other objects. We want to take the ideas of multiple perspective, process frameworks, signals, and multi-processing and integrate them directly into the ways procedures and arguments are specified, data is passed back and forth, etc. In particular we will be developing a notion of *factored description* in which a procedure is defined through a description based on multiple perspectives. This description may combine high level statements about the

structure of the process, its results, conditions on various parts, etc. along with detailed statements about the individual steps. The system should be able to look at and understand descriptions of its procedures as well as run them.

In most current formalisms there are completely different representations for the declarative statements (networks, or assertions, or clause sets) and the procedures. In those systems where there is a uniform base*, the declarative form is used primarily as a notation for writing programs as a sequence of steps to be executed. We want to greatly expand the conceptual tools for describing and talking about procedures from multiple perspectives.** We believe that this is necessary for two complementary reasons.

---

*As in MEMOD (Norman, Rumelhart and LNR Group, 1975). or for that matter, simple LISP structures.

**Systems like HACKER (Sussman, 1973) and MYCROFT (Goldstein, 1974) are first steps in this direction.

First, the ability to describe and reason about procedures is useful for making programs easier to write, and necessary for the kinds of self-conscious strategy choosing, debugging, explanation. and self-modification which are increasingly becoming a part of complex computer systems. One of the major beauties of LISP is the fact that programs are themselves built from the language's data structures (atoms and lists), making it easy to write editors, debuggers, program analyzers and programming assistants. We would like to apply this kind of self-analytic power at a higher level, using the reasoning, matching, and problem solving powers of KRL as a fundamental element in our tools for designing, building and working with KRL programs.

Second, as we explore the kinds of asynchronous, factored multi-process styles of program organization which are coming into existence, we will move away from the notion of a program as a sequence of steps (or simple control structures), and will explore alternative views of a process description in the language itself. In addition to being able to describe procedure definitions statically, we also need ways of describing and manipulating descriptions of dynamic states of processes. Even in systems such as LISP in which programs can be represented in the data structures of the language, the state of a process is represented in a totally separate set of data structures (stacks, registers, etc.). One advantage of production systems* is that all control information is explicitly represented in the data structures. We hope to retain this property of uniformity and visibility, while providing a more structured set of mechanisms for building and manipulating control structures. These include primitives for building a priority ordered agenda of things to be done, assigning descriptions (in the declarative forms of the language) to processes on the agenda or currently being run, and assigning and consuming shared resource measures.

*See Davis and King (1975) for a discussion of control structures used in production systems.

This extension of the way in which programs can be written is the largest part of what needs to be done. We hope it will be one of the major advances achieved by KRL as a programming language. It will involve a good deal of further research into how programs for a multi-process environment are naturally conceptualized, and how people can best

use the power of signals, procedural attachment, and other mechanisms which extend normal definition and control structures. We also need to find ways of implementing interpreters and compilers which operate from complex descriptions of a desired procedure rather than a step-by-step program.

Data objects: We need to formally specify the semantics of those data objects we are currently borrowing from LISP (atoms, strings, numbers, lists, arrays) and provide the necessary primitives. We will add some new data types more oriented towards a multi-processing approach, such as *streams, partially specified lists* and sets. There are currently some mechanisms for working with sets within KRL-0, but they need to be integrated much better with the procedures and the primitive use of lists. Along with the primitive data objects, there will be a corresponding set of descriptions (in the KRL formalism) for use in programs which explicitly manipulate data and do reasoning about its form.

A more uniform approach to indexing: KRL-0 contains a number of different mechanisms which can be viewed as carrying out a common task of using some kind of indexing mechanism to associate data objects (units or procedures) with names. Signal tables, the attachment of procedures to slots of units, and the associative index are very similar in structure. Other mechanisms (such as the retrieval of descriptions from fillerpairs in perspectives, and the use of variable names in a program context) can be put into the same mold. In future versions, we hope to provide a better-structured, more uniform mechanism for all of these, in order to reduce the diversity in the current system.

Integrated System: All of the interfaces between programs and the world (user interactions, file systems, etc.) are currently done using LISP and will have to be defined independently. This includes the obvious sorts of input-output, and also the user-interaction facilities for writing, filing, editing, compiling, running, and debugging programs. We believe that the expanded reasoning powers of KRL programs will make it possible to write systems which are more flexible and useful than those existing in LISP. One of our research goals will be to develop intelligent programming apprentices* within an integrated KRL system. In the area of input-output, we want to deal explicitly with different types of output device (random access, stream, formatted page) and input device (streams, pointing devices, asynchronous event devices (such as keysets)) in a style which makes it easy to apply the general tools of KRL to programs demanding sophisticated user interaction.

● !&e Hewitt and Smith (1975). Winograd (1975a) for some ideas in this direction.

Syntax: The current syntax for KRL-0 is quite clumsy, since it was designed to operate in a LISP based system with a minimum of intermediate parsing. Except for the use of multiple bracket types, it is essentially LISP syntax. This results in an inordinately large number of bracketing characters in complex descriptions, and sequences such as "))}}))}>]" are not uncommon. We need to work out a more natural syntax. This will become even more important when we design the forms for describing programs and integrate them with the existing description forms.

4c. Building a layered system

Throughout this paper we have described ways in which KRL provides a flexible set of underlying tools, rather than embodying specific commitments about processing strategies, or the representation of specific areas of knowledge. In terms of Figure 1, all we have described is the bottom layer. One of our major goals will be to build a set of strategy and knowledge modules on top of it. This will be done in the context of designing one or more specific systems for language understanding in a limited domain, with an emphasis on clean, well defined interfaces.

The construction of an integrated system will demand building many components. As we construct each one, we want to do it in a style which does not limit its usefulness to the specific context for which it was written. There is no one solution to the problems at a given level which will be satisfactory for all systems. But we believe that it is possible to develop a set of alternative modules at each level which are sufficiently broad and flexible that someone interested in working at the next higher level could choose between them, rather than building all the way down.

Over the course of several years and the design of several different systems, we hope to develop a large inventory of modules, each containing a substantial body of knowledge, and all expressed in a compatible formalism. If we are successful at finding the appropriate lines along which to decompose the knowledge which goes into language understanding (and thought processes in general),' it will be possible to construct from them programs of much greater size and complexity than those now feasible.

4d. Summary

We are in the process of developing a knowledge representation language which will integrate procedural knowledge with a richly structured declarative representation designed to combine logical adequacy with a concern for issues of memory structure and recognition-based reasoning processes. The representation provides for several independent dimensions of structuring which deal with the logical content, the relative accessibility of different pieces of knowledge, and the association of specialized processes with data at various levels of specificity.

The system provides a basic orientation towards a recognition process based on a procedural framework for matching. The control structure is based on multi-processing with explicit (user-provided) scheduling and resource control. Process frameworks and procedure directories are used to give the user detailed control over the semantics of the fundamental system operations. These include: adding new descriptions to memory; searching for a memory unit matching a given description; matching a given pattern against a specific description; and scheduling processes, based on resource allocation.

The system is complex, and will continue to get more so in the near future. We are intentionally trying to be eclectic rather than reductive, in order to maximize what we can learn from our experiments with early implementations. As continuing experience indicates to us which of the facilities are most important, and points out ways in which they can be simplified, we will refine the language. However, we do not expect that it will ever be reduced to a very small set of mechanisms. Human thought, we believe, is the product of the interaction of a fairly large set of interdependent processes. Any representation language which is to be used in modelling thought or achieving "intelligent" performance will have to have an extensive and varied repertoire of mechanisms.

References

Bobrow, D. G., Kaplan, R. M., Kay, M., Norman, D. A., Thompson, H., and Winograd, T. *GUS, a frame driven dialog system* Xerox Palo Alto Research Center, 1976.

Bobrow, D. G. and Norman, D. A. Some principles of memory schemata. In D. G. Bobrow and A. M. Collins (Eds.), *Representation and understanding. New* York: Academic Press, 1975, 131-150.

Bobrow, D. G., and Wegbreit, E. B. A model and stack implementation of multiple environments. *CACM* 16:10, 1973, 591-603.

Bruce, B. Case systems for natural language. *Artificial Intelligence,* 6:4, Winter 1975.

Carbonell, J. R., and Collins, A. M. Natural semantics in artificial intelligence. *American Journal of Computational Linguistics,* 1974, 1, Mfc. 3.

Dahl, O.J., and Nygaard, K. SIMULA--an ALGOL-Based Simulation Language. *CACM,* 9:9 1966, 671-678.

Davis, R., and King, J. An overview of production systems. Memo AM-271, Stanford University Artificial Intelligence Laboratory, October, 1975.

Deutsch, B. The structure of task-oriented dialogs. Proceedings of the IEEE symposium on speech recognition, Carnegie-Mellon University, April 1974.

Fahlman, S. A system for representing and using real-world knowledge. MIT Al-Memo-331, 1975.

Fillmore, C. Against a checklist theory of meaning. *Proceedings of the First Annual Meeting of the Berkeley Linguistics Society,* Institute of Human Learning, Berkeley, 1975.

Fisher, D.A. Control Structures for Programming Languages. Department of Computer Science, Carnegie University, May 1970.

Goldstein, I. P. Understanding simple picture programs. Al-TR-294, MIT Al Laboratory, September, 1974.

Hayes-Roth, F. and Lesser, V. Focus of attention in a distributed-logic speech understanding system. Department of Computer Science, Carnegie-Mellon University, January 1976.

Hendrix, G. G. Expanding the utility of semantic networks through partitioning. *Advance papers of the fourth international joint conference on artificial intelligence,* Tbilisi: 1975, 115-121.

Hewitt, C., Bishop, P., and Steiger, R. A universal modular ACTOR formalism for artificial intelligence. *Proceedings of the Third International Joint Conference on Artificial Intelligence, 1973,* 235-245.

Hewitt, C., and Smith, B. Towards a programming apprentice. *IEEE Transactions on Software Engineering,* SE-I, March 1975, 26-45.

Kaplan, R. M. A general syntactic processor. In R. Rustin (Ed.), *Natural language processing. New York:* Algorithmic Press, 1973 (a).

Kaplan, R. M. A multi-processing approach to natural language. *Proceedings of the 1973 National Computer Conference. Montvale,* **N.J.: AFIPS Press, 1973, 435-440 (b).**

Kaplan, R. M. On process models for sentence analysis. In Norman, D. A., Rumelhart, D. E. and the LNR Research Group, *Explorations in cognition,* **San Francisco: Freeman, 1975**

Katz, J.J., and Fodor J.A. The Structure of a Semantic Theory. In J. Fodor and J. Katz, (Eds.) *The Structure of Language,* **Englewood Cliffs, N.J.: Prentice Hall, 1964.**

Kuipers, B. A frame for frames: Representing knowledge for recognition. In D.G. Bobrow and A.M. Collins (Eds.), *Representation and understanding. New* **York: Academic Press, 1975, 151-184.**

Kulikowsky, C.A. A system for computer-based medical consultation. *Proceedings of National Computer Conference,* **1974.**

Lampson, B., Mitchell, J., and Satterthwaite, E. On the transfer of control between con texts. B. Robnet (Ed.) *Programming symposium, Paris* **1974. Heidelberg: Springer-Verlag 1974.**

Learning Research Group. *Personal Dynamic Media.* **Xerox Palo Alto Research Center, SSL76-1, 1976.**

Lehnert, W. Question answering in a story understanding system. Yale University Computer Science Research Report #57, 1975.

Meldman, J.A. A preliminary study in computer-aided legal analysis. MIT project MAC TR 157, 1975.

Minsky, M. A framework for representing knowledge. In P. Winston (Ed.), *The psychology of computer vision. New* **York: McGraw-Hill, 1975.**

Moore J, and Newell. A. How can MERLIN understand?. In Gregg (Ed.), *Knowledge and Cognition,* **Baltimore, Md.: Lawrence Erlbaum Associates, 1973.**

Nash-Webber, B. The role of semantics. In automatic speech understanding, in D.G. Bobrow and A.M. Collins, *Representation and understanding. New* **York: Academic Press, 1975, 351-383.**

Newell, A., and Simon, H. A. *Human problem solving.* **Englewood Cliffs, N.J.: Prentice Hail, 1972.**

Norman, D. A. and Bobrow, D. G. On data-limited and resource-limited processes. *Cognitive Psychology,* **1975, 7, 44-64.**

Norman, D. A., Rumelhart, D. E. and the LNR Research Group. *Explorations in cognition.* **San Francisco: Freeman, 1975.**

Paxton, W., and Robinson, A. System integration and control in a speech understanding system. AI Center Technical Note 111, Menlo Park: SRI, 1975.

Quillian, M. R. Semantic memory. in M. Minsky (Ed.), *Semantic information processing.* **Cam bridge: Massachusetts Institute of Technology Press, 1968.**

Reddy, D. R., and Erman, L. Tutorial on system organization for speech understanding. In D. R. Reddy (Ed.) *Speech recognition.* **New York: Academic Press, 1975**

Rosch, E. Cognitive representations of semantic categories. *Journal of Experimental Psychology: General.* **1975, 104, 192-233.**

Rosch, E., and Mervis, C. Family Resemblances: Studies in the internal structure of categories. *Cognitive Psychology,* **1975, 7, 573-605.**

Rubin, A. D. Hypothesis formation and evaluation in medical diagnosis (MIT-AI Technical Report 316). Cambridge: Massachusetts Institute of Technology, 1975.

Sacerdoti, E. The non-linear nature of plans. In *Advance Papers of the fourth international conference on artificial intelligence,* **Tbilisi: 1975, 206 - 214.**

Schank, R. *C.* **(Ed.),** *Conceptual information processing.* **Amsterdam: North-Holland, 1975(a).**

Schank, R. C. The structure of episodes in memory. In D.G. Bobrow and A. Collins (Eds.), *Representation and understanding.* **New York: Academic Press, 1975(b).**

Schank, R. C. and the Yale AI Project. SAM -- A story understander. Yale University Computer Science Research Report #43, August, 1975.

Schmidt, C. Understanding human actions. In B.L. Nash-Webber and R.C. Schank (Eds.), *Theoretical issues in natural language processing:* **Cambridge Mass., 1975.**

Shortliffe, E. *MYCIN: Computer-Based Medical Consultations.* **New York: American Elsevier, 1976.**

Srinivasin, C. The architecture of coherent information systems. *IEEE Transactions on Computers.* **April 1976**

Sussman, G. J. *A computational model of skill acquisition.* **Amsterdam: North Holland, 1975.**

Sussman, G., and McDermott, D. From PLANNER to CONNIVER - A genetic approach. *Fall Joint Computer Conference.* **Montvale, N. J.: AFIPS Press, 1972.**

Sussman, G. Winograd, T., and Charniak, E. Micro-planner reference manual (AIM-203). Cambridge, Mass.: Massachusetts Institute of Technology, 1970.

Teitelman, W. *INTERLISP reference manual.* **Xerox Palo Alto Research Center, December, 1975.**

Winograd, T. Frames and the declarative-procedural controversy. In D.G. Bobrow and A.. Collins (Eds.), *Representation and understanding.* **New York: Academic Press, 1975.**

Winograd, T. Five lectures on artificial intelligence (Stanford AI-Memo-246). Stanford, California: Stanford University, September 1974.

Winograd, T. Breaking the complexity barrier (again). *ACM SIGPLAN Notices,* **10:1, January 1975(a), 13-30.**

Winston, P. Learning structural descriptions from examples. In P. Winston (Ed.), *The psychology of computer vision.* **New York: McGraw-Hill, 1975.**

Woods, W.A. What's in a link?. In D.G. Bobrow and A. Collins (Eds.), *Representation and understanding.* **New York: Academic Press, 1975,.**