1986

Referential semantics for a register vector grammar natural language processing system /

Brian J. Murphy
Lehigh University

Referential Semantics For A Register Vector Grammar Natural Language Processing System

by

Bryan J. Murphy

A Thesis

Presented to the Graduate Committee

of Lehigh University

in Candidacy for the Degree of

Master of Science

in

Computer Science

Lehigh University

1986

This thesis is accepted and approved in partial fulfillment of the requirements for the Degree of Master of Science.

____
(date)

Glenn David Blank
Professor in Charge

[illegible]
CS Division Chairman

Lawrence J. Varnerin, Jr.
CSEE Department Chairman

Acknowledgments

I would like to thank the following people for their help in the work leading to and comprising this paper: Prof. Glenn Blank for explication of RVG and much editorial criticism, Nancy Solderitsch for much dialog on semantic theory, John Stevens for the extensive research and design effort in his own thesis - which my work proceeds from, Pat Salley for a lot of help with Scribe, Keith Werkman for useful shells and information regarding Emacs commands, all the people at work who have made *that* less of a hinderance these last two months, and my wife Cynthia for constant support on the home front.

Table of Contents

Abstract

The referential component of a language processing system trys to establish which members of a database of known entities are intended by the noun phrases and clauses of parsed statements. This thesis outlines procedural definitions for unquantified reference as represented by proper nouns, and quantified reference as indicated by the quantifiers "every" and "no" and the determiners "a" and "the." The significant structures and processes of a natural language system - a Register Vector Grammar (RVG) production system - are described in terms of their impact on reference. Referential processing is ordered by a *Scope List.* Actions that construct and order the Scope List follow the flow of the RVG parsing process, resulting in a solitary, preferred interpretation for most sentences. The hypotheses of Colmeraurer [4] and Stevens [10] for establishing the relative scope of quantifiers are examined and improved. The referential component identifies the Scope List objects and predicates with database items, and handles failed references by polling the user and updating the knowledge database.

Chapter 1
The Issues of Referential Semantics

1.1 The Referential Component

The Referential Component of a natural language processing system has the task of identifying the lexical objects and relations derived from input text with objects and events in the world. For our purposes "the world" will be understood to consist of a database of objects and predicates that the system knows about and that it can use for evaluating and responding to natural language statements.

The sentence "John loves Mary" involves the interaction of three semantic entities: John, Mary, and the predicate "love." The latter is constrained to take John as its subject and Mary as its object; or, using conventional predicate calculus notation: **loves(John,Mary)**. If the system is able to identify the structures representing John and Mary with objects Obj1 and Obj2 of the database, and finds a predicate P1(Obj1,Obj2) in the database that entails loves(Obj1,Obj2), then the system can verify the original statement as being true. A failure to match any of the three semantic constructs with database entities would signify that the statement was not true in the current state of the knowledge database.

I will describe the matching process used for *instantiation* - identifying a lexical construct with a specific database item - in the next chapter when I discuss reference mechanisms in RVG. That will be followed by an explanation of the RVG parse process and its role in building semantic entities to be instantiated. The fourth chapter will examine strategies for ordering the instantiation of such entities so as to derive valid interpretations, and the fifth will detail the instantiation process itself. The sixth chapter recaps the salient features of RVG referential semantics, and discusses requirements for expanding its coverage. First, however, I'll introduce some issues and complications inherent to referential semantics with which the current implementation deals.

1.2 Reference for Proper Nouns

In our example "John loves Mary," both the subject and direct object are proper nouns. For these to be matched with database objects correctly it is necessary that there be only one possible match for each, else the interpretation may be invalid. The referential component will not only need to find a database object which represents John; it will also have to determine that there is no other match to be found. Humans, of course, can determine from context which of several entities is intended. The current system can be expanded to do likewise via some mechanism for *focusing* reference on a subset of the database [5]. At present though, the system's domain of discourse is always the entire knowledge database, which is sufficient for demonstrating our approach to reference handling.

1.3 Quantification

The matching of lexical with database entities is constrained by any determiners and quantifiers that modify the lexical object. The current system recognizes the universal quantifier "every" and the negative quantifier "no" along with the determiners "a" and "the" as quantifiers of the objects they precede in text. Reference for each is handled differently from the others and from that used for proper nouns. I will illustrate the special treatment each requires by analysing the process for suitable variations on the simple sentence "a robot fell."

1.3.1 A: a robot fell.

Evaluating "a robot fell" involves:

```
1) matching "robot" with an object in the database
2) matching "fell(robot)" with a predicate in the
   database
```

In this case it does not matter if there are multiple matches for "robot" in the database: the sentence is true so long as the system knows of at least one object matching "robot" such that some database predicate verifies that the object fell.

1.3.2 THE: the robot fell.

"the" quantification imposes the same "one and only one" restriction that constrains reference for a proper noun. "The robot fell" is true only if there is exactly one robot in the domain of discourse. Verifying this sentence therefore requires:

```
1) matching "robot" with an object in the database
2) matching "fell(robot)" with a predicate in the
   database
3) verifying that there is no other match for "robot"
   in the database
```

1.3.3 EVERY: every robot fell.

The evaluation of "every robot fell" is handled like "a robot fell," but it is carried out for *all* database objects matching the universally quantified object "robot." If processing continues until no more database matches can be found for "robot" then the statement is true. Failure to instantiate "fell(robot)" for any instance of robot signifies the existence of a counterexample to the sentence, namely a robot that did not fall, and the statement is concluded to be false.

1.3.4 NO: no robot fell.

We have chosen to interpret "no" as though it were a combination of the universal quantifier with a negative particle. The negative particle reverses the truth value of the predicate occurring on the same clause level. Thus the sentence "no robot fell" and its equivalent, "every robot *did-not*-fall," are true only if the predicate "fell(robot)" is *not* true. In Chapter 3 we will address the significance of negation for evaluation of a predicate. To test a negative quantification, we will search the database for a *counterexample*; e.g.: a "robot" such that "fell(robot)" is true.

1.4 Scope: Priority Among Quantifiers

When more than one quantifier occurs in a phrase there is a potential for ambiguity in the interpretation, depending on the relative *scope* accorded the quantifiers. For example, the noun phrase (NP) "a man on the horse" must be interpreted with respect to two quantifiers, the "a" quantifying the head-noun "man" and the "the" which modifies the object of the postmodifying phrase "horse." By giving precedence to one or the other quantifier, we can arrive at two interpretations:

```
(1) any man on a horse by himself.
(2) any man on a specific horse.
```

People generally adopt a single interpretation, preferring one over alternatives: thus (2) seems a more natural reading for "a man on the horse" than does (1). The scope of "a" is in this case *dominated* by that of "the."

We represent the preferred ordering of quantifier scope by means of a

Scope List, with elements to the left dominating those to the right. We will adopt the notational convention that *quantifier*(**noun**) indicates that the semantic object **noun** has the referential constraints indicated by *quantifier*, which may be "a," "the," "every" or "no." By treating postmodifiers such as "on" as predicates relating the subject of the modifying phrase to the object, we can express (1) and (2) with the following Scope Lists:

```
(1s) a(man)the(horse)on(man,horse)
(2s) the(horse)a(man)on(man,horse)
```

Resolving the quantifications as ordered in (2s) will yield the preferred interpretation for "a man on the horse," whereas (1s) coincides with something like: "the horse that some man is on."

The previous example dealt with the simple case of a head-noun with a single postmodifying phrase. The objects of postmodifying phrases can have modifying phrases of their own, so the referential component must also be capable of handling constructions of the type: "*every* woman in *a* boat on *the* lake under *every* ...." Instead of $\gamma!$ readings, where γ is the number of quantifiers, we hope to get just one preferred reading, avoiding ambiguity as much as possible.

1.5 Parse-Time Referencing

An issue of referential semantics which the current implementation does not (yet) realize is the resolution of definite references (proper nouns and quantification indicated by "the") concurrent with parsing.

Given that the NP "the robot" with no postmodifying phrase is discovered during the parse, there is no reason that the reference should not be resolved during or prior to subsequent parsing. In fact there are at least two reasons that recommend parse-time instantiation:

1) **Psychological reality. Humans appear to resolve definite references well before the end of utterances.**
2) **Computational efficiency of disambiguation. Paths for subsequent parsing might be constrained by referential context.**

On the other hand, indefinite references (e.g.: "a robot") should await the end of the parse of an utterance. It seems more likely that, upon hearing the phrase "a car...," people wait for further input rather than immediately attempting to identify referents. Thus psychological reality seems to counterindicate the parse-time resolution of indefinite references. Moreover, the multiplicity of possible database matches can only be reduced by strictures entailed by later predicates. For example, there may be only one block such that "a block is on the table" is true, but the qualifying "on" relation, unknown while we are parsing the subject NP, may well be crucial for this identification. Thus computational efficiency also argues against parse-time resolution of indefinite reference.

The processing of "every robot..." or "no robot..." could begin during the parse, but would then consist of assembling a list of matches for "robot" which would have to be maintained until the parse constructed some structure to utilize or test them. This does not seem to be a psychologically realistic approach, and also lacks the tractability gains promised by parse-time handling of definite reference.

The current system was designed to handle definite reference during the parse, but does not actually perform that way at this time. All references are currently resolved after the parse has terminated.

1.6 Updating the Database

A final issue worth consideration in constructing the referential component is the need to update the knowledge database with new or altered facts. A sentence might not be accepted, either for incorporating objects and predicates unknown to the system, or for contradicting previously known data. The referential component will query the user when it encounters an unknown reference. The user may agree that the proposition should be rejected, or specify that it be made true by inserting the unknown objects and predicates into the database. The system will therefore need to keep track of the status of reference for semantic entities throughout the referential process.

Chapter 2
RVG Structures and Mechanisms

2.1 Introduction

To understand the referential process that has been implemented, it will help to first look at those aspects of an RVG system pertinent to referential semantics. I will cover only those portions that directly influence the referential component, but a full treatment of the overall system can be found in [2] and [3].

An RVG system undertakes to produce a single preferred parse of a natural language input, corresponding to typical human parsing performance. It uses syntax to order a set of actions which construct and refine the semantic structures that are eventually matched against or incorporated into the database. Semantic information is used to constrain the parse by restricting the set of legitimate continuations at each step of the process.

Actions ordered and invoked by the parse arrange the semantic objects and predicates for instantiation by constructing the Scope List. The Scope List is implemented as a linked list, the nodes of which associate a semantic entity with the appropriate mode of quantification. Predicate nodes are marked either 'pred' or 'pmod' to distinguish a main predicate and modifier-predicates from each other and from quantified objects. As mentioned in the previous chapter, nodes are put on the Scope List, front to back, in the order of their priority or dominance for the clause. Thus by following the order of the Scope List the referential component is able to, like the parse itself, seek a single preferred interpretation in an efficient manner.

Consider the sentence "the owner of every car has a license." The parser produces the Scope List:

```
every(car)the(owner)of(owner,car)a(license)has(owner,license)
```

Wherein the node for **of(owner,car)** will be marked 'pmod' while **has(owner,license)** will be marked 'pred'.

2.2 Ternary Vectors

RVG maintains syntactic and semantic data in **ternary vectors**: fixed length sequences of the values -, + and ?. For a specific vector type (e.g.: a syntax vector) each position in a vector represents a specific binary attribute (e.g.: "NOUN" or "LIVING"), and its value reveals whether that attribute is applicable, inapplicable or undefined/unknown with respect to the entity or state the vector represents. A value of + indicates that the feature applies, e.g.: +LIVING; a - that it does not, e.g.: -LIVING; and a ? means that the feature is unknown or irrelevant (?LIVING). To consider a very simple example, below are vectors distinguishing entities by means of five features.

```
                                        entity  vector
features: 1) SUBSTANTIAL                        <12345>
          2) MOBILE                     mouse:   +++-?
          3) LIVING                     Mary :   +++++
          4) HUMAN                      robot:   ++---
          5) FEMALE                     idea :   -----
```

Note that **mouse** is ?FEMALE: a mouse may be either female or not, but that fact is not significant to its identification as a mouse.

2.3 Match3 and Refine3

In constructing a semantic profile of an entity during the parse, the entity's current vector characterization is compared to any ternary vectors representing potential modifiers. If the two are consistent with each other, the entity's vector is *refined* by that of the modifier, becoming more tightly constrained in the process.

The simplest match between vectors, **Match3**, compares the values in all corresponding positions of two vectors, and succeeds so long as the value of + in one vector never corresponds to a - in the other. A - will match with - or ?, a + will match with + or ?, and a ? will match with anything.

```
    e.g.:       ???+++--- vs               ???+++--- vs
                ?+-?+-?+-                  ?+-??+?--
                    x x  fails                       succeeds
```

Match3 therefore tests for compatibility between two ternary vectors.

Refinement of a vector A by a vector B is effected by the ternary operation **Refine3** as follows:

```
    1) for every position of vector B such that the value is a -
       or +, the corresponding value of vector A is set to that
       of B.
    2) for all positions of vector B with a value of ?, the
       corresponding value of vector A is left unchanged.
```

```
    e.g.:        ???+++--- refined by
                 ?+-?+-?+- yields
                 ?+-++--+-
```

A Match3 generally precedes a Refine3, so as to ensure that successive refinements of parse structures maintain consistency with previously derived constraints.

2.4 The Entry

The basic unit for semantic structures in RVG is the **entry**. This structure is composed of a small set of dedicated slots that constrain and/or identify an entity and its connections to other structures in the text or database. The slots applicable to the current discussion include **LAB**el, the word or morpheme that identifies the lexical information represented by the entry; **INTR**insic, a ternary vector representing the semantic attributes that characterize and constrain the entry; an **INST**ance slot which prior to instantiation identifies the mode of quantification to be used, and afterwards indicates the matching database entity; a **REAS**on slot which holds status and failure codes for the reference search; and a fixed number of **ARG** slots which specify constraints for the arguments of predicate entries. The ARG slots most commonly in use are **ARG1** and **ARG2**. ARG1 constrains the direct object for transitive verbs and the subject for intransitives, while ARG2 constrains the subject for transitive verbs. The ARG slots will point to ternary semantic vectors or to entries.

Chapter 3
The Parse

3.1 A Simple Parse

The following example will illustrate the interactions of ternary vectors, entries, Match3 and Refine3 in the production of semantic structures during a RVG parse. For the purpose of illustration, we will use relatively short ternary vectors, and assume that they represent the following features:

```
        1) SINGULAR      2) ACTION       3) SUBSTANTIAL
        4) CONCAVE       5) ANIMATE      6) LIVING
```

For enhanced readability the vectors will be masked with the format:

```
                         ### ###
```

We will follow the parse of the sentence: "the robot kicked a block."

The Current Predicative State Register (CPSR) is a set of registers representing semantic roles such as PREDicate, SUBJect and direct OBJect, each of which can contain pointers to semantic entries fullfilling the associated roles. At the start of a parse the PRED slot of the CPSR is initialized with an amorphous entry whose vectors consist entirely of ?'s. As the first word, "the," introduces a noun phrase, the parser will create a semantic entry, and place a pointer to it in the SUBJ slot. STEP1 shows the structure built up in the CPSR so far:

```
STEP 1:                  Remaining text: the robot kicked a block.
  entry #1  (PRED)         entry #2  (SUBJ)
  LAB:                     LAB:
  INTR: ??? ???            INTR: ??? ???
  INST:                    INST:
  REAS:                    REAS:
  ARG1: ??? ???
  ARG2: ??? ???
```

The parser responds to the first lexeme "the" by updating the INST slot of the

NP entry currently under consideration, which in this case is that pointed at by the SUBJ slot. (The INST value will be used by the referential system, later.) Below, STEP2 shows the INST slot of entry #2 marked with 'the':

```
STEP 2:                      Remaining text: robot kicked a block.
  entry #1   (PRED)            entry #2   (SUBJ)
  LAB:                         LAB:
  INTR: ??? ???                INTR: ??? ???
  INST:                        INST: the
  REAS:                        REAS:
  ARG1: ??? ???
  ARG2: ??? ???
```

The system looks up the noun "robot" in its *lexicon*, finding a lexical entry with semantic constraints:

```
      lexical entry:          LAB:  robot
                              INTR: +-+ -+-
```

This structure is compared to the SUBJ entry by running Match3 against the INTR slots. Failure would indicate a failure in this branch of the parse. In this instance the Match3 trivially succeeds against the amorphous SUBJ vector, so the SUBJ slot is Refine3-ed by the lexical vector. Also, the LAB of the current NP entry is filled with the lexical entry's LAB value. After parsing the NP, the CPSR entries look like this:

```
STEP 3:                      Remaining text: kicked a block.
  entry #1   (PRED)            entry #2   (SUBJ)
  LAB:                         LAB:  robot
  INTR: ??? ???                INTR: +-+ -+-
  INST:                        INST: the
  REAS:                        REAS:
  ARG1: ??? ???
  ARG2: ??? ???
```

Next, the parser retrieves the lexical entry "look":

```
      lexical entry:          LAB:  kick
                              INTR: ?+- ---
                              ARG1: ?-+ ???
                              ARG2: ?-+ ?+?
```

The semantic constraints of the predicate must be matched against the (currently amorphous) entry pointed at by the CPSR's PRED slot. The lexical entry will look like this:

Since a SUBJ entry has been established, the ARG2 slot of the lexical entry is matched against the INTR slot of the SUBJ entry (entry #2) as well as the ARG2 of the PRED entry (entry #1). Upon success, the INTR vector of the SUBJ entry is Refine3-ed with the lexical entry's ARG2 vector, and the PRED entry is explicitly linked to the SUBJ entry via a pointer set in PRED's ARG2 slot. STEP 4a shows the resulting CPSR entries:

```
STEP 4a:                Remaining text: a block.
  entry #1  (PRED)       entry #2  (SUBJ)
  LAB:  kick             LAB:  robot
  INTR: ?+- ---          INTR: +-+ -+-
  INST:                  INST: the
  REAS:                  REAS:
  ARG1: ??? ???
  ARG2: entry #2
```

Furthermore, the ARG1 slot for "kicked" provides constraints for a direct object via a Match3 and Refine3 of the ARG1 vector of the entry PRED points at. The direct object is now required to be -ACTION and +SUBSTANTIAL. The parser creates a new entry (entry #3) to represent a direct object, and sets a pointer to it in the OBJ slot of the CPSR and in PRED's ARG1 slot. Having finished processing the main predicate, the CPSR entries look like this:

```
STEP 4b:                Remaining text: a block.
  entry #1  (PRED)       entry #2  (SUBJ)     entry #3  (OBJ)
  LAB:  kick             LAB:  robot          LAB:
  INTR: ?+- ---          INTR: +-+ -+-        INTR: ?-+ ???
  INST:                  INST: the            INST:
  REAS:                  REAS:                REAS:
  ARG1: entry #3
  ARG2: entry #2
```

The parser reacts to the next word "a" by updating the INST slot of the current NP, entry #3, as seen in STEP 5:

```
STEP 5:                    Remaining text: block.
   entry #1  (PRED)       entry #2  (SUBJ)       entry #3  (OBJ)
   LAB:  kick             LAB:  robot            LAB:
   INTR: ?+- ---          INTR: +-+ -+-          INTR: ?-+ ???
   INST:                  INST: the              INST: a
   REAS:                  REAS:                  REAS:
   ARG1: entry #3
   ARG2: entry #2
```

The parser then confirms that the lexical structure for "block" matches the constraints on the current NP (entry #3), and further refines the OBJ entry by setting the INTR vector to require +SINGULAR, -CONCAVE, -ANIMATE and -LIVING. Step 6 shows the final state of the CPSR entries.

```
STEP 6:                    Remaining text: .
   entry #1  (PRED)       entry #2  (SUBJ)       entry #3  (OBJ)
   LAB:  kick             LAB:  robot            LAB:  block
   INTR: ?+- ---          INTR: +-+ -+-          INTR: +-+ ---
   INST:                  INST: the              INST: a
   REAS:                  REAS:                  REAS:
   ARG1: entry #3
   ARG2: entry #2
```

The parse concludes successfully when it reaches the terminating punctuation with the semantic structures established.

It is at this point that referential processing begins.

3.2 Handling Postmodifiers

When RVG encounters a postmodifying phrase in the text, it parses it as a clause, taking the entry being modified as its subject. For example, in "the girl in the wagon waved," we treat "the girl in the wagon" by saying that the subject of the postmodifier "in the wagon" is its head noun, "the girl." The parse accomplishes this in the following manner.

When "in" is encountered, the following CPSR structures will have already been generated:

```
CLAUSE LEVEL 0                   Remaining text: in the wagon waved.
      entry #1   (PRED)    entry #2   (SUBJ)
      LAB:                 LAB:  girl
      INTR: ??? ???        INTR: +-+ -++
      INST:                INST: the
      REAS:                REAS:
      ARG1: ??? ???
      ARG2: ??? ???
```

The discovery of the preposition following a noun prompts the parser to fire the action POSTMOD. POSTMOD shifts attention to a new clause level, and to a new set of semantic (CPSR) registers which will represent semantic roles for the new clause. It also initializes a new entry for the PRED slot. Another action, ZEROSUBJ, then puts a pointer to the entry representing the head noun (entry #2) in the new SUBJ slot. The CPSR entries for the postmodifying clause will look like this:

```
CLAUSE LEVEL 1                   Remaining text: in the wagon waved.
      entry #3   (PRED)    entry #2   (SUBJ)
      LAB:                 LAB:  girl
      INTR: ??? ???        INTR: +-+ -++
      INST:                INST: the
      REAS:                REAS:
      ARG1: ??? ???
      ARG2: ??? ???
```

The preposition "in" is treated as the predicate of the postmodifier. The new PRED role is therefore refined with the lexical constraints for the predicate "in":

```
    lexical entry:              LAB:  in
                                INTR: --- ---
                                ARG1: ?-+ +??
                                ARG2: ?-+ ???
```

The SUBJ entry (#2) will undergo Match3 and Refine3 against the new PRED's ARG2 constraints. Upon success, PRED's ARG2 will point at the SUBJ entry. As a direct object is anticipated, an entry is created for it (entry #4) and Refine3-ed by the ARG1 vector from the predicate. The result is:

```
CLAUSE LEVEL 1                 Remaining text: the wagon waved.
        entry #3 (PRED)        entry #2 (SUBJ)      entry #4 (OBJ)
        LAB: in                LAB: girl            LAB:
        INTR: --- ---          INTR: +-+ -++        INTR: ?-+ +??
        INST:                  INST: the            INST:
        REAS:                  REAS:                REAS:
        ARG1: entry #4
        ARG2: entry #2
```

Normal parsing will continue using these structures until the word "waved" indicates to the parser that the postmodifying phrase has ended. Another action, MODCLOSE, will then return attention back to the main clause and its semantic registers.

3.3 The Locality of Negation

An operation performed during the parse which is worth special note is the propagation of negation onto a predicate entry. The normal interpretation of negatives by humans presumes that a negative particle affects the predicate occurring at the same clause level. Consider for instance the following sentence:

```
a boat not on a base topples.
```

Depending on whether negation acts on the main predicate or the *local* predicate, this statement yields one of the two following readings:

```
(1) some boat is on a base, and it does not topple.
(2) some boat is not on base, and it does topple.
```

Clearly, (2) yields the intuitive reading. Thus there is a *locality of negation* principle which referential processing must account for.

Thus, when the parser finds "no" before a noun, it sets a marker in the INST slots of the noun entry *and of the associated predicate entry*, to indicate "no"-quantification. The predicate's INST will eventually tell the referential process that a reversal of the truth value is necessary for that predicate. The handling of the negation at the correct level is thus assured.

Implementing the locality of negation yields another benefit in terms of processing efficiency. Consider the Scope List for our example:

```
a(boat) no(base) on(boat,base) topples(boat)
```

Now suppose that in the database we identify a "boat" and a "base" for which the predicate "on(boat,base)" is verified. The negation prescribed by the INST value of the "on(boat,base)" entry will prompt the referential process to immediately reject the current set of instantiations. In other words, if the boat is on a base, then it can't be the one intended in the sentence, and a new search should begin for another DB match for "boat." As the (failed) negation is local to the postmodifier predicate, processing with the bad instantiations terminates before any futile processing of the main predicate "topples(boat)" can begin. The main predicate will go unprocessed unless/until a set of instantiations are found that affirm that "there is some boat which is *not* on a base."

Chapter 4
Constructing the Scope List

4.1 Three Hypotheses for Ordering Quantification

John Stevens [10] developed procedures for adding nodes to the Scope List via parse driven actions, along with a family of recursive algorithms for driving the instantiation of node entries from the Scope List. His approach was adapted from three hypotheses on the governance of quantifier dominance suggested by Alain Colmeraurer [4]:

Hypothesis 1: The quantification introduced by the article of the subject of a verb dominates the quantification(s) introduced by the complement(s) closely related to that verb. In speaking of complements closely related to the verb, we exclude adverbial phrases, which will not be studied here.

Hypothesis 2: In a construction involving a noun and a complement of this noun, the quantification introduced by the article of the complement dominates the quantification introduced by the article of the noun.

Hypothesis 3: Whenever a verb, an adjective or a noun has two complements, the quantification is made in the inverse order of the natural order of their appearance; that is, the rightmost complement generates a quantification dominating the quantification generated by the other complement.

I will address the evaluation and implementation of these hypotheses one at a time.

4.1.1 Hypothesis 1

The first hypothesis appears to observe that the order of instantiation at the clause level is 1) subject 2) direct object and 3) predicate. Thus, the sentences:

```
1) A student ate every quiche.
2) Every student ate a quiche.
```

would produce the following Scope Lists:

```
1S) a(student) every(quiche) ate(student,quiche)
2S) every(student) a(quiche) ate(student,quiche)
```

Quantifiers on the same clause level are thus ordered by left-to-right syntax.

This rule is simple, intuitively agreeable, and elegantly suited to RVG parsing. Stevens managed Scope List insertion with the ordering prescribed by this hypothesis by means of two actions: NPCLOSE and CCLOSE [10]. **NPCLOSE** is fired whenever the parse recognizes the end of an unmodified NP, and appends a node to the Scope List for the NP entry. **CCLOSE** is triggered when the end of a main clause is reached, and inserts a Scope List node identifying the main predicate. The ordering of the Scope List proceeds directly with the parse flow:

```
text       : a               student        ate         ...
actions    : SUBJ:NP:DET(a) :N(student) :NPCLOSE:       ...
Scope List:                              a(student)     ...

text       :                every            quiche      ...
actions    : VTRANS(ate) :OBJ:NP:ADJ(every) :N(quiche):  ...
Scope List:                                              ...

text       : .
actions    : NPCLOSE:          CCLOSE(.)
Scope List:  every(quiche)      ate(student,quiche)
```

4.1.2 Hypothesis 2

This hypothesis asserts that the quantification on the object of a postmodifying phrase should be resolved prior to that on the NP being modified. This appears to be true for a large class of sentences:

```
e.g.: 1) every driver of every car has a license.
      2) a place in the country is nice.
      3) all the people in some town were irradiated.
```

The two interpretations found by varying the scope of the two occurrences of "every" in (1) are:

```
1a) every individual who drives all the cars has a license.
```

```
1b) for every car, each driver of that car has a license.
```

Clearly, 1b yields the more likely (or *preferred*) reading, and it is this one which utilizes Colmeraurer's second hypothesis.

There are, however, numerous examples of postmodifying phrases whose conventional interpretation contradicts Hypothesis 2. Stevens noted a pattern among the exceptions, most apparent in some where the precedence varied when "a" was replaced with the semantically equivalent "some," as in:

```
4) every lawyer in a city is rich.
5) every lawyer in some city is rich.
```

In (4) the common interpretation will concern *all* lawyers in *any* city, while (5) seems to indicate the lawyers of a particular place. The corresponding Scope Lists read:

```
4S) every(lawyer) a(city) in(lawyer,city) rich(lawyer)
5S) some(city) every(lawyer) in(lawyer,city) rich(lawyer).
```

A sampling of similar constructions lead him to conclude that a universal quantification ("all," "every," "each," "no") will dominate over that of the indefinite article "a"/"an," regardless of their relative position within a phrase. He therefore suggested an extension of Hypothesis 2, as follows [10]:

Hypothesis 2': In a construction involving a noun and a complement of this noun, the quantification introduced by the article of the complement dominates the quantification introduced by the article of the noun, *unless* the article of the complement is "a" ("an") and the article of the noun introduces a universal quantification (the article is "every" "all", "each", "any", or "no"), in which case the quantification introduced by the article of the noun dominates.

Stevens opted to design actions for implementing the original form of the Hypothesis, which has the same advantage as Hypothesis 1 of being expressible by a straightforward embedding of simple actions within RVG's parsing process.

As was mentioned in Chapter 3, discovery of a preposition following a noun causes the parse to invoke POSTMOD rather than NPCLOSE.

POSTMOD defines semantic entries for a new clause-level, which is then parsed normally. The object of the modifying phrase will be disposed of by the parser, including its addition to the Scope List, before the parse of the dependent clause is completed and processing at the level of the parent clause can resume. Eventually, the parse responds to the end of the postmodifying phrase by invoking MODCLOSE. This action restores the main clause entries preempted by POSTMOD, so that processing of the main clause can resume. As MODCLOSE also marks the completion of a modified entry, it also adds to the Scope List a node identifying the quantification for the modified NP, and one node for the postmodifying predicate constructed during the POSTMOD subparse.

The result is that, since the parser must complete processing for the postmodifier object before the head noun, the former gets added to the Scope List first, thus assuring the ordering prescribed by Colmeraurer's Hypothesis 2.

An illustration of the progression of parse actions and the addition of Scope List nodes shows POSTMOD and MODCLOSE at work:

```
text       : the              owner     of                              ...
actions    : SUBJ:NP:DET(the):N(owner):POSTMOD:ZEROSUBJ:                ...
Scope List:                                                             ...

text       :          every         car                                 ...
actions    : PREP(of):NP:ADJ(every):N(car):NPCLOSE:                     ...
Scope List:                                  every(car)                 ...

text       :            drives            .
actions    :   MODCLOSE:VINTRANS(drive)CCLOSE(.)
Scope List:    the(owner)of(owner,car)     drive(owner)
```

4.1.3 Hypothesis 3

Hypothesis 3 addresses the relative priority of quantifiers in constructions of recursively postmodified noun phrases, such as "*some* people in *a* boat on *every* lake under *the*...". The hypothesis provides that the quantification on the "deepest" level object of the complex phrase should take precedence over those of higher level. Thus the Scope List for:

```
1) the Eskimo in every boat on the lake paddles.
```

is:

```
the(lake) every(boat) on(boat,lake) the(Eskimo) in(Eskimo,boat)
      paddle(Eskimo)
```

The actions POSTMOD and MODCLOSE designed to implement Hypothesis 2 will without modification handle the insertion of Scope List nodes for recursively postmodified expressions in keeping with the quantifier domination exhorted by Hypothesis 3. Parsing finishes with the deepest level object, puts it on the Scope List, recovers the next higher level clause, puts *its* objects on the Scope List, etc. In fact, Hypothesis 3 is a generalization of Hypothesis 2.

4.2 Replacing Hypotheses 2, 3 and 2'

In tests with instantiating semantic entries according to these three hypotheses, Hypothesis 1 worked flawlessly for the domain of canonical declarative sentences handled by the present implementation. Hypothesis 2, however, failed more often than not: it is not general enough to make even a good default rule for the subset of quantification we wish to represent. Hypothesis 3 failed even more regularly than Hypothesis 2, as it is in effect an extension of the other (flawed) hypothesis.

A typical example of the failure of Hypothesis 2 (different from that accounted for by Stevens) was given in the first chapter:

```
the man on a horse fell.
```

According to Colmeraurer's Hypothesis 2, we should derive the following Scope List:

```
a(horse) the(man) on(man,horse) fell(man)
```

This reading, however, causes the referential system to verify the statement even if there are *two* men on horseback, so long as at least one of them fell. This is contrary to the normal usage for "the man on a horse," which should produce a unique referent. Colmeraurer's hypothesis actually corresponds to our results for "**a** man on a horse."

The apparent subservience of quantification by "a" to "the" as noted above, and to universal quantifiers as noted by Stevens, motivates a thorough comparison of the relative dominance of pairs of quantifiers in simple postmodifier constructions of the type:

```
quantifier1  noun1  preposition  quantifier2  noun2
```

All possible pairs using our selected quantifiers are shown in the table below, along with which of the two dominated in a sampling of simple phrases.

In the table, the head-noun/modifier-noun quantifications of a postmodifying predicate are indicated with **th** representing "the", **ev** representing "every", etc. The **head**, **pmod**, **either** or * following the hyphen denote respectively that the dominating quantification is that of the head-noun, the postmodifier-noun, both orderings are equivalent, or the conjunction of quantifiers is grammatically incorrect for English. It is noteworthy that, in every sample for each quantifier, the dominant member appears to be invariant.

QUANTIFIER DOMINANCE
LOCAL TO A POSTMODIFIED NP

a/a :	either	th/a :	head	ev/a :	head	no/a :	head
a/th:	pmod	th/th:	either	ev/th:	either	no/th:	either
a/ev:	pmod	th/ev:	pmod	ev/ev:	pmod	no/ev:	head
a/no:	head	th/no:	head	ev/no:	head	no/no:	*

For example, consider **th/no: head**, at the bottom of the second column. "th/no" is a shorthand for phrases with the format:

the *noun preposition* no *noun*

"head" indicates that for this case (th/no), the head noun dominates. Consider, for example:

```
the girl under no canopy got a blister.
```

My scope reading (in Scope List form) is:

```
the(girl) no(canopy) under(girl,canopy) a(blister)
        got(girl,blister)
```

This might be rephrased as: "there was only one girl who was not under any of the canopies, and that girl got a blister." Other patterns in the above table show scope preferences similarly:

1) **no** ingredient in **every** recipe is expensive.
2) **every** lawyer in **a** town eats meatloaf.
3) **the** person at **the** wheel makes the decision.

Colmeraurer's Hypothesis 2 thus fails for seven of the eleven quantifier pairs wherein order is significant.

The table can be collapsed into a small set of *local dominance rules*:

LOCAL DOMINANCE RULES

_/a :	head
_/th:	pmod
_/no:	head
(~no)/ev:	pmod
no/ev:	head

In general, priority seems to be determined entirely by the specific quantification on the *object* of the postmodifier phrase. The one exception,

no/ev:head, involves negation, as does **no/no:***, the one pair from our set of quantifiers which is proscribed for English. In the current implementation all of the consequences of negative quantification are handled as special cases. It seems reasonable, though, to conjecture that negation calls for its own logic for referential processing. This thesis suggests *some* of the issues involved:

```
1) Negation reverses the evaluation of predicates.
2) "no" is interpreted as the universal quantifier
   plus a negative particle on the associated
   predicate.
3) The associated predicate is always that on the
   same clause level as the "no."  (principle of
   local negation)
4) Negative quantification on the head noun
   effects the local dominance within a post-
   modified noun phrase.
```

While indicating an alternative treatment to that specified by Hypothesis 2 (and 2'), the above list does not suggest how the recursively postmodified NP of Hypothesis 3 should now be resolved. The latter will be invalid at any clause level at least as frequently as would Hypothesis 2 at the topmost level, which is too frequent for comfort. Fortunately, inspection of samples of extended postmodifiers suggests that the local dominance rules *can* be applied recursively.

Consider the following sentence:

```
the trunk in a closet in every house holds a body.
```

The Scope List for the preferred reading is:

```
every(house)a(closet)in(closet,house)the(trunk)...
          in(trunk,closet)a(body)holds(trunk,body)
```

The postmodifier predicate "in(trunk,closet)" *could* be placed after "a(body)" without altering the resulting interpretation. However, it is obvious that it will be more efficient to test a database entry against *all* relevant conditions as soon as it is introduced, so as to minimize the work spent on fruitless paths. Moreover, it has been demonstrated that the ordering used is simply and directly derived by RVG syntax and semantics.

We will try to maintain as much as possible the economy of the actions used by Stevens, as we undertake the construction of the Scope List shown above. First, "every(house)" is inserted onto the Scope List by NPCLOSE at the end of the extended NP. The deepest and first completed postmodifier phrase, "...a closet in every house" inserts "a(closet)" *before* "every(house)", and the predicate-node "in(closet,house)" after both. At this point:

```
Scope List = every(house)a(closet)in(closet,house)
```

The parse then reverts to the higher level postmodifier phrase: "the trunk in a...." The correct ordering of the Scope List depends on "the(trunk)" being attached at the end of the Scope List.

If we compare the subject quantification with that on the front of the partially constructed Scope List, we find that the __/**ev:pmod** rule shows the desired result: append "the(trunk)" to the Scope List and then add "in(trunk,closet)".

The above example was described using a single Scope List to which the parser added nodes at all clause levels. This works only if we have a solitary postmodified NP as the first NP in the sentence. To generalize, we need to posit that changes in clause level also cause a change in Scope List level. The local Scope Lists are appended to or prefixed as appropriate when MODCLOSE fires. The necessary operations are:

```
1) adding the header-node to either the front or back
   of the lower level Scope List, as prescribed by
   the applicable local dominance rule
2) appending the lower level Scope List to the end of
   that of the next higher level clause
3) appending the predicate node to the end of the
   unified Scope List.
```

4.3 Embedding Local Dominance Rules

While the few local dominance rules identified are simple in themselves, I have yet to hit upon a schema for ordering the necessary primitive actions by means of ternary features in a way that is not cumbersome. I have therefore opted to combine them within a single action so that the aggregate can replace the previous processing of MODCLOSE.

A concise characterization of the new processing for MODCLOSE, where:

```
        SList1 represents the Scope List for a parent level
        Slist2 represents the Scope List for a dependent level
    and SubjNode represents the pending node for a head-noun
```

is:

```
MODCLOSE:
    1)   IF List2 starts with no(noun) AND
              SubjNode = no(noun)
         THEN   /* double negation! */
            MODCLOSE fails
         ELSE
         IF SList2 starts with ev(noun) AND
               SubjNode = no(noun)
         THEN prefix SubjNode to SList2
         ELSE
         IF SList2 starts with no(noun) OR a(noun)
         THEN prefix SubjNode to SList2
         ELSE
              append SubjNode to SList2.
    2)   Append the modifier-predicate node to SList2.
    3)   Append SList2 to SList1.
```

The attachment of local Scope Lists might be understood easier by an example. Consider the following sentence and its parse:

```
SENTENCE: the teacher hit a student in every class near the
          office.
```

```
text       :the              teacher    hit                  a...
actions    :SUBJ:NP:DET(the):N(teacher):NPCLOSE:VTRANS(hit):...
Scope List:                  <level 0:> the(teacher)         ...


text       :                student    in                   ...
actions    :OBJ:NP:DET(a):N(student):POSTMOD:ZEROSUBJ:       ...
Scope List:                            <level 1:>            ...
```

```
text        :          every          class    near             ...
actions     :PREP(in):NP:ADJ(every):N(class):POSTMOD:ZEROSUBJ...
Scope List:                         <level 2:>                  ...

text        :                the          office    .
actions     : :PREP(near):NP:DET(the):N(office):NPCLOSE:        ...
Scope List:                                                     ...

text        :
actions     :                MODCLOSE...
Scope List: the(office)            ...
```

At this point there are three different Scope Lists accounted for by the parser, with the following contents:

```
level 2: the(office)
level 1: [PENDING: every(class)]
level 0: the(teacher) [PENDING:  a(student)]
```

MODCLOSE must first reconcile the level 2 Scope List with its head noun node "every(class)". The __/**the:pmod** rule requires placing the head node second, producing "the(office)every(class)." Having consolidated clause level 2 with level 1, MODCLOSE can now append the postmodifier predicate. Two Scope Lists remain:

```
level 1: the(office)every(class)near(class,office)
level 0: the(teacher) [PENDING: a(student)]
```

A second MODCLOSE will fire, and reconcile the pending node, "a(student)," and the level 1 Scope List via the __/**the:pmod** rule. At the conclusion of this second MODCLOSE the parse will have reverted back to clause level 0 with the following Scope List:

```
level 0: the(teacher)the(office)every(class)
           near(office,class)a(student)in(student,class)
```

All that remains at this point in our example is for CCLOSE to fire, which will add the main-predicate node "hit(teacher,student)" to the end of the level 0 Scope List. Parse and Scope List construction terminate together.

Chapter 5
Using the Scope List

5.1 Referential Processing

Having finished the parse and fully ordered the Scope List, we turn to the task of referencing the database. The current system passes the Scope List to a routine called **QuantSearch**, which after checking the quantification indicated in the first node, passes the Scope List on to one of the procedures: **A**, **THE**, **EVERY**, **NO**, **ZERO**, **MOD** or **PRED**. The first four handle the specified mode of quantification, **ZERO** instantiates an unquantified object (e.g.: a proper noun), **MOD** instantiates a modifier/predicate, and **PRED** handles the main predicate. Upon success, each records the unique index of the matched database (DB) entity in the INST slot of the matching parse entry. All except PRED then invoke QuantSearch, passing it the Scope List minus its first node. PRED, called at the end of the Scope List, passes back to the cascade of calling procedures the truth value of the input clause with respect to database referents.

Consider a simple example. The sentence: "the child smiled" yields the following parse entries:

```
parse-entry # 1          parse-entry # 2
LAB:  child              LAB:  smile
INTR: vector A           INTR: vector B
INST: the                INST: pred
REAS:                    REAS:
                         ARG1: parse entry #1
```

The instantiation of these will start when QuantSearch passes the Scope List: **the(child)smile(child)** to THE for processing. Suppose that THE identifies child uniquely with the DB object with a unique index, say **376**. The

INST slot of the entry gets that index number:

```
parse-entry # 1          parse-entry # 2
LAB:  child              LAB:  smile
INTR: vector A           INTR: vector B
INST: 376                INST: pred
REAS:                    REAS:
                         ARG1: parse entry #1
```

The abridged Scope List, **smile(child)** is passed on to another invocation of QuantSearch. The second QuantSearch will in turn call PRED, which will search the database seeking some entry that matches parse entry #2. To match this, a DB entry's ARG1 must point at DB entry #376.

5.2 Match3, AsymMatch3, and DBMatch3

All instantiations are accomplished by traversing the database and comparing each DB entity in turn with the parse entry to be matched. The database entities are themselves entry structures, with basically the same structure as parse (CPSR) entries. Evaluating a DB entry against a parse entry consists of comparing the contents of corresponding slots.

The matching that goes on between slots is neither an equality test nor Match3. It is permissable for a matched DB datum to be more specific than that of the corresponding parse entry, but the reverse is not true. If there's a "red shoe" represented in the database and sentence refers to "the shoe" then the reference should succeed, but if the database entry does not specify color and a sentence refers to "red shoe" then the two cannot be automatically equated with each other. Ternary vectors will therefore be matched using the ternary operation **AsymMatch3**. AsymMatch3 compares a *target* vector against a *goal* vector, succeeding if the goal is consistent with and no less specific than the target, and otherwise failing. The match is accomplished by making any - or + of the target vector match by equality against the goal, while allowing a ? in the target to match anything:

```
target: ???+++---          target: ++??-
goal  : ?+-?+-?+-          goal  : ++-?-
          x xxx fails                    succeeds
```

In comparing any two slots, one of four cases will arise:

```
       DB slot points to:      Parse Entry slot points to:
(a)       vector                  vector
(b)       vector                  entry
(c)       entry                   vector
(d)       entry                   entry
```

The first case has been addressed already. In case (b), the vector of the DB slot is *a priori* less specific than the entry pointed at by the parse slot, so this situation causes a failure of DB reference. In Case (c), however, the more tightly constrained DB entry *will* match with the vector of the parse slot, *if* AsymMatch3 succeeds between the parse vector and the DB entry's INTR vector. In case (d) the DB entry pointed at from the DB slot must first be matched against the Parse slot entry, which is a simple recursion of the entry match being interrupted.

A gray area remains in the matching of Parse versus DB entries. Suppose AsymMatch3 fails on two slots for which Match3 succeeds. Although the DB and parse data are semantically consistent, the parse data is more specific. Whereas the success of AsymMatch3 guarantees a match, its failure entails, not a nonmatch, but rather that the DB information is *underconstrained.* That is, the DB lacks sufficient data to resolve the match.

Consider once more a DB object for a "shoe" which has has a feature value ?RED. Although a parse entity "red shoe" will fail to AsymMatch3 with this object, it is not safe to assume that the feature is **-RED.**

In this situation the current system will poll the user. The poll for our example entries might be:

```
DO YOU MEAN shoe305 FOR shoe ?
```

where **shoe305** is a database referent and **shoe** the LAB slot value of the parse entry. If uncertain about the reply, the user can use other system routines to browse the desired database entries. A response of "no" will cause the current DB entry to fail the overall matching attempt. A positive response will cause the program to Refine3 the appropriate vector of the DB entry with the more specific Parse vector, and the match will succeed. Thus the database may "learn" new information, and the referential system neither rejects nor accepts such references arbitrarily.

This overall process is combined into **DBMatch3**, which resolves the match in this manner:

```
DBMatch3:     IF AsymMatch3 succeeds THEN DBMatch3 succeeds
              ELSE
              IF Match3 fails THEN DBMatch3 fails
              ELSE
                prompt user and react to response.
```

5.3 Recursive Instantiation

Now it is worthwhile to examine the implementation of the individual procedures that handle the various quantifiers.

5.3.1 PRED

PRED searches the database entries for a successful DBMatch3 with the main predicate. Its reaction to success or failure depends on the contents of the parse entry's INST slot. If the latter is empty then PRED responds to a successful DBMatch3 by:

1) setting the parse entry INST slot with the index of the matching database entry.
2) setting a success status in a parameter to be passed back to its calling procedures.
3) terminating.

If the INST slot is empty and no match is found the response is to set a -1 in the REAS slot, indicating to the system that no database match exists, and therefore to regard the proposition as false.

If the negative quantifier "no" occurred at the main clause level of the input sentence, the INST slot of the predicate entry will have been set to signal negation. In this case, failure to find a match induces PRED to terminate with a success status. A valid DBMatch3 will in this case constitute a counterexample to the negation, inducing PRED to:

1) set the index of the damning DB entry in the REAS slot, thus flagging this Scope List node as a failed reference, along with storing the counterproof for conceivable responses by the system.
2) terminate with a fail status.

5.3.2 MOD

MOD behaves like PRED in that it seeks any DBMatch3 against its associated predicate entry, and varies its response to a match (or lack of the same) according to whether the INST slot initially indicates negation or not. Its behavior differs from PRED due to the necessity of passing on a Scope List to other procedures, the failure of which may force MOD to *reinstantiate* against a different DB entry.

Before starting its search of the database, MOD saves the INST value in a temporary variable.

If no match is found, MOD responds exactly like PRED.

If negation is not in effect, it will respond to a successful DBMatch3 by:

1) putting the DB entry's index in the predicate node's INST slot.
2) passing the tail of the Scope List to QuantSearch.
3a) If QuantSearch returns a success status, MOD will likewise terminate successfully.
3b) If QuantSearch returns a fail status, MOD will look in the ARG2 slot for the pointer to the failed postmodifying phrase's head-noun entry. It will then set the REAS slot

of *that* entry with the index of the DB predicate for which the reference failed, and then terminate with a fail status.

The reason for saving the failed INST value in the REAS slot of the predicates subject in (3b) is so that when the failure propagates back to the procedure handling the instantiation of the predicate's subject entry, that procedure may distinguish a failure of the modifying phrase from that of an independent clause element.

If negation is in effect, a successful DBMatch3 causes MOD to terminate with a fail status.

5.3.3 ZERO

As a proper noun must be unique, ZERO will never reprocess once the referent has been identified. Thus it tests its INST and REAS slots as soon as it begins, and if it finds a valid DB index in the former and a null value for the latter, it will immediately pass the tail of its Scope List to QuantSearch, leaving the previous instantiation intact.

Upon a DBMatch3 success it will:

1) set the INST slot with the DB index.
2) invoke QuantSearch, and
3) terminate, passing on whatever status it gets back from QuantSearch.

Otherwise, if ZERO fails to find a single DBMatch3 in the database, it terminates with a fail status and -1 set in the REAS slot.

5.3.4 A

The procedure A will run DBMatch3 against every DB entry until a success occurs or the entire database is traversed. In the former case, it will set the INST slot of the Parse entry with the index to the matching DB entry and call QuantSearch, passing it the beheaded Scope List. If QuantSearch returns a fail status, A will continue its trek through the database trying for another match. If QuantSearch returns a success status, A will do likewise and terminate.

If the entire database is checked without a successful termination occuring, A will set a -1 in its REAS slot, and terminate with a fail status.

5.3.5 THE

For the same reason and by the same approach as ZERO, THE will never seek to reprocess an entry it has successfully instantiated.

Otherwise THE seeks until it gets a successful DBMatch3, or else marks its failure and terminates in the same way as does A.

Upon finding a single DBMatch3 success, it sets its REAS slot with a null value and then invokes QuantSearch. If the latter returns a fail status then THE checks the REAS slot. A null value signifies that the clause failed, causing THE to pass on the fail status and terminate. A non-null REAS value indicates that the failure was of a postmodifier of the current entry. In this case THE behaves as if the previous DBMatch3 had failed, and continues its DB search.

This last twist in the processing functions as if, in examining the phrase, "the man with a gun," the system finds a database object that matches "man," only to realize later (when MOD fails to DBMatch3 **with(man,gun)**), that

that DB object doesn't *really* match "man" at all. It continues with its DB search from the point at which it made the "mistake."

If QuantSearch succeeds, then THE will continue its search through the database, seeking a second match. If it fails to find one, then there is indeed one and only one match for the "the"-quantified entry, so it terminates with a success status. On the other hand a second match, if not discounted by a MOD failure, will cause THE to fail. This is a failure of the uniqueness condition of "the"- quantification. It is marked as such for the system through the assignment of the index of the first DB match to the INST slot and that of the second to the REAS slot. The system can thereafter find the (first) two competing instantiations by inspection of these two slots.

5.3.6 EVERY

Should EVERY fail to find any DBMatch3 for its entry, it will quit with a status of 'undefined': avoiding the confusion that might be caused by a specious success due to a null member set. EVERY will otherwise succeed unless it finds an instantiation for which the remnant of the Scope List fails.

It will seek any DBMatch3, and having set INST, it will invoke QuantSearch. If QuantSearch returns a fail status, EVERY responds to a dependent MOD failure in the same way as does THE. A non-local failure causes it to terminate, passing on the fail status.

Barring the return of a terminating failure from QuantSearch, EVERY will traverse the entire database, terminating with a success status so long as at least one DBMatch3 has succeeded.

5.3.7 NO

NO simply calls EVERY, using identical parameters. This is because negation is a universal quantification that reverses the truth value of the associated predicate. Once the parse completed processing the lexeme "no," the truth-reversal function of negation devolved to the PRED or MOD procedure, via the negation indicator in the predicate's INST slot. From that point the "no" quantification was distinguished from that of "every" so that the correct local dominance rules were invoked to order the Scope List, and so that any displays or traces of the Scope List would be readily comprehensible to the viewer.

5.4 Updating the Database

If the highest level QuantSearch returns a success status, then every parse entry has been instantiated as required, and the input sentence represents a fact entered in the database. In this case the message: **"That fact is known"** is displayed to the user.

A failed reference results in the message: **"Input text not verified."** At present we are not storing negated predicates in the database so the referential component will terminate at this point if the failed Scope List has a 'pred' or 'mod' node with a negative indicator. Otherwise the procedure will query: **"Do you wish to update the Database?"** A reply of "no" causes referential processing to end.

An affirmative response will cause the Scope List to be traversed again, and instantiation forced. For each node one of the following cases will apply:

```
1) The node has been successfully instantiated.
2) The node has failed instantiation.
3) The node was never matched against the database.
```

5.4.1 Processing an Instantiated Node

A successful instantiation is recognized by the node entry's having the index of a valid DB entry in its INST slot, and a null value in its REAS slot. In this case there is usually nothing more to be done: the valid INST value of the node entry is left intact and processing advances to the next node on the Scope List.

An exception is an instantiated node with "every"-quantification. This will have been instantiated to the last DB match found for this entry before the subsequent Scope List failed. EVERY is reinvoked for this node, but with a flag **DBUpdate** set to force a solution at every subsequent point of failed reference, by the means detailed in the next section.

5.4.2 Processing for a Failed Instantiation

Failure of reference is indicated by a non-null REAS value. There are only two such failures that have to be handled. The first is caused by the nonexistence of any matching DB entry. This is the sole cause of failure in PRED, MOD, ZERO and A nodes, and is identified by the value of -1 in the REAS slot. It is corrected by a procedure called **AddToDB**, which creates a new database entry that mirrors the semantic and relational constraints of the parse entry. AddToDB sets the new DB entry's index in the parse entry's INST slot, and assigns the REAS slot a null value.

The second failure is due to a multiplicity of matches for "the"-quantification. The INST and REAS slots of the offending parse entry will contain the two matches identified. A routine called **WhichOne** resolves the conflict for the current Scope List by polling the user in exactly the same format as does DBMatch3. If "the person" is the failed reference and the INST and

REAS values are 308 and 412, WhichOne may ask:

```
DO YOU MEAN woman308 FOR person?
```

An affirmative response to this question results in the approved instantiation being set in the parse entry's INST slot. A negative reply will invoke the same question for the second match:

```
eg: DO YOU MEAN man412 FOR person?
```

A second negative will cause WhichOne to begin its own database search for a match, starting after the second refused index. It will make the same query with respect to every DB match until one is approved or the possibilties are exhausted. The latter eventuality will result in a call to AddToDB to provide a default match.

The procedure WhichOne could prove unwieldy in a large domain of objects, but we assume that use of a large domain presupposes a method of restricting the search to some subdomain by some mechanism for *focusing* the search, such as that proposed by Grosz and Sidner [5].

5.4.3 Processing an Untried Node

This situation will apply to those nodes on the Scope List subsequent that at which the initial fatal reference failure occurred. These nodes are distinguished by a null value in the REAS slots of the associated parse entries; and INST slots with either null values or quantification indicators (e.g.: pred, no, etc). The correctly quantified reference is attempted for each node. However, if a fail status returns to the first of these nodes then AddToDB or WhichOne as appropriate is called. Whenever the first such node is resolved, whether normally or by one of the two forcing procedures, the process begins anew for the remainder of the Scope List until all have been resolved.

Chapter 6
Conclusion

This paper presents a set of issues that a reasonably robust implementation of referential semantics might be expected to be responsive to, and details the processes by which the current implementation handles them. It describes the exigencies of quantification following from the normal use of "every," "no," "a" and "the" in the interpretation of input text, along with the instantiation of proper nouns and predicates.

Semantic objects and predicates are ordered for efficient reference on a Scope List. The RVG natural language processing system in which the referential component is embedded makes for a simple and direct process of building the Scope List, largely due to the efficacy of ternary representation and the straightforward ordering of actions possible in the parse. The process of matching parse structures with database entities takes advantage of several distinct match operations that can apply among ternary vectors. Underconstrained database information is handled by polling the user.

Antithetical results surfaced during implementation for some procedures which John Stevens designed, based on Colmeraurer's rules. The rules have therefore been revised to account for *local dominance* in quantifier scope for postmodifier constructions.

In using the Scope List for interpretation of an input statement a distinction must be made between a reference failure local to a postmodifier phrase and failure of an independent clause-level structure, as the former failure can be recovered from. Finally, the system permits the addition of non-negated entries to the database.

There are a number of directions for possible development of the RVG referential component. The current implementation requires little modification to handle questions about database objects. Yes/no questions such as "is the block on the table" are resolvable by instantiating for "the block is on the table." WH-questions such as "where is the beef?" can be handled by DBMatch3ing "at(beef,??)" against the database, allowing ?? to match with anything. Reference for imperatives consists of identifying the preconditions for the command action and producing entries that represent the result events.

The simplest extension to the scope of the current referential component with the greatest potential impact for quantification is probably the handling of plurals. This would effect quantification by such terms as "some," "most" and all forms of enumeration. Other extensions presuppose other ambitious enhancements to the overall system. Here is a list of some of the more common quantification types that might be added, along with system features they depend upon:

1) **adjectives such as "many", "big", "almost all".**
Interpretation of such relativistic terms presupposes *normative* knowledge in the system, or processes for deriving such knowledge. For instance: a "small elephant" is still a "large animal," at least by human standards. See ps 462-478 of Lyons [7].

2) **pronouns.**
These require a retention of discourse entries for use in referencing, along with some form of *discourse focus* for pairing pronouns to the appropriate objects; e.g.:
"Wilma gave Betty *her* keys."
It might be proposed as a default that a parse should *prefer* to link "her" with the subject "Wilma" unless counterindicative contextual information intervenes. See Ritchie [9] for a treatment of context sensitivity for constraining parse.

3) **genitive quantifications.**
Constructions such as "John's camera" can constrain the objects applicable for "camera" only if the system is cognizant of such complex relations as possession.

Certain constructions related to genitive relations *reverse* our scope predictions for other prepositional post-modifiers. Contrast these two sentences:

1) Sal knows an island in every ocean.
2) Sal knows an island with (possessing, having, etc.) every species (of palm tree).

Our rules, derived from Colmeraurer and Stevens, predict these scope readings:

1s) every(ocean) an(island) in(island,ocean) ...
2s) every(species) an(island) with(island,species)...

(1s) seems right: for every ocean there is an island in that ocean that Sal knows. But (2s) does not: it's not that Sal knows a species of every island, but that there is a particular island that he knows on which one may find every species in question. The right scope reading for (2) should therefore reverse the quantifier order of (2s):

2s') an(island) every(species) with(island,species)...

This new wrinkle seems to be related to a class of stative predicates that consistently have this effect (*have*, *possess*, *contain*, etc.). If so, this distinction could be handled by distinguishing syntactic categories that are in turn associated with different scope list building actions.

4) ambiguous definite reference.
The handling of multiple objects identified in text by the same label requires that the system be able to restrict its attention to subdomains of the database, and determine the extent of such subdomains during discourse. Thus the correct referent for "John" depends on any prior discourse (e.g.: we have been discussing the Kennedy brothers) and context (e.g.: John Stevens is the only person named John familiar to those in dialog). As mentioned previously, Grosz and Sidner offer a method of appropriately partitioning the realm of discourse objects in [5].

References

1. Bickerton, D. "Creole Languages". *Scientific American 249* (1983), 116-122.

2. Blank, G. D. "A New Kind of Finite-State Automaton: Register Vector Grammar". *IJCAI 85: Proceedings of the Ninth International Joint Conference on Artificial Intelligence 2* (1985), 749-755.

3. Blank, G.D., Solderitsch, N. and Stevens,J.C. Natural Language Semantics using Ternary Feature Vectors. CSEE Department, Packard Lab 19, Lehigh University.

4. Colmeraurer, A. An Interesting Subset of Natural Language. In *Logic Programming*, Academic Press, New York, 1982, pp. 45-66.

5. Grosz, B.J. and Sidner, C.L. Discourse Structure and Proper Treatment of Interruptions. In *IJCAI 85: Proceedings of the Ninth International Joint Conference on Artificial Intelligence*, International Joint Conference on Artificial Intelligence,2, 1985, pp. 832-839.

6. Leech, G.. *Semantics.* Chaucer Press, 1981.

7. Lyons, J.. *An Introduction to Theoretical Linguistics.* Cambridge University Press, 1979.

8. Rand, A.. *Introduction To Objectivist Epistemology.* New American Library, 1966.

9. Ritchie, G. Semantics in Parsing. In *Parsing Natural Languages*, Academic Press, 1983.

10. Stevens, John C. Reference and Quantification in a Register Vector Grammar Natural Language Processor. Master Th., Lehigh University,1985.

Vita

Bryan James Murphy, the son of Bryan James and Donna Murphy was born on May 7, 1955 in Valparaiso, Indiana. He attended Lehigh University and received his Bachelor of Arts Degree in Mathematics in June of 1977. He taught high school mathematics to the junior and senior classes at Navua High School, Fiji Islands; and was Acting Head of the Math Department there from January of 1978 through December of 1979. He has since been employed as an application programmer in Bethlehem, Pa. He has been pursuing graduate level study in Computer Science at Lehigh University on a part-time basis since 1980.

Vita

Bryan James Murphy, the son of Bryan James and Donna Murphy was born on May 7, 1955 in Valparaiso, Indiana. He attended Lehigh University and received his Bachelor of Arts Degree in Mathematics in June of 1977. He taught high school mathematics to the junior and senior classes at Navua High School, Fiji Islands; and was Acting Head of the Math Department there from January of 1978 through December of 1979. He has since been employed as an application programmer in Bethlehem, Pa. He has been pursuing graduate level study in Computer Science at Lehigh University on a part-time basis since 1980.