THE UNIVERSITY OF CALGARY

RV-Tools:
Development Tools for Building Register Vector Parsers

by

David R. Astels

A THESIS

SUBMITTED TO THE FACULTY OF GRADUATE STUDIES

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE

DEGREE OF MASTER OF SCIENCE

DEPARTMENT OF COMPUTER SCIENCE

CALGARY, ALBERTA

SEPTEMBER, 1994

THE UNIVERSITY OF CALGARY

FACULTY OF GRADUATE STUDIES

The undersigned certify that they have read, and recommend to the Faculty of Graduate Studies for acceptance, a thesis entitled, "**RV-Tools**: Development Tools for Building Register Vector Parsers" submitted by David R. Astels in partial fulfillment of the requirements for the degree of Master of Science.

[illegible]

Bruce MacDonald,

Department of Computer Science

[illegible]

John Archibald,

Department of Linguistics

[illegible]

Mildred Shaw,

Department of Computer Science

Date Nov 25, 1994

Abstract

Natural language understanding is and always has been an important area of research and development in artificial intelligence. Natural language understanding is made up of several processing stages, including morphological, lexical, syntactic, semantic, and discourse. The stages of processing morphology, lexicon, and syntax are commonly combined and called parsing. For natural language understanding to be widely useful an efficient model of language must be used for each stage, and it must be convenient for a system developer to implement language processors using the model.

The Register Vector (RV) model of language is an efficient parsing model, having fixed space complexity and linear time complexity. RV is a low-level formalism, which results in its efficiency but also makes it difficult to build RV parsers.

The work described in this thesis is an attempt to make it easier to develop RV parsers by providing a set of interactive tools. These tools allow non-linear, interactive, incremental browsing/editing of parsers, immediate error feedback, as well as interactive debugging.

The system described here has been implemented using Smalltalk-80 on Sun 3 and SparcStation platforms. This work is the first to try to develop a convenient, interactive environment for developing RV parsers.

Acknowledgements

I would like to thank the following people, without whom this thesis would not have been possible:

- Bruce, my supervisor, for his moral and financial support, and for being a friend as well as a supervisor;
- NSERC, for their financial support in the form of PGS A & B scholarships;
- The faculty and staff of the computer science department at the University of Calgary, who made me feel at home from the start, and who were always friendly and helpful;
- My friends who read drafts of the thesis, for their insightful and helpful comments, specifically David McFadzean and Murray Mowchenko;
- My brother, Stephen, for his ongoing support and his assistance with the property simplification algorithm;
- Ted O'Grady, for his contagious conviction that Smalltalk is THE ONLY way to program;
- Ernie Chang, my boss, for giving me the time off work when I needed it to complete this work;
- Kate, my wife, for giving me the time and encouragement to complete the thesis.

Dedication

For Kate, Tasha, and Jason: my family.

Contents

List of Tables

List of Figures

CHAPTER 1
Introduction

Natural language understanding is and always has been an important area of research and development in artificial intelligence. One of the goals of this area is to enable computers to communicate with people in a natural way.

According to (Bar & Feigenbaum, 1981),

> "Artificial Intelligence is the part of computer science concerned with designing intelligent computer systems, that is, systems that exhibit the characteristics we associate with intelligence in human behavior – understanding language, learning, reasoning, solving problems, and so on."

Natural language understanding is the area of artificial intelligence that studies the use of natural language by computer. This area consist of several sub-areas of processing, including morphological, lexical, syntactic, semantic, and discourse. The initial phases of processing (morphological, lexical, and syntactic) are commonly combined and referred to as parsing.

If a language understanding system is to be used interactively, all components must be efficient. Given an efficient model of language parsing, and it must be convenient for a system developer to implement language processors using the model if it is to be used. Many of the systems used for work in natural language parsing, Context Free

Grammars and Augmented Transition Networks for example, have time complexity of $O(n^3)$. This is the price paid for the great power of these system. However, the power of these system is not required to model natural language performance. This is because humans are "limited, finite devices" (Miller & Chomsky, 1963). Register Vector (RV) is a new model of language which models language performance, thus using limited, finite resources. It is efficient, having fixed space complexity, and linear time complexity. The subject of this thesis is the design and development of a set of tools for convenient implementation of RV parsers.

The first step in the work this thesis describes was the implementation of an RV processor. This was done building on Blank's work as well as my own previous work. The resulting system conforms to the current specification of RV at the time the work began (early 1992), with the exclusion of discontinuous idioms (see Chapter 9). This system has been designed with the goal in mind of creating natural language front-ends to end-user applications. As such it does not support the entirety of human language usage. For example, it can not parse more esoteric forms such as poetry. The target languages are subsets of a natural language such as might be used in information retrieval applications[1].

In addition to reimplementing the RV parser engine, I have developed textual and graphical browsers for manipulating the various parts of an RV parser, which are as follows:

- syntactic boundaries;
- syntactic ordering features;
- productions;

[1]Blank's later work involves building an interface to an Air Traffic Information System (ATIS) application.

- semantic features;
- semantic relations;
- morphosyntactic features;
- paradigmatic morphological variation;
- lexical entries.

The thesis work also includes a debugger that allows single stepping, breakpointing, and examination of parser data structures.

RV is a new model of language developed in the mid nineteen-eighties by Blank[2] (Blank, 1985; Blank, 1989). Since it is such a recently developed model there are few tools supporting the construction of RV parsers. The value of RV-Tools is that it provides a convenient, incremental, and interactive way of constructing and debugging RV parsers. RV-Tools is meant to be used by researchers that need to develop RV parsers. They may be using RV to provide natural language capabilities to another system, or working on extending RV itself.

1.1. Motivation

The RV formalism (Astels, 1991; Blank, 1985; Blank, 1989; Blank, 1991) is attractive for use in building natural language parsers. This is mainly due to its efficiency. Storage requirements are determined by the parser designer, and do not grow during operation. RV has a linear time complexity, i.e. $O(n)$ where n is the number of words in the sentence. The number of syntax productions does not affect the time complexity because only a small fraction of them are searched any time one is required. Although it uses limited resources, RV has been carefully designed to model the complexity of language performance by native speakers. This is done by limiting

[2]Glenn Blank is the foremost researcher working on RV.

embedding depth and memory of alternative interpretations in a way that reflects a native speaker's abilities. These limitations enable the fixed space and linear time complexities.

A fundamental problem with RV is that it is a more awkward formalism than more familiar ones (e.g., augmented transition networks (Woods, 1970) and phrase structure grammars (Chomsky, 1957)). Figures 1.1, 1.2, and 1.3 show three systems for recognizing the same simple subject-verb-object language. Subjects and objects have the same form: a name, or a noun optionally preceded by a determiner.

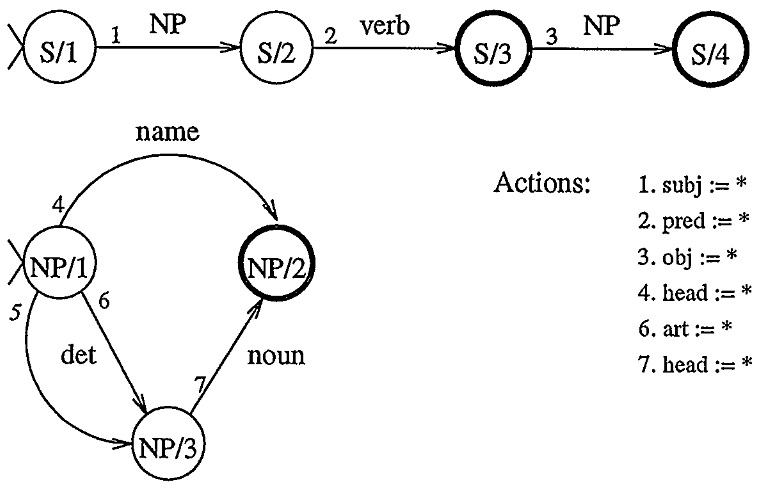


FIGURE 1.1. Augmented Transition Network Example

$$\begin{aligned}
\text{SENT} &\rightarrow \text{SUBJECT VP}\\
\text{VP} &\rightarrow \text{VERB}\\
\text{VP} &\rightarrow \text{VERB OBJECT}\\
\text{SUBJECT} &\rightarrow \text{NP}\\
\text{OBJECT} &\rightarrow \text{NP}\\
\text{NP} &\rightarrow \text{NAME}\\
\text{NP} &\rightarrow \text{NOUN}\\
\text{NP} &\rightarrow \text{DET NOUN}
\end{aligned}$$

FIGURE 1.2. Phrase Structure Grammar Example

SUBJ	+???-	-??++
VERB	-+??-	?-???
OBJ	?-+?-	??-++
CLOSE	--??-	+++--
DET	???+?	???-?
NOUN	????+	???--
NAME	???++	???--

FIGURE 1.3. RV Example

Building an RV parser can be a difficult procedure since designers find it tedious and awkward to specify a language using the RV model. Two approaches to making it easier are: a) automate the process (by using a learning mechanism), and/or b) provide a good development environment. This thesis presents the latter approach, as well as some preliminary work on automated vocabulary acquisition.

1.2. Goals

The goal of this work is a set of RV development tools that meet the following criteria:

- convenient editing of parsers, through the use of structured editors;
- automatic validity checking of identifiers, and suggestion of spelling corrections;
- identification of incorrect actions;
- automatic updating of all uses of an identifier when it is changed (i.e. renamed);
- protection against removal of referenced identifiers;
- the use of shorthand notations where useful;
- the ability to view the parser from various viewpoints;
- support for freely cross referencing information;

- the ability to view the parser at various levels of granularity;
- convenient testing and debugging of parsers, which involves:
  - sentence testbeds;
  - single stepping;
  - access to internal structures;
  - provision of relevant, useful information;
  - convenient alteration of the acquisition mechanism's operational parameters.

Smalltalk-80 was chosen as the implementation language for several reasons:

- Smalltalk is a dynamic, object-oriented language; (this allows interactive debugging, garbage collection, and all the advantages of object-oriented development)
- The Smalltalk environment includes an interactive, visual development environment to use as an example; and
- The implementation of Smalltalk's development environment is available to be used as a basis upon which to build development tools.

1.3. Scope of the Thesis

This thesis describes the RV model of natural language parsing and the development of a set of interactive development tools for building RV parsers. It does not address the problems and issues involved in natural language understanding, or reasoning about the information contained in a language act. This thesis concerns only a system for translating a textual surface form in a natural language into a representation that could be used as input to an understanding system.

1.4. Thesis Outline

This thesis presents the Register Vector parser formalism and a set of development tools for building RV parsers.

Chapter 1 described what was accomplished and the motivation.

Chapter 2 provides a detailed definition and description of the RV formalism as this system implements it.

Chapter 3 examines existing RV development systems. Basic operation is discussed and limitations/shortcomings are detailed. It also briefly discusses how RV-Tools differs from those presented, and presents some related work in the area of lexicon acquisition.

Chapter 4 describes the central concepts of object oriented programming, Smalltalk-80, and the Model-View-Controller application framework.

Chapter 5 describes extensions that I have made to the RV formalism defined in Chapter 2.

Chapter 6 describes the implementation of the RV parser engine that is the central element of the development system.

Chapter 7 describes the design and operation of the RV development tools. This chapter and the next present the majority of this work's original content.

Chapter 8 describes the vocabulary learning mechanism that is part of the development system.

Chapter 9 evaluates the development system, RV parser engine, and vocabulary learning mechanism. Directions for future work are explored.

Chapter 10 summarizes the contributions of the research presented in this thesis.

Appendix A compares RV to non-deterministic finite automata, showing the effects on each formalism as syntactic constraints are weakened.

Appendix B presents several of Blank's RV parsers that were used to test the development tools, along with test sentences.

Appendix C contains the lexicon and sentences used in testing the vocabulary acquisition mechanism (described in Chapter 8).

Appendix F is a user manual for the development system.

1.5. Summary

This chapter briefly introduced RV: a new parsing formalism with favorable time and space complexity. However, RV is a low level formalism, and as such is awkward for parser developers to use. This problem is dealt with by developing a set of tools that make it easier and more convenient to develop RV parsers. The scope of the work presented in this thesis is limited to language parsing.

CHAPTER 2

Register Vector Processors

This chapter defines the RV processor and describes its design. Examples of simple RV parsers are provided. An RV parser is a set of data structures that specify a language. The RV processor is an engine that uses an RV parser to translate sentences in the specified language into a semantic structure. An RV processor operates in $O(n)$ time, where n is the number of words in the sentence (Blank, 1989).

2.1. Definition

An RV parser is defined formally as a nine-tuple:

$$M = (\mathcal{S}, \mathcal{I}, \mathcal{F}, \mathcal{C}, \mathcal{B}, \Sigma, \alpha, \lambda, \delta), \text{where:}$$

$$\begin{aligned}
\mathcal{S} &= \text{a finite set of states, where a state } s \\
& \text{ is a binary n-vector over } \{+, -\} \\
\mathcal{I} \in \mathcal{S} &= \text{the initial state} \\
\mathcal{F} \subset \mathcal{S} &= \text{a set of accepting states} \\
\mathcal{C} &= \text{a finite set of syntactic categories} \\
\mathcal{B} &= \text{a set of syntactic boundaries} \\
\Sigma &= \text{the set of input symbols (i.e. the vocabulary)} \\
\alpha &= \text{a set of possible actions} \\
\lambda : \Sigma \to 2^{\mathcal{C}} &= \text{a mapping from input symbols to categories (the lexicon)} \\
\delta : \mathcal{S} \times \mathcal{C} \to \mathcal{S} \times 2^{\alpha} &= \text{the transition relation} \qquad \blacksquare
\end{aligned}$$

States are encoded by vectors which are ordered sets of elements. Each element can have a value of + or -. Each element of the state vector represents some constraint which is called an *ordering feature*. Possible features include: *the subject has been parsed*, and *expect an indirect object*. Vector elements (i.e. ordering features) are named. For example, the element indicating whether the subject has been parsed could be called `S`.

State vectors are a special case of *ternary vectors*. These are vectors of three-valued elements. Elements can have a value of ? along with + and -. The ? value is a *don't care* and is used to allow discontinuous constraints[1] to pass unaffected through one or more states. The ? element value is a generalized +/- value, thus state vectors are fully specific ternary vectors.

There are three fundamental operations on ternary vectors (in all cases n is the size of the vectors):

`match:`

$$A \text{ match: } B = \bigwedge_{i=1}^{n} (a_i = b_i) \vee (a_i = ?) \vee (b_i = ?)$$

This operation checks if A and B match. Vectors match if each corresponding pair of elements match. A pair of elements match if either is ?, or both are the same.

`change:`

$$A \text{ change: } B = (\forall i = 1 \ldots n),\ b_i \leftarrow \begin{cases} b_i & \text{if } a_i = ? \\ a_i & \text{otherwise} \end{cases}$$

All non-? elements of A are copied to the corresponding position of B.

[1]These are constraints that are not syntacticly adjacent.

`refine:`

$$A \text{ refine: } B = (\forall i = 1 \ldots n),\ a_i \leftarrow \begin{cases} b_i & \text{if } a_i = ? \\ a_i & \text{otherwise} \end{cases}$$

All ? elements of A are replaced by the corresponding elements of B.

The syntax specification of an RV parser consists primarily of a set of productions which have the basic form (*cat*, *type*, *condition*, *change*, *actions*). The transition relation δ is embodied in these productions as well (see below for details).

Cat is the unique name of the production. This corresponds to a member of C which is the set of syntactic categories. For example, there would be a production named **NOUN**, another named **VERB**, and so on.

Type indicates the type of the production. There are four possibilities.

N*on-lexical*: These productions are not constrained by the input, i.e. their use is not dependent on the current input token. Also, they do not consume input. An example is **SUBJECT**.

S*emi-lexical*: These are constrained by input but do not consume it. They are used to implement subcategorization[2]. An example is **TRANSITIVE**.

L*exical*: These productions are constrained by input, and they do consume input. An example is **VERB**

I*nit-final*: These are a special form of lexical productions. Their *change* vectors define the start state. Because of this, all init-final productions must have the same *change* vector. Note that this means that their *change* vectors must not be generalized, i.e. contain no ? elements. The *condition* vectors of init-final productions define the set of accepting/final states for the parser, i.e. $\mathcal{F}$.

[2]The capability of a syntactic entity to determine the complements that it has.

Condition and *change* are ternary vectors. *Condition* is compared, using **match**, to the parser state. If these two vectors match, the production can be used if all of its actions can be successfully executed. *Change* is used to modify the parser state using the **change** operation to produce the next state. *Actions* is a subset of α, possibly empty. These are operations that will be performed when a production is used, after the state transition is made. These actions can perform various tasks ranging from saving backtracking information to construction semantic representations of the input. However, there are a small number of possible actions and they are very simple. Actions are described in detail in §2.9.

The production data structures are shown in Figure 2.1. The productions are in the center. On the left is the mapping from production names (*cat*) to productions, and on the right is the ordering of non-lexical productions.

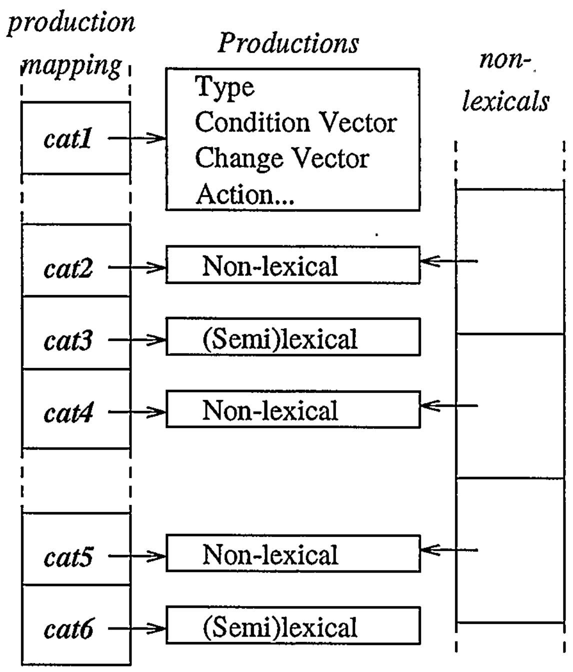


FIGURE 2.1. Production organization.

The transition relation, δ, is a set of triples (s_1, c, s_2) such that:

(1) There is a production (*cat*, *type*, *condition*, *change*, *actions*);

(2) (s_1, **match:** *condition*) returns **true**;

(3) *cat* and c are the same;

(4) After (*change* **change:** s_1), s_1 is the same as s_2.

There is a subtle different between transitions and productions. Since the *condition* and *change* vectors of productions can be generalized, each production can specify a set of transitions.

Figure 2.2 shows the organization of an RV parser. The various parts of the diagram will be discussed in the rest of this chapter. The diagram shows how the various data structures are related (the relation shown is a *has–a* relation). A single box denotes a singular instance of a structure, while two solid boxes indicates exactly two instances (see the ordering feature vectors, there are exactly two per production, condition and change), and the addition of a dashed box indicates that there can be any number of instances.

2.1.1. Operation. Figure 2.3 shows the top level parse algorithm. The operation of the parser engine is divided into cycles. A cycle is considered to be one pass through the core parser algorithm, which is shown in Figure 2.4. Before every invocation of CYCLE the parser performs backtracking and the parser state is saved in the boundary register `Curr` after each successful pass through the algorithm. In this way, the current state is saved between invocations of CYCLE.

The LOOKUP algorithm will be described later (§2.7.4), but a brief description is required here. LOOKUP scans the specified input stream, trying to recognise words. Several words could be recognized. For example, if the input was *"onto..."*, both *"on"* and *"onto"* would be recognised. Each recognised word is encapsulated in an

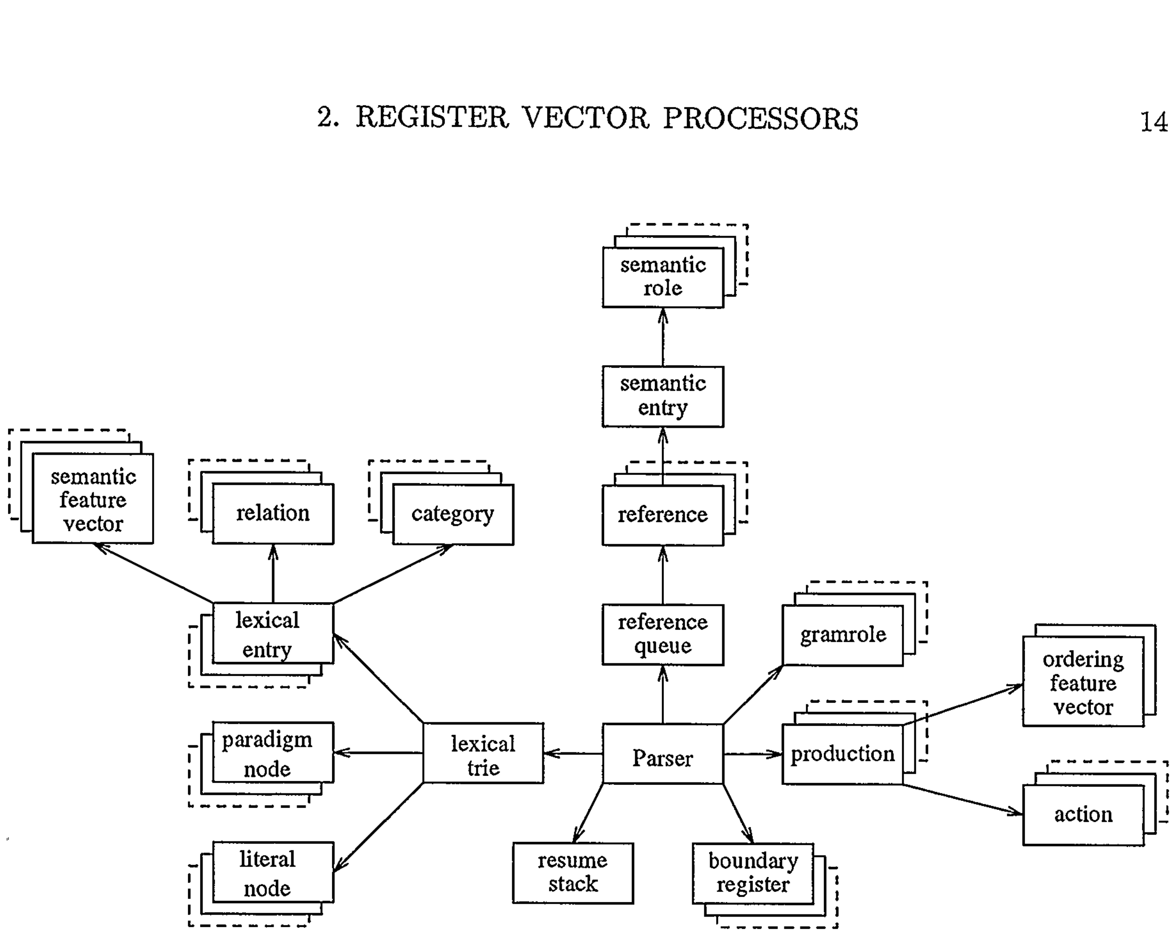


FIGURE 2.2. RV parser organization.

```
PARSE

**interpretations** is assigned LOOKUP(input, lexical trie root, all properties)
      (see Figure 2.10)
if **interpretations** is empty
   signal an unknown word
else
   store the initial state in the WORD and CURR boundaries
   push the WORD boundary on the backtracking stack
   push the CURR boundary on the backtracking stack
   while the backtracking stack is not empty
      pop a register off the backtracking stack and update the parser state from it
      call the CYCLE algorithm (see Figure 2.4)
```

FIGURE 2.3. Algorithm PARSE

```
CYCLE

found is assigned FIND-PRODUCTION() (see Figure 2.5)
if found is a valid production
  disable found from being used again for this interpretation
  found.change change: currentState
  if found.type ∈ {Lexical Init-final}
    if the input is completed and found.type = Init-final
      append a copy of the grammatical role Main0 to parserResults
    else
      interpretations is assigned
          LOOKUP(remaining input, lexical trie root, all properties)
              (see Figure 2.10)
      if interpretations is empty
        signal an unknown word
      else
        save the current state in Word
  save the current state in Curr
```

FIGURE 2.4. Algorithm CYCLE

interpretation. The collection of these interpretations is returned by LOOKUP. This interpretation collection is sorted in decreasing order on the amount of input that each interpretation accounts for. This ordering ensures that, for example, *"onto"* is considered before *"on"*. Interpretations also record what productions have been used while the interpretation was being used. This is done to avoid loops in production use.

When a production is considered for use, three tests are performed which must all succeed for the production to be used.

(1) The production must not have been previously used, as recorded in the current interpretation.
(2) The condition vector of the production must match the current state vector.

```
FIND-PRODUCTION

 *"look for a lexically constrained production"*
while the current interpretation is valid
  let **intrep** be the current interpretation
  for each production **prod** $\in$ **interp**.lexical-entry.categories
    if (**prod**.condition match: **currentState**) and each of **prod**.actions
        can be executed in the context of **interp**
      return **prod**
    else
      advance to the next interpretation
 *"look for a non-lexical production"*
for each non-lexical production, **nonlex**
  for each **intrep** $\in$ **interpretations**
    if (**nonlex**.condition match: **currentState**) and each of **nonlex**.actions
        can be executed in the context of **interp**
      return **nonlex**
return nil
```

FIGURE 2.5. Algorithm FIND-PRODUCTION

(3) All actions associated with the production must be successfully performed.

Lexical and semi-lexical productions are checked first. They are specified by the categorization of the most recent interpretations of the upcoming input characters. The search begins with the active interpretation, and continues through those remaining until one is found that specifies a usable (semi)lexical production.

If no (semi)lexical production is found non-lexical productions are checked. When non-lexical productions are searched, each one is checked for each of the possible interpretations of the upcoming input. When a usable production is found, the interpretation with which it is usable becomes the active interpretation.

The algorithm is shown in Figure 2.5.

Ordering Features: `S V O DET HEAD`

GRAMMAR				LEXICON	
cat	flag	condition	change	word	cats
SUBJ	N	+S -HEAD	-S +DET..HEAD	*George*	NAME
VERB	L	-S +V -HEAD	-V	*ate*	VERB
OBJ	N	-V +O -HEAD	-O +DET..HEAD	*apple*	NOUN
CLOSE	I	-S -V -HEAD	+S..O -DET..HEAD	*the*	ART
ART	L	+DET	-DET	*an*	ART
NOUN	L	+HEAD	-DET..HEAD	.	CLOSE
NAME	L	+DET..HEAD	-DET..HEAD		

FIGURE 2.6. Example RV Grammar and Lexicon

2.2. A Simple Example

To help clarify the above description a simple example is provided here. The following conventions will be used in this and subsequent examples.

(1) Features will be referred to by symbolic names only (e.g. `NAME`). These names are taken from the 'Ordering Features' section of the grammar. Correspondence is by position. The order of these features define the meaning of each vector element.

(2) The notation `+NAME` will mean that the feature `NAME` is given the value +. Similarly for `-NAME` and `?NAME`.

(3) The notation `NAME1..NAME2` will indicate the list of features from `NAME1` to `NAME2`, inclusive. A ternary value can be applied to an entire range as well as a single feature (see item 2).

production	state before SVODH	state after SVODH	input consumed
SUBJ	+++--	-++++	—
NAME	-++++	-++--	*George*
VERB	-++--	--+--	*ate*
OBJ	--+--	---++	—
ART	---++	----+	*an*
NOUN	----+	-----	*apple*
CLOSE	-----	+++--	.

TABLE 2.1. Trace of "*George ate an apple.*"

The grammar and lexicon for a simple language is shown in Figure 2.6. In this example, the lexicon simply associates character sequences with a syntactic category. This example shows how the sentence "*George ate an apple.*" is processed.

Firstly, the current state is initialized by the *change* vector of the init-final production CLOSE: +++--. Also, the first word is read from the input: "*George*". The only lexical category associated with "*George*" is NAME. Since the NAME production requires a state matching ???++ it will not fire yet. The non lexical production SUBJ does match the current state [i.e. **match**(+++--, +???-) → **true**] and thus fires. The current state is changed by SUBJ's *change* vector (-??++), becoming -++++. Now, since (-++++ **match:** ???++) is **true**, the NAME production can fire, consuming "*George*", reading the next word: "*ate*", and changing the current state to -++--. This process continues, resulting in the sequence of production activations shown in Table 2.1.

2.3. Embedding

One of the reasons cited for the claim that natural languages require highly complex parsing systems is clause embedding. This occurs when a modifying clause is embedded in the middle of a sentence. In the sentence "*Men who eat quiche hate* *pizza*", the embedded clause is "*who eat quiche*". The embedded clause is a sentence within a sentence: *who* is the subject, *eat* is the verb, and *quiche* is the object.

It may seem at first that center embedding poses a problem because it seems to require an unlimited, recursive solution. Fortunately, there are constraints on clause embedding in natural languages. Most importantly, human memory limitations place constraints of the depth of center embedding (Blank, 1989; Miller & Chomsky, 1963). The following example (from (Blank, 1989)) demonstrates this. One level of embedding is understandable, but two levels are difficult to understand without extra effort:

(1) The mouse $\boxed{\text{the cat chased}}$ squeaked.

(2) The mouse $\boxed{\text{the cat } \boxed{\text{the dog bit}} \text{ chased}}$ squeaked.

Because of this restriction, the allowable depth of center embedding does not have to be large. Blank claims that only two levels are required beyond the level of the sentence (Blank, 1989). To support this, the parser allows three levels of embedding, the main sentence (level 1) and two levels below that. The level is controlled by a pair of actions `shiftdown` and `returnup` which embed and unembed respectively[3].

This embedding mechanism is required only for center embedded clauses. Both left embedding (genitive phrases, see (3) below) and right embedding (complement phrases, see (4) below) are not limited in many natural languages. They can simply reuse parts of the state vector that have already been finished with, or that will not be used again until the embedded clause has been processed. Although this sometimes causes alternative interpretations to be lost, it does not degrade RV's performance to a level below that of a native speaker.

[3]Actions are discussed in detail in §2.9.

(3) $\boxed{\boxed{\boxed{\text{My mother}}\text{'s girlfriend}}\text{'s husband}}$'s car broke down.

(4) I saw a dog $\boxed{\text{that chased a cat } \boxed{\text{that caught a mouse } \boxed{\text{that ate some cheese.}}}}$

2.4. Backtracking

RV copes with ambiguity by using bounded backtracking. The parser maintains a finite set of *boundary registers*. These hold information about the parser's state at points in the parse to which backtracking can occur. These points are generally at natural syntactic boundaries[4] such as the beginnings and ends of clauses and phrases. The finite bounded backtracking scheme models the limitations of human short term memory; when a boundary is saved any prior contents are forgotten. Church (Church, 1982, p. 57) has noted that "in some sense, [bounded] backtracking, lookahead, and parallelism are all very similar". Boundary backtracking is the primary factor in achieving linear time complexity.

The backtracking mechanism is central to the parser's operation in addition to managing ambiguity. As the algorithm in Figure 2.4 shows, boundary registers `Curr` and `Word` are used to store the parser's state after each cycle and input token, respectively. The first step in the parse cycle is to backtrack. Since boundary registers are accessible in the opposite order they were saved, the first register used will be the `Curr` register. If nothing can be done in that state, backtracking will retrieve the next register stored on the stack.

Part of a parser's specification is a list of boundaries which is used to generate a set of boundary registers. There is a boundary register for each boundary at each allowable embedding level.

[4]See (Carrithers & Bever, 1984) and (Garrett & Bever, 1970) for evidence of syntactic boundaries.

2.5. Subcategorization

Subcategorization deals with the capability of a noun, verb, or other syntactic entity to determine the complements that it has. For example, the verb "*put*" requires both an object and a location. Sentence (5) is correct, but neither (6) nor (7) are.

(5) I put *something somewhere.*

(6) *I put *something.* (no location)

(7) *I put *somewhere.* (no object)

Syntactic entities can also limit their complements. An example of this is the verb *hope* which cannot take a progressive clause (8) or a bare infinitive (9).

(8) *I hope *leaving.*

(9) *I hope *leave.*

Subcategorization also constrains how phrases are mapped semantically. Depending on the verb, the same phrase can have different roles. The following two sentences demonstrate this, in (10) *Sam* is the agent, in (11) it is the object:

(10) Sam is *eager* to please.

(11) Sam is *easy* to please.

RV treats subcategorization as constraints on the grammar that are built into the lexicon. This makes subcategorization easy to implement at the cost of additional complexity in the lexicon. By using semi-lexical productions, a lexical entry can enable the use of its accepted complements.

2.6. Subcategorization Example

The example in Figure 2.7 (from (Blank & Owens, 1990)) shows the use of semi-lexical productions to implement subcategorization. The example parses use the following format:

$$\overbrace{\texttt{SUBJ:NAME:}}^{\textit{productions used}}\quad \underbrace{\textit{George}}_{\textit{input consumed}}\quad ;$$

The following shows how this grammar processes some sample sentences (for now we ignore word form differences):

(12) George hopes.

SUBJ:NAME:*George*; INTRANS:THAT:INF:VERB:*hopes*; CLOSE:.;

(13) George believes Martha.

SUBJ:NAME:*George*; TRANS:XO:THAT:VERB:*believes*; OBJ:NAME:*Martha*; CLOSE:.;

(14) George believes.

SUBJ:NAME:*George*; TRANS:XO:THAT:VERB:*believes*; XOBJ:CLOSE:.;

It is of particular interest to compare (13) and (14). Of interest is the use of the semi-lexical production XO which enables a truncated object. The XOBJ production supplies the truncated object if XO has been used and there is no real object.

2.7. The Lexicon

Lexical analysis in an RV system is designed for time and space efficiency, and is based on characters rather than tokens. This allows multi-word idioms to be recognized.

The lexicon consists of two things:

(1) a collection of structures representing words that the system can recognise; and

(2) a way of indexing these structures so that they can be efficiently recognised.

`Features:  S TR V O DET HEAD XO THAT INF`

GRAMMAR				LEXICON	
cat	flag	condition	change	word	catlist
SUBJ	N	+S	-S +DET..HEAD	love	TRANS VERB
TRANS	S	-S +TR	-TR	believe	TRANS XO THAT
INTRANS	S	-S +TR	-TR -O		VERB
XO	S	-TR +O	+XO	leave	INTRANS VERB
THAT	S	-TR -INF	+THAT	hurry	INTRANS TRANS
INF	S	-TR	+INF		VERB
VERB	L	-TR +V	-V	hope	INTRANS THAT
OBJ	N	-V +O	-O +DET..HEAD		INF VERB
XOBJ	N	-V +O +XO	-O -XO	to	INF
THAT	L	-V +THAT	+S..O -DET..INF	that	THAT
CTHAT	N	-V +THAT	+S..O -DET..INF	George	NAME
INF	L	-V +INF	-S +TR..O -DET..INF	Martha	NAME
CLOSE	I	-S..HEAD	+S..O -DET..INF	robot	NOUN
ART	L	+DET..HEAD	-DET	the	ART
NOUN	L	+HEAD	-DET..HEAD	a	ART
NAME	L	+DET..HEAD	-DET..HEAD	.	CLOSE

FIGURE 2.7. A Grammar Implementing Subcategorization

The lexicon is formally a many-to-many mapping from entry identifiers (word forms) to entries (structures representing semantically related words).

2.7.1. Lexical Entries. Each lexical entry has a unique identifier which is usually the word stem, although there are no constraints on it other than uniqueness. Each entry also contains a set of syntactic and subcategorizing categories which are the names of lexical and semi-lexical productions, respectively. These are considered when this entry is recognised in the input.

The ordering of categories is important as it determines the order in which the parser considers productions. Two general rules of thumb are to place the most commonly used categories early in the list, and to place semi-lexical categories before the lexical category they subcategorize. The reason for this is that lexical categories

Name:	love
Categories:	PASS TRANS ADJPASS NOUN
Wordpath:	love_BED_
	lov_LOVE_
Semantics:	inh -ANIMATE..WHITE +STATE..FEELING -ACTION..TRANSFER
	subj +ANIMATE -STATE..TRANSFER
	obj -STATE..TRANSFER

FIGURE 2.8. Definition of "*love*"

cause the input token to be consumed, and it is thus no longer available. Since semi-lexical categories are dependent on the token they have to be processed first, while the token is available. Semi-lexical categories are subordinate to lexical categories in that a lexical category can be subcategorized by a specific set of semi-lexical categories. For example, transitivity related categories only subcategorize the VERB category.

Each lexical entry also has at least one spelling associated with it. This specifies a path through the lexical trie (see §2.7.3). A lexical entry can be used to represent a single word or a collection of semanticly related words. For example, Figure 2.8 shows the definition of the lexical entry "*love*". This can either be a verb, noun, or passive adjective. The verb and noun forms have different sets of endings, thus requiring two wordpaths.

The final item contained by a lexical entry is a semantic definition. Details on this are given later, in §2.8.

2.7.2. Paradigms.

A paradigm[5] is a collection of ordered pairs (*string*, *properties*) where *properties* is a set of morphosyntactic properties. The pairs are ordered

[5]To be consistent, the use of the term *paradigm* is the same as Blank's (Blank, 1989): "a table of pairs, each associating an orthographic (or possibly phonological) substring with its morphosyntactic properties".

on the length of *string*, in descending order. This ordering is imposed to ensure that the most specific *string* is matched[6].

2.7.2.1. *Morphosyntactic Properties.* Morphosyntactic properties are properties that a word form has as a result of its morphology. For instance *"called"* has the property *past* due to the *"-ed"* affix.

There are two types of properties defined by the RV system: agreement and non-agreement. Agreement properties are used in performing agreement checks (e.g. subject—verb agreement). Agreement properties are grouped by the "feature" they are associated with. To illustrate this, Table 2.2 shows several agreement features and their associated properties.

Feature	Properties
gender	*male*, *female*
number	*singular*, *plural*
person	*first*, *second*, *third*
tense	*present*, *past*

TABLE 2.2. Sample agreement properties

Non-agreement properties are not involved in agreement checks and include properties such as *past-participle*, *infinitive*, and *nominal*. They are used for enforcing agreement within a word (see §2.3 for an example) and as a constraint on production selection. For example, a typical production to process nouns would include the action `lexprop <nom>` to insure that a noun was being processed. All noun forms of words would be given the *nom* property by a noun specific paradigm. In fact an empty paradigm can be created for this purpose, containing a single pair with an empty string and the *nom* property.

[6]As shown in Figure 2.10, paradigm processing is stopped once a *string* is matched.

Word	From P3	From P4	Intersection	Validity
creeping	{inf pres3 prespart}	{prespart}	{prespart}	√
creeps	{inf pres3 prespart}	{pres3}	{pres3}	√
creept	{inf pres3 prespart}	{past pastpart}	ϕ	×
creep	{inf pres3 prespart}	{inf}	{inf}	√
creping	{past pastpart}	{prespart}	ϕ	×
creps	{past pastpart}	{pres3}	ϕ	×
crept	{past pastpart}	{past pastpart}	{past pastpart}	√
crep	{past pastpart}	{inf}	ϕ	×

TABLE 2.3. Possible combinations of paradigms P3 and P4

Agreement feature names (e.g. *"gender"*) are not currently used in the RV system, but the properties are grouped into feature related sets (e.g. *male*, *female*). So, for example, the properties in Table 2.2 would be defined as <*male*, *female*> <*singular*, *plural*> <*first*, *second*, *third*> <*present*, *past*>. These properties work in combination to constrain agreement, e.g. *"we"* is first person plural. To support this, the Cartesian product is taken over all agreement sets. The result is the set of compound agreement properties that are used in checks, e.g. *male:plural:first:present*.

If there are multiple paradigms in the spelling of a word, an intersection is made of the property sets selected from each. If the resulting set is empty then the word does not have consistent morphology. Table 2.3 shows how paradigms P3 and P4 in the path for *creep* are used to eliminate invalid spellings. See Figure 2.11 and Table 2.4 for the lexicon and paradigm definitions, respectively.

2.7.3. Lexical Trie. The lexicon is indexed by a structure similar to a trie(Aho, Sethi & Ullman, 1986). This is a character based structure, and as such supports multi-word idioms. This is possible since space characters are allowed in the spelling of a word. As mentioned above, this is not a true trie in that leaves, which are lexical entries, can have multiple parents[7]. This capability allows the direct support of multiple spellings for a single lexical entry. An example of this was shown in Figure 2.11. The entry *"call"* has two spellings: one corresponding to the verb form, and another for the noun form. In this case the different spellings are required due to the different (and mutually exclusive) sets of allowable endings for each form.

Each internal trie node has a pattern that input is matched against. The pattern is either a literal string, or a paradigm. Each internal node also has one or more descendants that are either lexical entries or tries. There are two types of internal node, depending on the pattern in it:

(1) literal — the pattern is a string literal that is matched against characters in the input stream;

(2) paradigm — the pattern is a paradigm, as described in §2.7.2.

The structure of the internal nodes and leaf nodes (i.e. lexical entries) is shown in Figure 2.9.

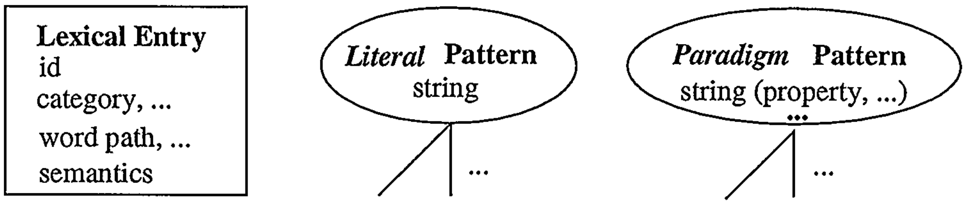


FIGURE 2.9. Trie node structure.

2.7.4. Word Recognition. The RV parser uses the lexicon to find the categories and properties of words in the input. The LOOKUP algorithm of Figure 2.10 outlines how this is done. The algorithm traverses the trie to find a path from the

[7]Internal nodes can currently have only one parent.

```
LOOKUP (input, node, props)

interpretations is initialized empty
case node
  leaf
    if input is empty or the first character is non-alphabetic
      add (node, props) to interpretations
  string pattern
    if node.string matches a prefix of input
      add to interpretations:
        $\bigcup_{\text{child}}$ LOOKUP (remaining input, node.child, props)
  paradigm pattern
    for each (string, properties) of the paradigm
      if string matches a prefix of input and props $\cap$ properties $\neq \phi$
        add to interpretations:
          $\bigcup_{\text{child}}$ LOOKUP (remaining input, node.child, props $\cap$ properties)
        return interpretations
return interpretations
```

FIGURE 2.10. Algorithm LOOKUP

root to one or more leaves that match the input stream and has a consistent set of properties. LOOKUP is a recursive algorithm, and is initially called with the root of the lexicon trie and the set of all possible properties as its second and third arguments, respectively. It returns a list of interpretations (details in §6.3.2.1) — possible lexical entries plus the properties they would have — for the upcoming input. Interpretations are ordered to prefer longer matches, so idioms are preferred over literal interpretations and more specific matches are considered first (e.g. "*onto*" vs. "*on*"). The properties associated with an interpretation can be used for agreement (in tense, number, gender, etc) and to further constrain the selection of productions.

2.7.5. An Example Lexicon. As an example, Figure 2.11 shows the trie for a simple lexicon. The syntactic categories used in this example are: Verb, Noun, and

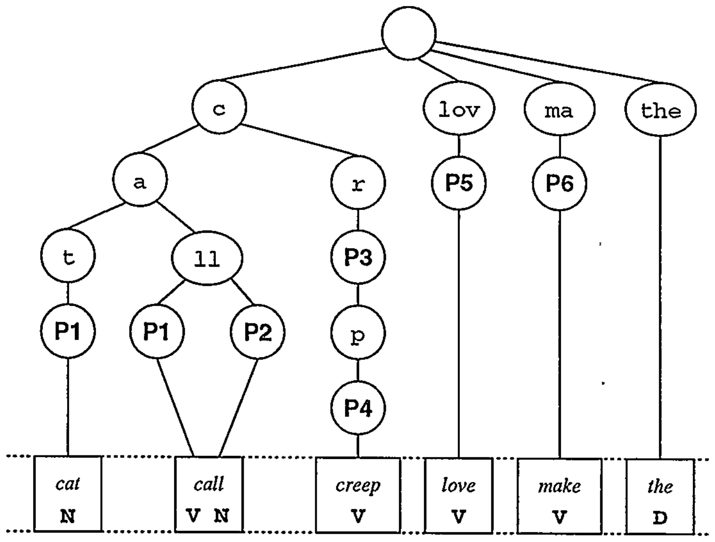


FIGURE 2.11. Example Lexical Trie

Determiner. Table 2.4 lists the six required paradigms including two for "creep", one for the suffixes and one for the "e"/"ee" variation.

2.8. Semantics

A parser has two main purposes: a) to verify that an input is in the language; and b) to convert the input into a more useful form. In the case of an RV parser, this more useful form is a semantic structure that represents the meaning of the input. In order to maintain constant space and linear time complexities this is done simply, efficiently, and in a consistent manner.

2.8.1. Semantic Entries. The representation used for meaning in RV follows the general design criteria of being simple, consistent, and using limited resources. Actions (see §2.9) are provided to build a graph corresponding to the meaning of an input. This meaning graph is built up of frame-like elements referred to as *semantic entries*. Being frame-like, each semantic entry consists of a collection of slots. The

P1 (e.g. "cat")		P2 (e.g. "call")		P3 (e.g. "creep")	
s'	{genpl}	*ing*	{prespart}	*ee*	{inf, pres3, prespart}
's	{gensng}	*ed*	{past, pastpart}		
s	{pl}	*s*	{pres3}	*e*	{past, pastpart}
$	{sng}	*$*	{inf, pres}		
P4 (e.g. "crept")		P5 (e.g. "love")		P6 (e.g. "make")	
ing	{prespart}	*ing*	{prespart}	*king*	{prespart}
s	{pres3}	*es*	{pres3}	*kes*	{pres3}
t	{past, pastpart}	*ed*	{past, pastpart}	*de*	{past, pastpart}
$	{inf}	*e*	{inf}	*ke*	{inf}

TABLE 2.4. Paradigms used by the example lexical trie. *$* means an empty string.

term *semantic roles* is used to refer to the names of these slots. Semantic roles are completely arbitrary and defined as part of the parser specification. Examples of common semantic roles are: `Agent`, `Instrument`, and `Beneficiary`. However, using such meaningful slot names can be misleading. This is because the use of slots depends on the context of the semantic entry (e.g. what verb does it represent?). Blank advocates the use of meaningless semantic roles such as `arg1`, `arg2`, etc. In addition to the predefined semantic roles, there can be any number of modifier slots. These are labelled `Mod1`, `Mod2`, ..., `Modn`. Figure 2.12 shows the organization of the semantic data structures, while Figures 2.13 and 2.14 show the structure constructed for the inputs "*George gave Martha the candy.*" and "*Is the block sitting on the table red?*", respectively.

Semantic entry slots can impose constraints on possible fillers. These are called selectional restrictions and are implemented using ternary vectors. Unlike vectors in productions, elements in selectional restriction vectors are labeled by semantic properties (e.g. ANIMATE, HUMAN, ...). The ternary **match** operation is used to

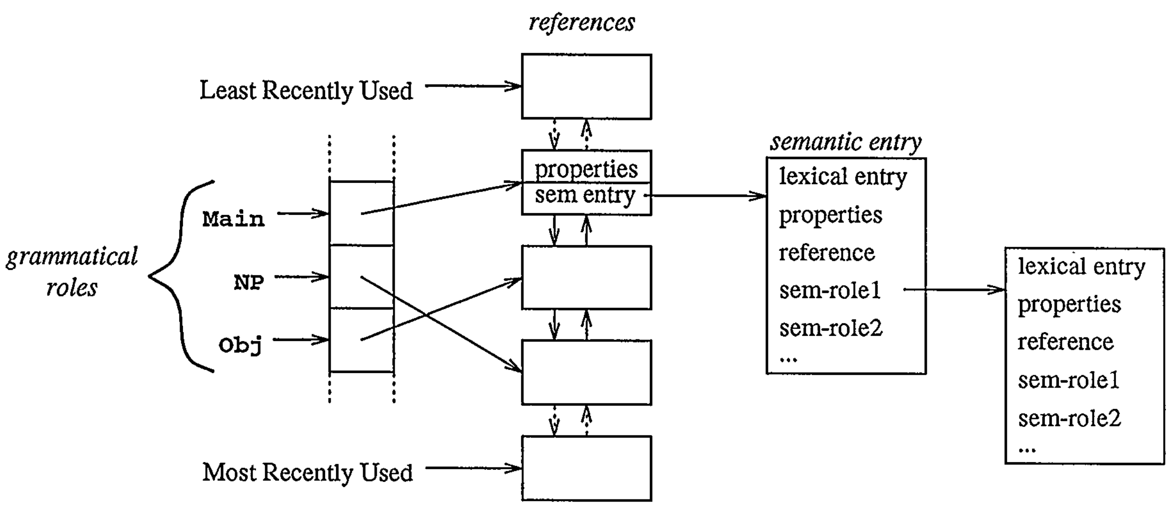


FIGURE 2.12. Semantic Data Structures.

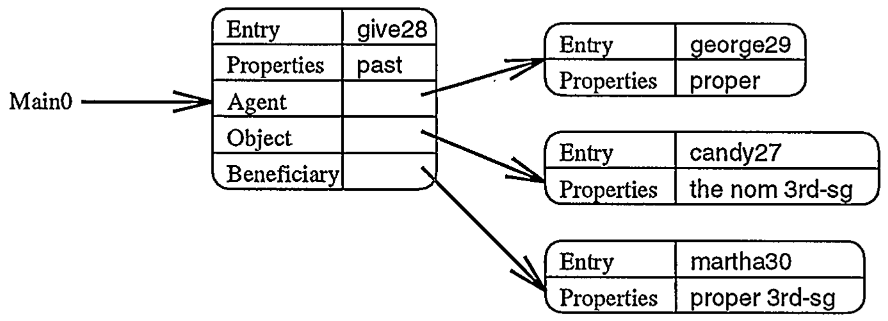


FIGURE 2.13. Structure for "*George gave Martha the candy.*".

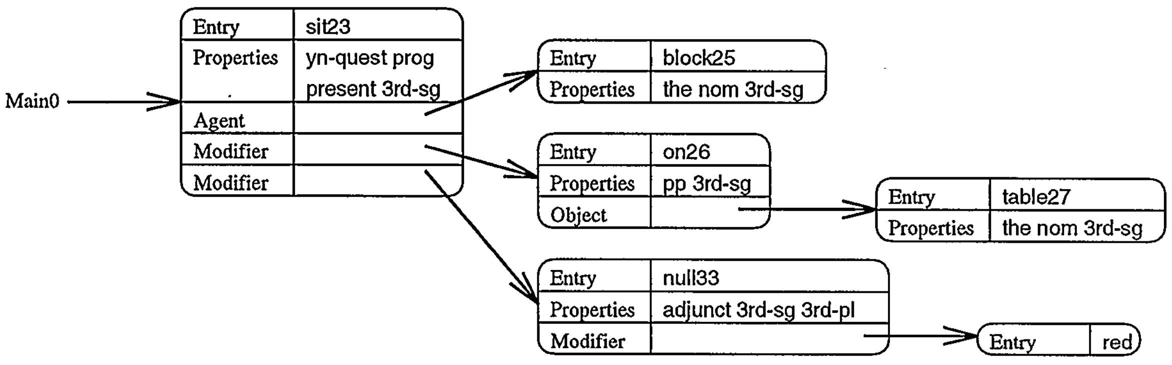


FIGURE 2.14. Structure for "*Is the block sitting on the table red?*".

test role fillers for appropriateness. A role has a vector pattern associated with it and every entity has a vector embodying its inherent properties. These two (inherent properties and role pattern) are matched to test whether an entity can fill a role.

To illustrate the selectional restriction mechanism, consider the parsing of sentences 15 and 16 below:

(15) "*George gave Martha the candy.*"

(16) "*George gave Martha today.*"

In both cases, assume that "*George gave Martha*" has been successfully parsed, resulting in the structure shown in Figure 2.15.

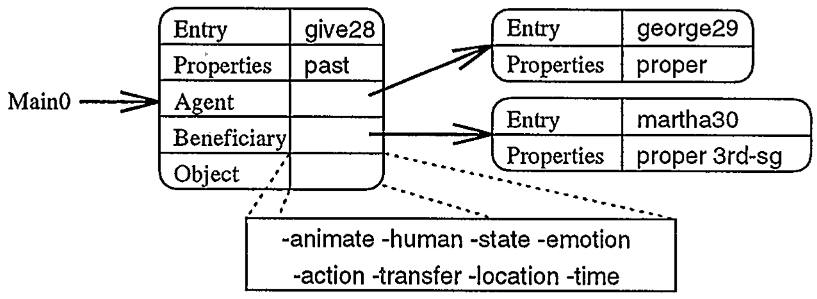


FIGURE 2.15. Structure for "*George gave Martha*".

In (15), the object is "*the candy*" which has the following inherent properties:

{-ANIMATE -HUMAN +MOBILE ?ROUND -STATE -EMOTION
-ACTION -TRANSFER -POSSESSION -LOCATION -TIME -DEST
-INSIDE -SUPPORTED -PERM}

This matches the selectional restriction of the object slot of *give*, specifically:

{-ANIMATE -HUMAN -STATE -EMOTION
-ACTION -TRANSFER -LOCATION -TIME}

Other features are not specified for the slot (i.e. they are implicitly '?') and so are ignored by the match. Thus "*candy*" matches and is used to fill the slot. Figure 2.13 shows the resulting structure.

The potential object of (16) is "*today*" which has the element +TIME in its inherent properties. Since the object slot of "*give*" specifies -TIME, "*today*" is rejected as a filler. The parser would then try any other possible interpretations of the sentence. When none are found, the sentence is rejected as being ungrammatical.

When a filler is found for a role, the inherent properties of the filler are updated from those of the slot. This is done using the `refine:` operation. This fills in any ? elements in the filler's vector with corresponding values from the slot's. The filler is then placed in the slot, replacing the selectional restriction vector.

2.8.2. Relations. A *relation* is a connection between constituent items in a sentence which is implied by complex predicates. A relation is much the same as a semantic entry. The difference is that a slot in a relation can be filled by a slot filler of the predicate implying the relation. For example, Figure 2.16 shows the relations FROM-POSS and TO-POSS which are used with verbs that result in a transfer of possession. Figure 2.17 shows how these relations are used in the semantic structure corresponding to "*George gave Martha the candy.*".

FROM-POSS		TO-POSS	
Role	Filler	Role	Filler
Object:	^Object	Object:	^Object
Beneficiary:	^Agent	Beneficiary:	^Beneficiary

FIGURE 2.16. The FROM-POSS and TO-POSS relations.

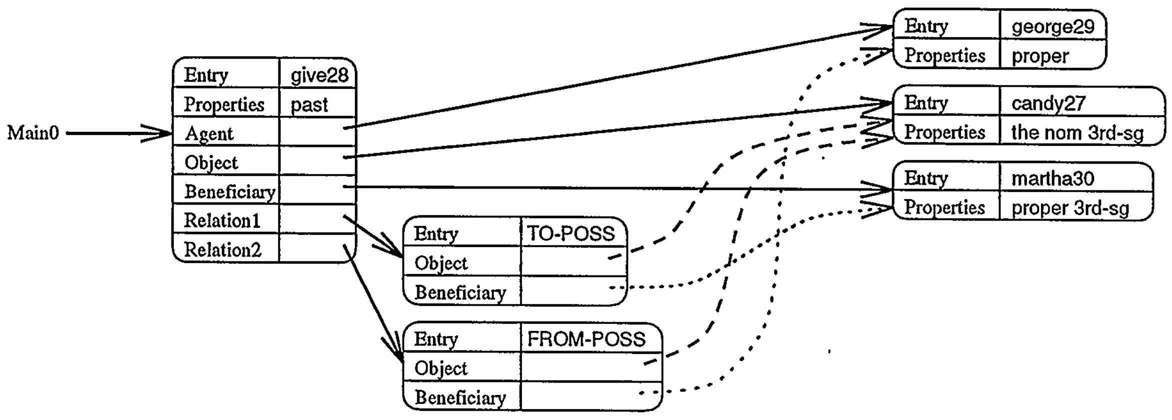


FIGURE 2.17. Usage of relations

2.8.3. Grammatical Roles. As a meaning structure is being built, entries are accessed through *grammatical roles* (sometimes referred to as gramroles). These correspond to the set of boundary registers[8]. This is done since syntactic boundaries correspond to semantic units (e.g. subject, noun phrase, etc).

The parser contains a dictionary that binds each grammatical role to a *reference*. A reference contains two pieces of information: a set of morphosyntactic properties (used for agreement) and a semantic entry. There are a fixed number of references in the system, which are allocated when the parser is created. Because of this, it is conceivable that a situation would occur in which all references have been used. When a new reference is required the one that has been least recently used is reused. This handling of reference storage is a factor in maintaining constant space complexity.

[8]This is a 1-1 and onto correspondence, so the same names are used for corresponding boundaries and grammatical roles, SUBJ for example).

2.8.4. Agreement. As discussed in §2.7.2.1 there are two types of morphosyntactic properties: agreement and non-agreement. This section is only concerned with agreement properties, since they are the ones that convey agreement information.

When morphosyntactic agreement of a grammatical role is checked, either against another grammatical role or a reading, only the agreement properties are used. They are extracted from the property sets of both items involved and intersected. If this intersection is empty the check fails. If the intersection is non-empty, it is assigned as the agreement property set of the LHS and the check succeeds. The non-agreement properties of both items remain unchanged except in one case. If agreement is being checked against the current interpretation, and it succeeds, the non-agreement properties of the reading are added to those of the grammatical role. This is done to initialize the non-agreement properties of the grammatical role, since none are assigned when the grammatical role is given a new reference.

This agreement method is directional and does not suffer from the problem of non-distinctness that recent unification-based approaches do (Blank & Labuda, 1991). Directionality allows one item to restrict another, possibly leaving the first neutral. This capability is not required so much for English, but is for other languages with a less impoverished agreement system such as German.

2.8.5. An Example. For example, the actions for three productions (`NAME`, `SUBJ`ect, `BITRANS`itive verb) are shown in Figure 2.18[9]. I will use these to illustrate how agreement, selectional restrictions, and meaning-graph building are performed in parsing "*George gave*" in sentence 15 or 16.

[9]Actions are described in detail in the next section.

When "*George*" is recognized, it enables the NAME production. When NAME is used its actions are executed. They assign the lexical entry "*George*" to the grammatical role *NP* and add properties specifying a proper noun and third person singular[10].

Next, "*George*" is recognized as the subject of the sentence by the SUBJ production. The associated actions do two things:

- Set a backtracking point in the Subj boundary register.
- Set the Subj grammatical role to refer to the same semantic entry as NP (which was just loaded with the lexical entry for "*George*").

Finally, the BITRANS production is used to process "*gave*"[11]. The actions construct the semantic entry for the main predicate of the sentence:

- Save a backtracking point in the Pred register.
- Connect the current lexical entry ("*give*") to the Pred grammatical role.
- Check agreement and selectional restrictions on the subject role, and if the checks succeed assign the previously constructed subject to the subject role of the new predicate structure.

2.9. Actions

This section describes the actions of RV. There are twelve actions, organized into five categories. Each of these categories is considered separately.

2.9.1. Embedding. Embedding actions control the level of embedding being used.

[10]By adding these properties with the production responsible for parsing names, it saves the work and storage requirements involved in specifying them for all names in the lexicon

[11]Actually, the INTRANSitive and TRANSitive productions would be tried as well. The backtracking mechanism would eliminate those that did not match the argument structure of the sentence.

NAME

`NP=lex`	*set the lexical entry for the name*
`NP addprop <proper 3rd*sg>`	*set the appropriate properties for a name*

SUBJ

`save Subj`	*set a backtracking point*
`Subj:=NP`	*the most recent noun phrase is the subject*

BITRANS

`save Pred`	*set a backtracking point*
`Pred=lex`	*set the lexical entry for the verb/predicate*
`Subj=Pred.subj`	*check agreement and selectional restrictions. If these are consistent, set the subject of the predicate*

FIGURE 2.18. Actions for the `NAME`, `SUBJ`, and `BITRANS` Productions.

shiftdown

Embed one level deeper. Fails if the parser is currently embedded two levels below the main sentence, succeeds otherwise.

returnup

Unembed one level. Fails if the parser is currently at the main sentence level, succeeds otherwise.

2.9.2. Boundary Registers. Boundary related actions manage the saving and restoring of parser state information. Much of this work is done automatically by the parser engine during its normal operation, but there are situations where it useful to do this explicitly.

save *boundary*

Save the current parser state in the boundary register named *boundary*. The parser engine saves the state in the **CURR** register after each production application[12] and in the **WORD** register after each token is read from the input. The save action is used to place explicit backtracking points. Always succeeds.

adjoin *boundary*

This is used to explicitly restore the parser state from the boundary register named *boundary*. This action is rarely used. Always succeeds.

2.9.3. Property Manipulation. There are actions to test for and modify morphosyntactic properties.

lexprop *properties*

This action tests the current input token's property set. This is done by intersecting *properties* and the token's properties. This action is very useful when a word having certain properties is required. For example, in the sentence "*I had to go.*", the verb following "*to*" has to have the property `infinitive`. Succeeds if the result is non-empty, fails otherwise.

gramrole addprop *properties*

The addprop action adds the properties in *properties* to the property set of the semantic entry associated with *gramrole*. An example of this was shown in §2.8. Always succeeds.

2.9.4. Semantics. There are several actions used for building the semantic structure from an input.

gramrole new

This assigns a new semantic entry to the grammatical role *gramrole*. Always succeeds.

[12]This is an integral part of the engine's operation.

gramrole := null

Assigns the null semantic entry to the grammatical role *gramrole*. This is used to create placeholder nodes in the semantic structure or to add nodes where there is no corresponding lexical entry in the input. For example, this action is useful when parsing adjectives. A null role is created for the expected noun. The adjectives then modify this null entry, which is eventually filled in by the head noun. Always succeeds.

gramroleL := *gramroleR*

These are three variations on a single action. All three assign the contents of the grammatical role *gramroleR* to *gramroleL*. The first form is a simple assignment. The second has *gramroleR* prefixed by "^". In this case, the contents of *gramroleR* from the next more shallow embedding level are used. The final form prefixes *gramroleR* with "\". This form uses the contents of *gramroleR* from the next deeper embedding level. Succeeds if *gramroleR* has a valid value, fails otherwise.

gramroleL = lex

gramroleL = *gramroleR[.role1[.role2]]*

These actions actually do the structure building. The first form matches the semantic contents of *gramroleL* with that of the current input token. If they are compatible, the contents of *gramrole* are refined by that of the input token. The second form operates between *gramroles*. It can optionally take the name of a role within *gramroleR*'s contents. In that case the specified role is filled by *gramroleL*. The second role specifier can be present if the first identifies a relation role. Succeeds if the action was successfull (i.e. contents were compatible), fails otherwise.

gramroleL -> lex

gramroleL -> ***gramroleR***

These are the actions that add modifiers to a semantic entry. The first form adds the current input token as a modifier of *gramroleL*[13], while the second adds *gramroleR* as a modifier. Always succeeds.

2.9.5. Agreement. The actions in this section check agreement of morphosyntactic properties. Agreement checks are performed by intersecting the agreement properties of the two arguments and setting those of the left argument to the result if it is non-empty.

gramroleL agree lex

gramroleL agree ***gramroleR***

The first form checks agreement of *gramroleL* with the current input token. The second form checks it with another grammatical role. Succeeds if the intersection of the two agreement property sets is non empty, fails otherwise.

2.10. Summary

This chapter described, in detail, the design and operation of the RV formalism. Several parsers were provided as examples. The chapter began with a formal definition of RV, then went on to describe the underlying data structures and operations on them. The discussion covered all areas of the parser's responsibility: syntax, lexicon, and semantics.

[13]The seemingly backwards arrow notation was kept to be consistent with Blank's notation.

Chapter 3

Related Work

This chapter presents work related to that described in this thesis. First two RV development systems are described. The second section briefly describes some lexical acquisition systems that have been reported.

3.1. Existing RV Development Systems

This section discusses the two known existing development systems for RV parsers: Blank's, which provides the parser developer with tools with which to construct RV parsers, but conforms to the traditional edit-compile-debug approach; and Reed's which compiles phrase structure grammars to RV productions. Finally, the ways in which RV-Tools differs will be discussed.

3.1.1. Blank's System. Glenn Blank developed the original RV development system (Blank, 1989; Blank, 1991; Blank & others, 1992). This system is text based and command line oriented. Syntax rules and lexical entries are specified using a text editor (such as Emacs). Each is contained in a separate file. Sections 3.1.1.1 and 3.1.1.2 show sample syntax and lexicon specification file, respectively.

This system is typical of the traditional compiled language paradigm: a source file is edited, compiled and then tested. If problems are detected the sequence is repeated. Figure 3.1 shows the organization of Blank's system.

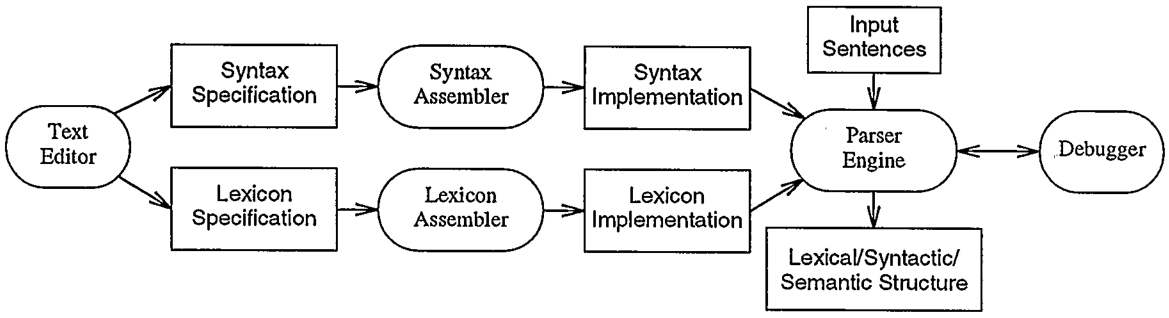


FIGURE 3.1. Data flow in Blank's Development System

Once the syntax and lexicon specification files have been prepared they are processed to create syntax and lexicon implementation files which are used by the parser. Operation of the parser is controlled by a set of command menus. The appropriate menu is printed at any time, and a selection is made by means of a keystroke. Section 3.1.1.3 shows part of a typical session, while Section 3.1.1.4 shows part of a debugging session.

3.1.1.1. *Example RV parser specification.*

```
{A fragment demonstrating affix agreement as described in agree91.ps}
morphosyntactic_properties
   <first second third> <sg pl> pres past pastpart prespart inf nom

ordering_features
   OPEN S TOP TENS MODAL AUX HAVE BE NEG V IO O THAT XO XIO BARE INF
   PPR PASS MAINC
   NP DET NUM ADJ HEAD DENOM NEND NROLE REL
   INFL PREP PNP GAP XS RELEND

macros features
  ##NPon       +DET..HEAD -DENOM..NEND  {enable/require noun phrase up to head}
  ##NPoff      -DET..REL                {disable or require noun phrase off}
  ##NPmod      -DET..HEAD +DENOM..NROLE {Past head of NP}
  ##NPend      -DET..HEAD +NEND +REL    {Before end of NP}
  ##NRole      -DET..HEAD +NROLE        {condition for SUBJ, OBJ, etc.}
  {ClauseOn initializes requirements for most clauses--note embedded #NPoff}
  ##ClauseOn   +S..O -THAT..PASS +PPR +NP -AUX -INFL..RELEND #NPoff
  ##ClauseOff  -OPEN..O -DET..GAP {disable/require constituents off}
```

```
default cond #NPoff ?DENOM ?NROLE ?REL {in clause, not phrase} -OPEN

boundaries Topic Tense Subj Pred Obj Clause NP NPmod

productions

p TENSE    N morph pres past {TENSE must observe a tensed morphological form}
             cond +TENS +NROLE -XS -PREP change -TENS +INFL -NP +PNP
             action save Tense
p TENSV    N morph pres past {TENSV fires for tensed main verbs (not auxiliary)}
             cond +TENS +NROLE -PREP     change -TENS..BE +INFL +AUX +NP

p BE       L cond +INFL +BE -TOP         change -MODAL -HAVE -BE -INFL

{Verb subcategories: INTRANS for no object, TRANS for one, BITRANS for two}
p INTRANS S cond -S +O +INFL +AUX  change -MODAL..NEG -NP -O -IO
           action save Pred  {INTRANS rules out OBJ or IOBJ}
p TRANS    S cond -S +IO +INFL +AUX change -MODAL..NEG -IO  action save Pred
p V        L cond -S +V -AUX..NEG +INFL change -V -INFL -REL -PNP

p CADJ     L cond -BE -S +V               change #ClauseOff -NP
             action save Pred

{The following productions follow NPEND (and assign grammatical roles)}
p SUBJ      N cond #NRole ?NEND +S -TENS +V   change -S #NPoff -TOP
              action save Subj
p OBJ       N cond #NRole ?NEND -V +O         change -O -NP -DENOM -NROLE
              action save Obj

{NP introduces a noun phrase (skipped by PREP and other prep productions)}
p NP        N cond +NP #NPoff ?REL -INFL -AUX -PREP -PNP  change #NPon +NROLE
              action save NP     {save at opening of phrase}
{Noun phrase productions}
p DET       L cond #NPon            change -DET  {determiner}
p NOUN      L morph nom {Must be a nominal, not a verb affix}
              cond #NPon ?DET..ADJ  change #NPmod  +REL
p NAME      L cond #NPon            change #NPmod              {George, etc.}
p PRON      L cond #NPon            change #NPoff +NROLE -PREP -AUX
              action save NPmod     {Pronouns have no post-modifiers}
p NPEND N cond -HEAD +NEND          change #NPoff -AUX ?NROLE -PREP
              action save NPmod

{Relative clauses come in several varieties, which cross-categorize:
 They may open with explicit relative pronouns: RELR or RELC are Lexicals,
   or not: RELRO or RELCO are Non-lexicals rivalling RELR and RELC
 They may right embed--RELR or RELRO--if clause is "on the table": -S..O -GAP
   or center-embed--RELC or RELCO--if clause not yet seen S,V,O or GAP
 Next, they may be full relative clauses--REL--enabling SUBJ, AUX, GAP
   or reduced relative clauses--REDREL-disabling SUBJ, AUX, GAP
}
```

```
p RELR      L cond #NPend -S..O -GAP {"clause on the table": -S..O -GAP}
              change #ClauseOn +OPEN -TOP -MAINC ?RELEND {thru from RELC} +GAP
              action save Clause {right-embed, but allow resumption of adjuncts}
p RELC      L cond #NPend +O -RELEND {clause not yet on table: +O -RELEND} -PNP
              change #ClauseOn +OPEN -TOP -MAINC +RELEND
              action shiftdown {center-embed--next clause level}
p RELRO     N cond #NPend -S..O -GAP {"clause on the table": -S..O -GAP} ?NP
              change #ClauseOn +OPEN -TOP -MAINC ?NP {thru to REL} ?RELEND +REL
              action save Clause {right-embed, but allow resumption of adjuncts}
p RELCO     N cond #NPend +O -RELEND {clause not yet on table +O -RELEND} -PNP
              change #ClauseOn +OPEN -TOP -MAINC +RELEND +REL {+REL==no SubjGap}
              action shiftdown {center-embed--next clause level}
{REDREL--reduced relative clause (no SUBJ, AUX, NGAP), e.g. "cat sleeping..."}
p REDREL    N morph pastpart prespart
              cond +OPEN -MAINC +REL
              change -OPEN..BE -NP {Disable SUBJ, AUX} -REL
{REL--full relative clause (with SUBJ & AUX), e.g. "cat [which] I pet"}
p REL       N cond +OPEN -MAINC +NP  change -OPEN +GAP {NGAP in full rel clause}
{RELEND marks end center-embedded post-modifiers: shifts up a clause level}
p RELEND N cond #ClauseOff +RELEND
            change -DET..DENOM -REL {Continue noun phrase at higher level}
            action returnup  {return from center-embedding}

{WH-questions require an NGAP in place of an NP: +GAP}
p WH      L cond +OPEN +MAINC change +GAP +XS -OPEN -NP action save Topic
{NGAP meets GAP requirement; NGAP requires an NROLE production}
p NGAP      N cond ?DET..NEND -NROLE -REL +GAP
              change #NPoff +NROLE -GAP

{Main clause opening productions}
p OPEN    N cond +OPEN +MAINC  change -OPEN  action save Topic

{Clause InitFinal production}
p CLOSE     I cond #ClauseOff -RELEND ?NEND
              change +OPEN #ClauseOn +MAINC
semactions
p DET        NP <= lex
p NOUN       NP <= lex
p NAME       NP <= lex
p TENSE      Tense <= lex
p TENSV      Tense <= lex
p SUBJ       Subj := NP Subj <= Tense
p V          Pred <= lex
p OBJ        Obj := NP
p REL        Topic := ^NP
p REDREL     Subj := ^NP
p WH         Topic <= lex
p NGAP       NP := Topic
```

3.1.1.2. *Example RV Lexicon.*

```
paradigms
m BED   s <third*pl nom>
        / <third*sg nom>

m SHEEP / <third nom>

m PULL  / <first second third*pl pres>
        s <third*sg pres>
        ing <prespart>
        ed <past pastpart>

m LOVE  e  <first second third*pl pres>
        es <third*sg pres>
        ing <prespart>
        ed <past pastpart>

m BE    are <pres pl>  is <pres third*sg>  am <pres first*sg>
        was <past first*sg third*sg>  were <past pl second*sg>
        be <inf>  been <pastpart>  being <prespart>

m A     / <third*sg>
m THE   / <third>

entries
e clock         cat NOUN        morph clock_BED_
e sheep         cat NOUN        morph sheep_SHEEP_
e tick          cat INTRANS V NOUN morph tick_PULL_ m tick_BED_
e graze         cat INTRANS V   morph graz_LOVE_
e be            cat BE          morph _BE_
e fat           cat CADJ
e hungry        cat CADJ
e a             cat DET         morph a_A_
e the           cat DET         morph the_THE_
e that          cat RELC RELR DET
e .             cat CLOSE
```

3.1.1.3. *Sample session.*

```
%rvg
Register Vector Grammar, Version 3.9.6
Copyright 1990 -- G. Blank, Lehigh University
Grammar assembled from sep92.syn
Lexicon assembled from sep92.lex
Parser,Trace,Sentence file,Grammar,Lexicon,Words,Os,Help,eXit RVG:s
Keyboard,From file:sentin.txt,Change file,Goto,Repeat,Help,eXit menu:f
Reading from sentin.txt
Parser,Trace,Sentence file,Grammar,Lexicon,Words,Os,Help,eXit RVG:p
  #We start by honoring our first first family once more:
1)George loves Martha.
```

```
Parse succeeds
S:NP:NAME:George; NPEND:TENSV:SUBJ:TRANS:love; NP:NAME:Martha; OBJ:CLOSE:.;
Main0:love0 present 3rd-sg
   Lsubj:George1 proper 3rd-sg
   Lobj:Martha2 proper 3rd-sg
Inspect,Roles,TraceMenu,SentenceMenu,Grammar,Lexicon,Os,Help,eXit,<space>:
#A famous syntactically ambiguous sentence:
2)Time flies like an arrow.
Parse succeeds
S:IMP:TRANS:time; NP:NOUN:fly; OBJ:PPR:PREP:like; INDEF:an; NOUN:arrow; C:
   PPEND:CLOSE:.;
Main0:time3 imperative present inf 1st-sg 1st-pl 2nd-sg 2nd-pl 3rd-pl
   Lobj:fly4 nom 3rd-pl
   mod:like5 pp 3rd-sg
      Lobj:arrow6 a nom 3rd-sg
Inspect,Roles,TraceMenu,SentenceMenu,Grammar,Lexicon,Os,Help,eXit,<space>:
3)Is the block sitting on the table red?
Parse succeeds
S:QUES:BE:be; NP:DEF:the; NOUN:block; SUBJ:PROG:INTRANS:sit; PPR:PREP:on;
  DEF:the; NOUN:table; PPEND:ADVNP:ADJ:red; ADJHEAD:C:ADVNPEND:CLOSE:?;
Main0:sit23 yn_quest prog present 3rd-sg
   Lsubj:block25 the nom 3rd-sg
   mod:on26 pp 3rd-sg
      Lobj:table27 the nom 3rd-sg
   mod:null33 adjunct 3rd-sg 3rd-pl
      mod:red
Inspect,Roles,TraceMenu,SentenceMenu,Grammar,Lexicon,Os,Help,eXit,<space>:
```

3.1.1.4. *Sample debugging session.*

```
%rvg
Register Vector Grammar, Version 3.9.6
Copyright 1990 -- G. Blank, Lehigh University
Grammar assembled from agree2.syn
Lexicon assembled from agree2.lex
Parser,Trace,Sentence file,Grammar,Lexicon,Words,Os,Help,eXit RVG:t
Quick,Backtracking,Step,Changes,To breakpt,No prompt,Options,Help,eXit menu:b
Parser,Trace,Sentence file,Grammar,Lexicon,Words,Os,Help,eXit RVG:t
Quick,Backtracking,Step,Changes,To breakpt,No prompt,Options,Help,eXit menu:s
Parser,Trace,Sentence file,Grammar,Lexicon,Words,Os,Help,eXit RVG:p
Sentence: the fat sheep is hungry.
OPEN:
Inspect,Roles,TraceMenu,SentenceMenu,Grammar,Lexicon,Os,Help,eXit,<space>:
OPEN:NP:
Inspect,Roles,TraceMenu,SentenceMenu,Grammar,Lexicon,Os,Help,eXit,<space>:
OPEN:NP:DET:the;
Inspect,Roles,TraceMenu,SentenceMenu,Grammar,Lexicon,Os,Help,eXit,<space>:i
)the fat sheep is hungry.
OPEN:NP:DET:the;
LexEntry: *the
```

```
 Categories: DET
 morph props:third-sg third-pl
ContPt:none
Syntactic state vector at ClauseLevel 0:
-OPEN+S+TOP+TENS+MODAL-AUX+HAVE+BE+NEG+V+IO+O-THAT-XO-XIO-BARE-INF+PPR-PASS
+MAINC+NP-DET+NUM+ADJ+HEAD-DENOM-NEND+NROLE-REL-INFL-PREP--PNP-GAP-XS-RELEND
Choose a production (or List,Registers,Help,<Enter> to quit):r
 1:SynState now 2:Curr         3:Prod         7:TopicO      13:NPO
Choose a state register NUMBER (or <enter> to quit):13
Showing NPO:
)the fat sheep is hungry.
OPEN:
LexEntry: *the
 Categories: DET
 morph props:third-sg third-pl
ContPt:none
Syntactic state vector at ClauseLevel 0:
-OPEN+S+TOP+TENS+MODAL-AUX+HAVE+BE+NEG+V+IO+O-THAT-XO-XIO-BARE-INF+PPR-PASS
+MAINC+NP-DET-NUM-ADJ-HEAD-DENOM-NEND-NROLE-REL-INFL-PREP--PNP-GAP-XS-RELEND
Choose a production (or List,Registers,Help,<Enter> to quit):NOUN
NOUN's cond differs:+HEAD
```

3.1.2. Reed's System. When RV was still in the process of maturing, Jonathan Reed developed a system that compiled phrase structure grammar rules into an RV representation by using a finite state automaton as an intermediate representation (Reed, 1987; Reed, 1989). His system used RV as an efficient engine underlying a phrase structure grammar specification. It was able to make use of much of RV's efficiency in terms of processing speed, but it generated many more syntax rules than would be necessary in a pure RV approach. Also, not all phrase structure grammars can be converted to finite state automata. Thus, the applicability of this method is limited. Specifically, finite state automata can not implement embedding so phrase structure grammars that use embedding can not be used with this method. This severely limits its use in natural language parsing systems.

To illustrate the operation of the compiler, the English auxiliary system is used as an example (Reed, 1989). The phrase structure grammar to RV compiler first converts the phrase structure grammar specification (Figure 3.2) to a finite state automaton (Figure 3.3) which is then reduced.

AUX → modal
AUX → modal not
AUX → have
AUX → have not
AUX → be
AUX → be not
AUX → modal have be
AUX → modal not have be
AUX → modal have
AUX → modal not have
AUX → modal be
AUX → modal not be
AUX → have be
AUX → have not be

FIGURE 3.2. Phrase structure grammar for the English auxiliary system

Using standard finite state automaton reduction, the resulting automaton would have the characteristic that a single state could be entered by arcs labeled with different symbols. This complicates the conversion to RV so a reduction is used that does not merge states that are entered by arcs with different labels (Figure 3.4).

This finite state automaton is then converted into a set of RV productions (Figure 3.5). The resulting productions conform to the following rules:

(1) Production names and input symbols are equivalent;
(2) Vector elements correspond to productions, in the same order as productions are listed;
(3) The condition vector of a production contains a + in the element corresponding to itself, and - in all others;

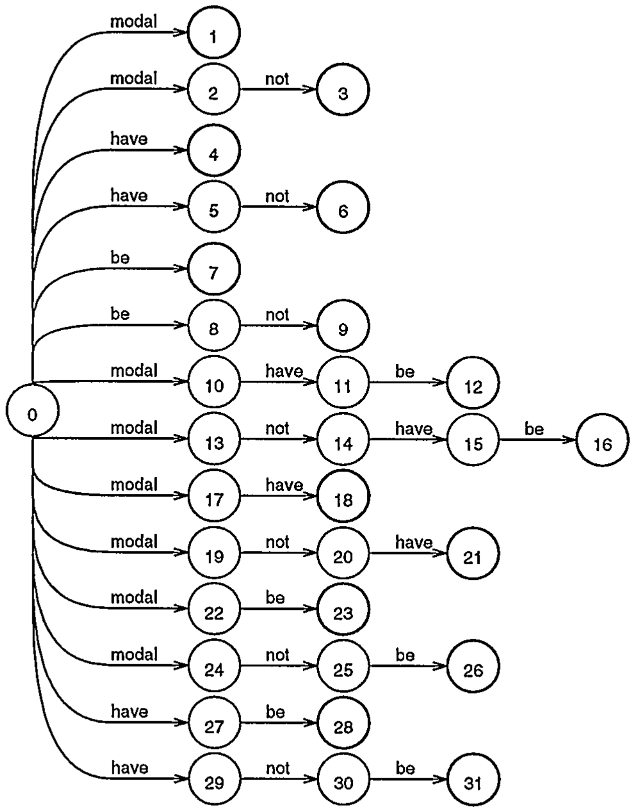


FIGURE 3.3. Initial finite state automaton for the auxiliary system.

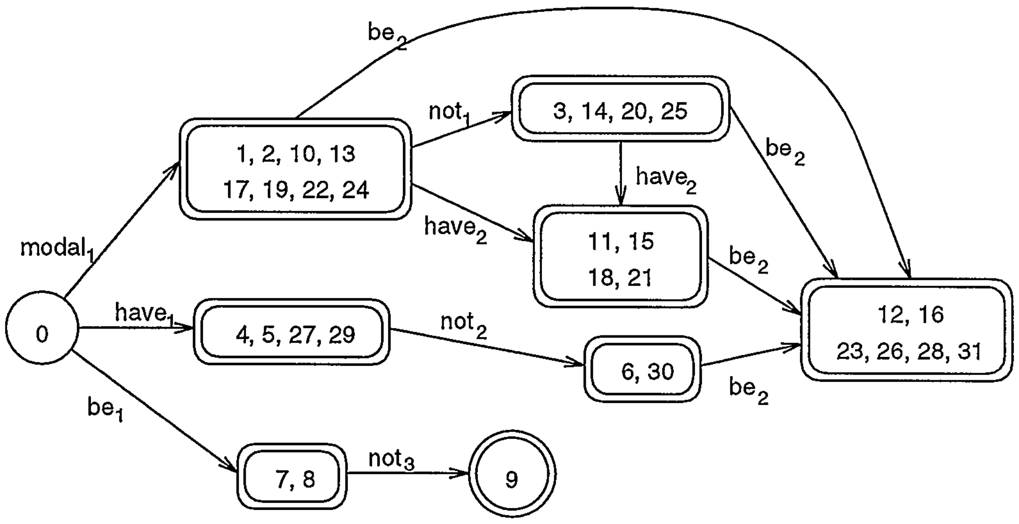


FIGURE 3.4. Reduced finite state automaton for the auxiliary system.

(4) The result vector of a production contains a + in all positions for which the finite state automaton changes state;

(5) If a production represents a final state, the elements of the result vector corresponding to "close" contains +, otherwise it contains -.

The resulting RV parser should be hand optimized to attain maximum efficiency (Figure 3.6).

Label	condition vector	change vector
init		+-+-+----
modal$_1$	+????????	-+----+++
not$_1$	?+???????	------+++
have$_1$	??+??????	---+---++
not$_2$	???+?????	-------++
be$_1$	????+????	-----+--+
not$_3$	?????+???	--------+
have$_2$	??????+??	-------++
be$_2$	???????+?	--------+
close	????????+	

FIGURE 3.5. Resulting set of RV productions.

Label	condition vector	change vector
init		+++-
modal	---?	-+++
not	???-	???-
have$_1$	+++?	--++
be$_1$	+++?	---+
have$_2$	-++?	--+-
be$_2$	-?+?	----
close	-???	

FIGURE 3.6. Hand optimized RV productions

Reed's system is interesting but does not fully address the problem of developing RV parsers. It simply uses RV as an efficient implementation system for phrase structure grammars. Also, the system generates only productions useful for syntactic analysis: there is no support for handling agreement and semantics which must be added manually to the resulting productions. These productions may not be designed appropriately, thus requiring some redesign. Finally, the lexicon has to be built completely by hand.

3.1.3. How RV-Tools Differs. RV-Tools follows the Smalltalk approach of developing in an interactive, largely modeless environment. There is no compilation step, and editing can be performed while the parser is being debugged, with changes taking effect immediately. This results in a interactive, incremental parser development environment. The organization of RV-Tools is shown in Figure 3.7.

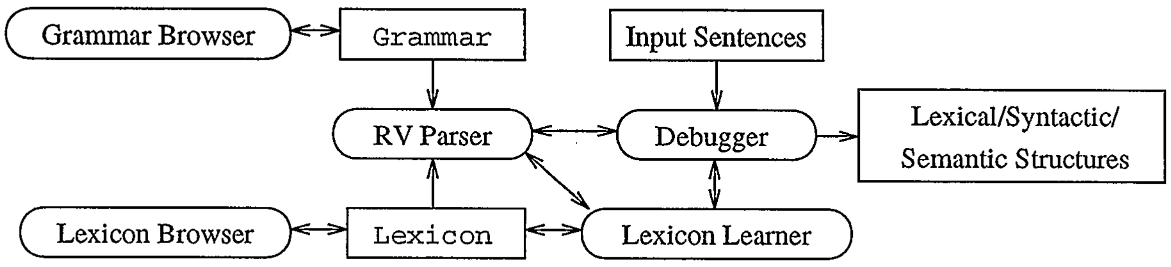


FIGURE 3.7. Diagram of RV-Tools.

3.2. Lexical Acquisition Systems

This section describes some of the notable lexical acquisition systems, and comments briefly on how they relate to the one presented in this thesis.

Harris (Harris, 1977) developed a mechanism for acquiring vocabulary by learning associations between actions/objects and words. This is the inspiration for the category pruning mechanism described in Chapter 8.

Hasting et. al. (Hastings, Lytinen & Lindsay, 1991) describe a system for learning word–concept associations within a fixed domain. All concepts involved are known ahead of time and stored in a hierarchy, along with their type and the types of any slot fillers. Slot filler types are used to incrementally induce the meanings of unknown words. This system does not learn any morphology, it just labels concepts.

Kazman (Kazman, 1991) presents a psychologically motivated model of lexical acquisition. This system has no prior knowledge of affixes. It discovers stems and affixes by comparison of different word forms. It then uses these affixes with other words of the same type. The system does use a frequency based reinforcement mechanism to prune out any unused combinations (such as *foots* and *goed*).

Paul Kogut (Kogut, 1992) worked on learning lexical semantics for RV from both text corpora, and the lexical database "Wordnet", developed at the Princeton Cognitive Science laboratory. This is a good way to learn a large, standard vocabulary for a new domain, but it does not provide a way to dynamically adapt to a specific user and domain.

3.3. Summary

This chapter described the two existing, documented systems which support development of RV parsers: Blank's and Reed's. RV-Tools' philosophical differences with these were described briefly:

- Blank's system has a traditional edit–compile–test development paradigm whereas RV-Tools provides various views of the systems under development, is interactive, and allows modification of the parser at any time;
- Reed's system was designed to use RV as an efficient mechanism for implementing phrase structure grammars, and does not address direct development of RV parsers. RV-Tools, on the other hand, is targeted at the parser developer who is working directly with the RV formalism.

Finally, related work in the area of automated lexical acquisition was briefly described. Most of the systems presented strive to be psychologically valid, while those that are primarily pragmatic are generally not interactive or incremental. This contrasts to the vocabulary acquisition mechanism of RV-Tools, described in Chapter 8, which is designed for interactive, incremental operation.

Chapter 4

Brief Introduction to Smalltalk-80

This chapter provides a brief overview of object-oriented programming concepts as they apply to Smalltalk-80. It also introduces the concepts involved in the Model-View-Controller paradigm used by the Smalltalk-80 system.

The concepts and terminology introduced here provide background for later chapters which deal with RV-Tools implementation issues.

4.1. An Introductory Example

Before any details about Smalltalk-80 are described, I present a simple illustrative example.

The example presented is that of the turtle in a turtle graphics environment, such as LOGO. A turtle in a graphics system has the capability of moving in a straight line and turning any angle, all while dragging along a pen that can either be touching the surface the turtle is traversing or held away from it. The turtle responds to a small set of commands: move forward a specified amount, turn clockwise by a specified number of degrees, lower the pen so that it touches the surface, and raise the pen away from the surface.

Using an object-oriented approach we would describe the turtle as shown in Table 4.1. Figure 4.1 shows how the turtle responds to a sequence of instructions.

Attributes	Actions
location	move *distance*
direction	turn *angle*
pen status	pen *up/down*

TABLE 4.1. Description of Turtle

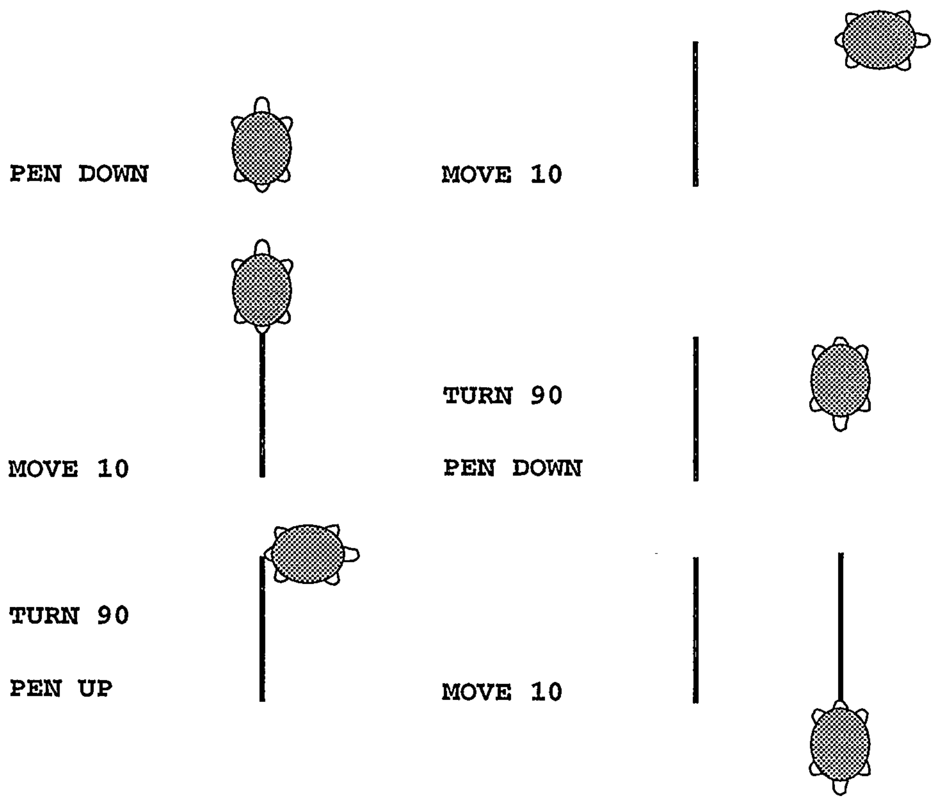


FIGURE 4.1. Response of Turtle to Instructions

The simple turtle that is described here uses several object-oriented principles. The main point is that the programming model of the turtle is a self-contained entity. It has certain attributes that are not directly accessible from outside, and it understands and responds to a set of commands. To illustrate other properties of object-oriented programming, more turtles are added to the system. One carries a selection of different colored pens. This new turtle would behave the same as the original one with an additional attribute indicating the color of the pen being dragged, and another action for selecting a specific pen. Yet another turtle could respond to the turn instruction by interpreting the number as an angle measured in radians rather than degrees. These three turtles would be related by inheritance. Each of the new turtles would inherit attributes and actions from the original, while possessing additional capabilities, and possibly modifications to some of the inherited actions. Such a relationship can be represented by an *inheritance hierarchy* such as in Figure 4.2.

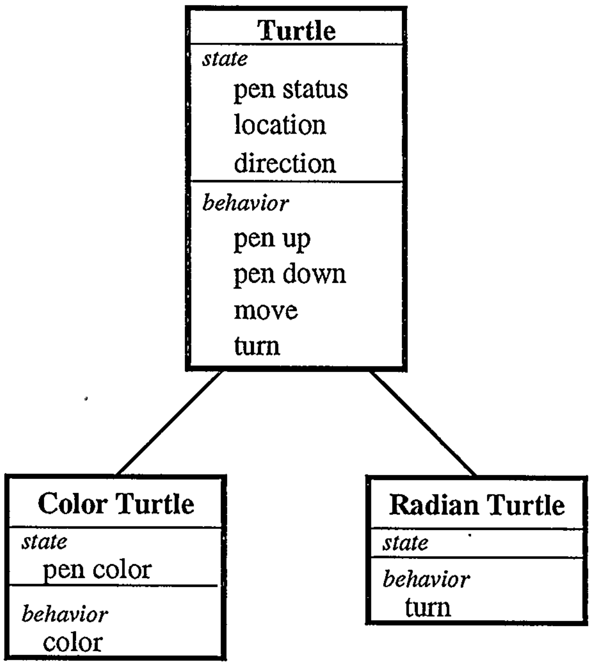


FIGURE 4.2. Inheritance Hierarchy of Turtles

4.2. Object Oriented Programming with Smalltalk-80

This section briefly describes some of the concepts central to object-oriented programming in Smalltalk-80. To do so, I will show how the turtle in the previous section could be implemented.

4.2.1. Classes. The class is the unit of encapsulation in Smalltalk. A class defines the state and behavior of all objects that are instantiations of that class. To implement the turtle example, three classes would be needed: `Turtle`, `ColorTurtle`, and `RadianTurtle`. Figure 4.3 shows the class declarations for these.

```
View subclass: #Turtle
    instanceVariableNames: 'position penStatus direction '

ColorTurtle subclass: #Turtle
    instanceVariableNames: 'color '

RadianTurtle subclass: #Turtle
    instanceVariableNames: ''
```

FIGURE 4.3. Turtle class declarations.

4.2.2. Instance Variables. A class's state is defined by its instance variables. Each member of the class will have a private copy of these variables. Again, see Figure 4.3 for the definition of the turtle classes' instance variables on the lines starting with `instanceVariableNames`.

4.2.3. Methods. A class's behavior is defined by its methods. Each method defines a single behavior. The behaviors of `Turtle`, `ColorTurtle`, and `RadianTurtle` are defined by the methods shown in Figures 4.4, 4.5, and 4.6, respectively.

```
move: distance
    "Move the turtle distance units"

    | newPosition |
    newPosition := position + ((distance * direction sin)
                               @(distance * direction cos)).
    penStatus == #down
        ifTrue: [self graphicsContext displayLineFrom: position
                                      to: newPosition].
    position := newPosition

turn: degrees
    "Turn the turtle degrees degrees clockwise"

    direction := direction + degrees degreesToRadians

penDown
    "Start drawing"

    penStatus := #down

penUp
    "Stop drawing"

    penStatus := #up
```

FIGURE 4.4. Turtle class method definitions.

```
color: inkColor
    "Set the drawing color"

    color := inkColor.
    self graphicsContext color: inkColor
```

FIGURE 4.5. ColorTurtle method definitions.

```
turn: radians
    "Turn the turtle radians radians clockwise"

    direction := direction + radians
```

FIGURE 4.6. RadianTurtle method definitions.

4.2.4. Objects. Objects are the things that actually do most of the work in an object-oriented system. Objects are *instances* of classes, i.e. a concrete embodiment of the specifications defined by a class. In Smalltalk, classes are themselves objects (instances of the meta-class), and so classes can have state and behavior as well.

To illustrate the use of objects, consider a group of turtles working together as shown in Figure 4.7. Each turtle is an independent object, an instance of the `Turtle` class. The code to produce the operation shown in Figure 4.7 is shown in Figure 4.8.

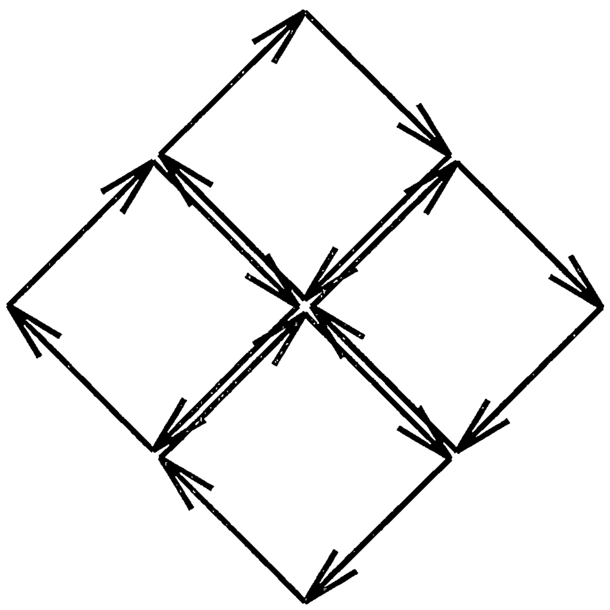


FIGURE 4.7. Cooperating turtles.

4.3. The Model-View-Controller Paradigm

The application framework paradigm that Smalltalk-80 uses is called Model-View-Controller. In this paradigm, applications are composed of three pieces: models, views, and controllers. Models encapsulate the data processing parts of applications,

```
| turtles |

"set up an array of four turtles"
turtles := Array with: Turtle new
                 with: Turtle new
                 with: Turtle new
                 with: Turtle new.

"set the initial orientations"
turtles inject: 45 into:
    [:angle :t |
    t turn: angle;
      penDown.
    angle + 90].

"move each turtle to draw the design"
4 timesRepeat:
    [turtles do:
        [:t |
        t move: 10;
          turn: 90]]
```

FIGURE 4.8. Code for cooperating turtles.

views are responsible for presenting the data to the user, and controllers handle user interaction. This division of labor is illustrated in Figure 4.9.

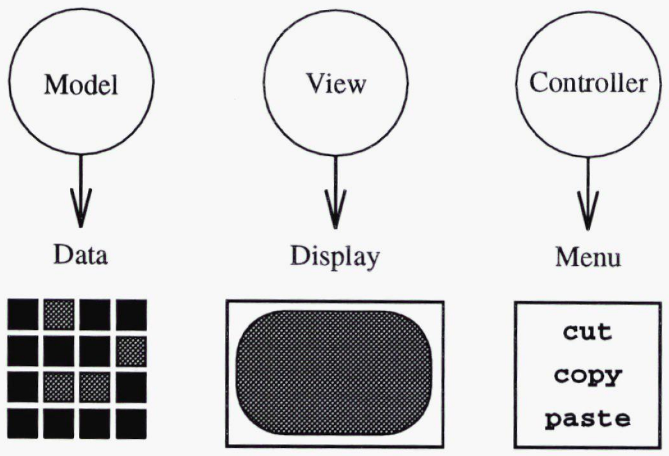


FIGURE 4.9. Models, Views, and Controllers.

Views and Controllers are generally paired: a specific view would visually represent a list and a specific controller would work in conjunction with it allow the user to operate on the list. A feature of this paradigm is that it allows a model to have several possible view–controller pairs. This is shown in Figure 4.10. In this example, the data in the sales history model can be represented as a bar graph, a textual list, or both. Different views can presents the same aspect of the model in different ways, or they can present different aspect of the model.

Another facet of this paradigm is the use of dependents to propagate change. Each model has a list of views that represent its various aspects. This structure is shown in Figure 4.11. Whenever the model's data changes, the model informs its dependent views of the change. Views that are responsible for representing the data that has changed update their representation accordingly.

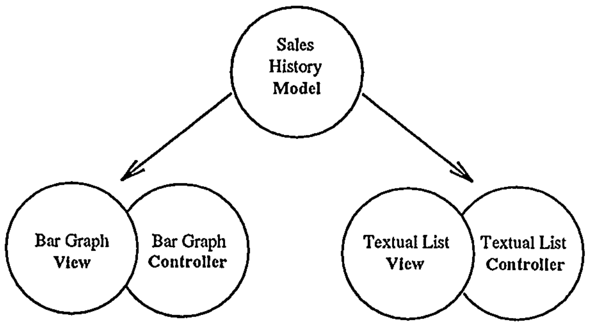


FIGURE 4.10. A Model can have multiple Views

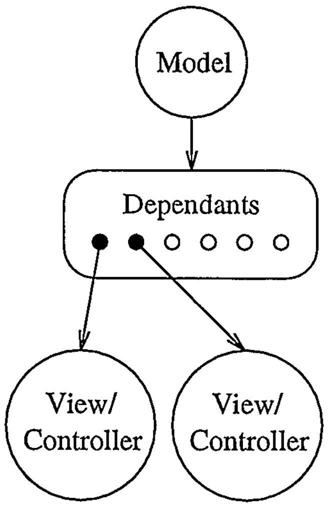


FIGURE 4.11. Model–View dependency structure.

4.4. Summary

This chapter introduced some basic concepts of object-oriented programming as it applies to Smalltalk-80. This is required as background to the implementation sections of this thesis. An object-oriented system is made up of objects which have state and behavior. This state and behaviour is defined by the object's class. Classes can be defined to inherit from another. This allows a new class to specialize an existing class, without having to replicate all of the information. This chapter also introduced the Model-View-Controller paradigm that forms the basis of the RV development environment. This paradigm provides the ability of separating data from its representation.

CHAPTER 5

Extensions

I have extended RV in several ways in support of the development tools and lexical acquisition mechanism, or to increase efficiency. This chapter describes these extensions and the reasons for them.

5.1. The diff: Vector Operation

A fourth element value has been added to the definition of ternary vectors. They are still referred to as *ternary* vectors, both for consistency and because of the semantics of the new element value. The new value is denoted by '`@`' and is functionally identical to '?' as far as the **match:**, **change:**, and **refine:** operations are concerned. The addition of the '`@`' value supports a convenience feature in the editing of vectors. A default vector can be defined (for both condition and change vectors). When the user defines a vector, that definition is combined with the appropriate default. When a vector definition is to be displayed the user expects to see the definition that they previously specified, so the default must be removed. This is done by a new vector operation called **diff:**, which is defined as:

$$D \leftarrow A \text{ diff: } B = (\forall i = 1 \ldots n),\ d_i \leftarrow \begin{cases} ? & \text{if } a_i = b_i \\ @ & \text{if } b_i = ? \wedge a_i \neq ? \\ b_i & \text{otherwise} \end{cases}$$

The '`@`' value in the output of **diff:** specifies that the element in B was '?', whereas '?' indicates that the values of A and B were the same. This is used to allow the user to override elements in the default vectors with a '?' value. For example,

$$(++-?+) \text{ diff: } (?-+?+) \rightarrow (@-+??).$$

5.2. Lexical Acquisition

An experimental lexical acquisition mechanism has been added to RV, and is fully described in Chapter 8. In this section, I will describe the changes to the RV definition that were required to support the acquisition mechanism.

5.2.1. Handling of Unknown Words. In the original description of RV encountering an unknown word is an error condition (see Figures 2.3 and 2.4). This has been changed so that encountering an unknown word invokes the acquisition mechanism to try and learn the spelling and categorization of the word. The unknown word consist of characters from the current position in the input to the next punctuation or space character. The character sequence is extracted from the input and passed to the acquisition algorithm. This algorithm will be discussed in Chapter 8.

5.2.2. Paradigm–Set Nodes. As described in §2.7 the lexicon originally has two types of internal node. A third is added to support the learner: the paradigm-set node. This type of node represents a set of possible paradigms, rather than a single one. The paradigms in the set are those that contain a string for each suffix matched by the node. The set of suffixes matched by the node are also maintained. Since the learner only uses suffixes, paradigm-set nodes are used only as immediate parents of leaf nodes (i.e. lexical entries).

Paradigm–set nodes contain two pieces of information: a set of strings that the node has been used to account for, and a set of productions that account for those strings. The internal structure is shown in Figure 5.1.

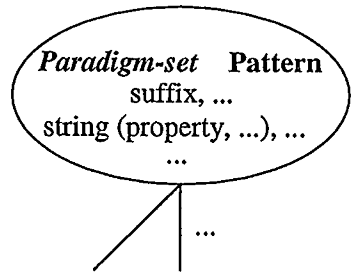


FIGURE 5.1. Internal Structure of the new Paradigm-Set node.

The addition of a new type of node in the lexical trie requires an extension of the LOOKUP algorithm of Figure 2.10. The extended algorithm is shown in Figure 8.5 and will be discussed in §8.2 since the changes constitute part of the acquisition mechanism.

5.2.3. Aging Mechanism. The lexicon acquisition mechanism tries to learn the categories of the lexical entry being acquired. It almost always assigns an overly general set of categories. To cope with this, a category aging mechanism has been added to the system. This mechanism keeps track of how often a category has led to the entry being used in a successful parse. Categories that prove to be highly useful (are frequently found in a successful parse) are made persistent, while those that prove otherwise are eventually removed.

5.3. Production Selection

In order to increase the efficiency of the parser, the production selection algorithm has been extended. The new algorithm is shown in Figure 5.2.

In addition to the previously defined tests (see §2.1.1) there is an additional one. Either the previously used production was not semi-lexical, or the production being considered fits into the active subcategorization framework. If the current production

```
FIND-PRODUCTION

"look for a lexically constrained production"

while the current interpretation is valid
  let intrep be the current interpretation
  for each production prod ∈ interp.lexical-entry.categories
    if (prod.condition match: currentState) and each of prod.actions
        can be executed in the context of interp
      return prod
    else
      advance to the next interpretation
|     if the most recently used production was semi-lexical
|       return nil

"look for a non-lexical production"

for each non-lexical production, nonlex
  for each intrep ∈ interpretations
    if (nonlex.condition match: currentState) and each of nonlex.actions
      can be executed in the context of interp
      return nonlex
return nil
```

FIGURE 5.2. Production Selection Algorithm

is semi-lexical this means that the intersection between the set of productions it can subcategorize and the active subcategorization set is non-empty. If the current production is lexical, it must be a member of the active subcategorization set. Non-lexical productions are not considered since the fact that the previous production was semi-lexical means that non-lexical productions won't be searched.

When a semi-lexical production is used, the active subcategorization set is intersected by that of the production. The active set is reset to include all productions when a lexical production is used.

In support of this change, semi-lexical productions now store the set of lexical productions that they can be used with. For example, the **TRANSITIVE** production can subcategorize the **VERB** production.

The benefit of this change is that it limits the number of dead-end branches in the search for successful parsers and the use of irrelevant productions. An example of irrelevant productions would be using **TRANSITIVE** and **NOUN** together, which could occur with a word that can be either a transitive verb or a noun such as *"call"*.

5.4. Summary

This chapter has described improvements and extensions to the RV formalism as described in Chapter 2. These include:

- the **diff:** vector operation and the '@' element value;
- a lexical acquisition mechanism and support for it, both the parser's handling of unknown words and the addition of a new type of node for the lexical trie; and
- an extension of the production selection algorithm to use information about which lexical productions are associated with each semi-lexical production,

in order to reduce the number of impossible production sequences that are considered.

CHAPTER 6

Parser and Lexicon Implementation

This chapter details my implementation of the RV parser and lexicon that was described in Chapter 2, including extensions to that description. The lexicon will be described first, as understanding the parser depends on understanding the lexicon.

6.1. The Lexicon

The lexicon involves only the LOOKUP algorithm detailed in Figure 2.10. The data structure that comprise the lexicon are those to the right of (and including) the *lexical trie* in Figure 2.2. The only complex structure is the lexical entry which is described next.

6.1.1. Lexical Entries. Each set of semantically related words (for example, the verb and noun "*call*") is represented by a *lexical entry*, which is made up of several pieces of information:

- a name — a label used to identify the entry;
- a list of lexical and semi-lexical productions — the productions that will be examined when the lexical entry is recognized in the input stream;
- a collection of wordpath(s) — nodes in the lexical trie (See §2.7.3); (These are the ends of paths through the trie that constitute a valid spelling for the entry.)

- an set of semantic roles — defines the semantics of the entry; (See §2.8)
- the entry category — for subdividing the entries in the lexicon browser (See §7.5);
- a comment — describes the entry;
- a reference to an RV parser — the parser of which the entry is part.

The last three items from the above list serve no inherently functional purpose, they are either for development support (category and comment) or are an implementation detail (associated parser).

6.2. Agreement and Semantics

Section 2.8 described the design of the semantic and agreement mechanisms. In this section we look at some of the implementation details of various aspects of this subsystem.

6.2.1. Grammatical Roles. In addition to specifying boundary register names, the list of boundaries is used to generate a list of grammatical roles, often called *gramroles*. Grammatical Roles are used for morphosyntactic agreement and semantic processing. As described in §2.8.3 references are managed using a least recently used policy. The class `ReferenceQueue` is used to implement this policy. Figure 6.1 shows the structure used by `ReferenceQueue` to manage references. When a new reference is required it is taken from the least-recently-used end of the list. Whenever a reference is used it is moved to the most-recently-used end of the list. The use of a doubly linked list increases the time efficiency of reference use.

6.2.2. Semantic Information. The atomic pieces of semantic information in this system are stored in *semantic roles*. In addition to the semantic information, a

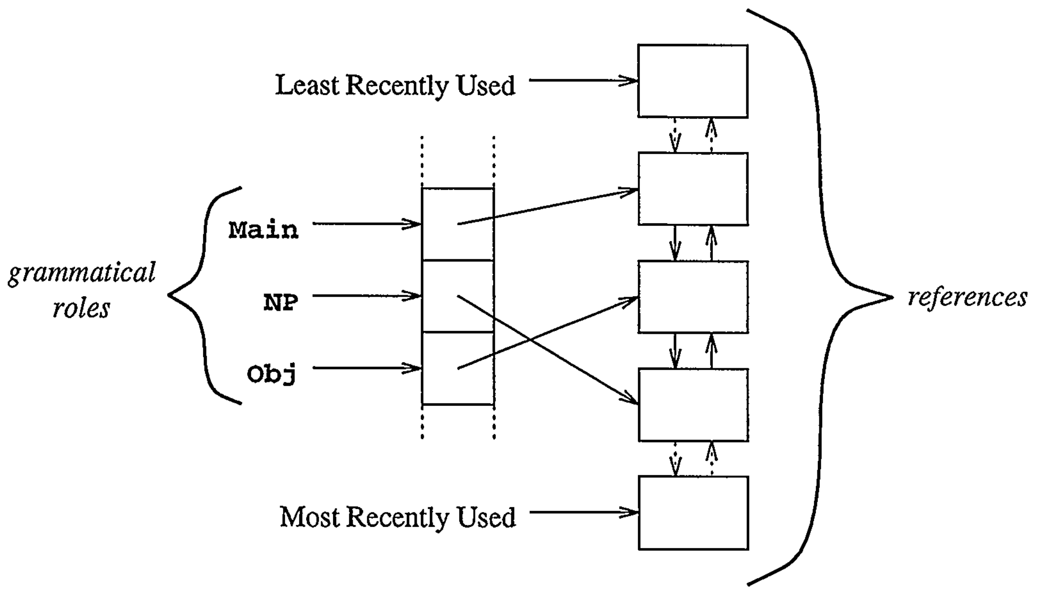


FIGURE 6.1. Reference Storage Structure

semantic role contains an ordered collection to store a binding array used to support relations. (see §6.2.4). The information stored in a semantic role can be one of:

Symbol: the name of a relation;

Semantic Entry: a semantic structure;

Lexical Entry: a reference to an entry in the lexicon (this is often the case in modifying roles);

Ternary Vector: a semantic feature vector, used to implement selectional restrictions.

Semantic roles are stored in dictionaries that organize the information by storing each semantic role under a role name.

6.2.3. Semantic Entries. *Semantic entries* encapsulate atomic pieces of complex meaning. Their main component is a semantic role dictionary. Another important piece of information is a reference to the lexical entry associated with this semantic entry. The final piece of information is the index of the reference that refers to this semantic entry.

As was discussed earlier, morphosyntactic properties are stored in references, which are dynamic in that they can be modified by backtracking. This means that the properties stored in the references associated with the semantic entries in the structure built by the parser must be retrieved whenever an interpretation is found. For this purpose semantic entries also contain a property set. When an interpretation is found, this property set is assigned the properties of the associated reference. Also at that time, a copy of the semantic structure rooted in the grammatical role `Main1` is made. The parser collects all copies of all interpretations and returns them along with the associated productions traces.

6.2.4. Relations. Semantic relations are specializations of semantic entries. One difference is that they have no associated lexical entry. In addition, role fillers can be references to roles in the semantic entry that references an instance of the relation. These roles in the relation instance are updated to reflect the value of the associated role in the semantic entry.

When the semantic information of the lexical entry is used by an agreement check, the contents of the reference are copied to a new semantic entry which is used in place of the relation to fill the associated role.

Furthermore, the role fillers in a relation can have only two types of values:

`Symbol`: a reference to a role of the enclosing predicate;

`TernaryVector`: a semantic feature vector.

Role	Initial Contents
inh	-ANIMATE..FEELING +ACTION..POSSESSION
obj	-ANIMATE..HUMAN -STATE..TRANSFER
dat	+ANIMATE..HUMAN -STATE..LOCATION -SUPPORTED
rel1	$FROM-POSS-TEMP
rel2	$TO-POSS-TEMP

FIGURE 6.2. Semantic Information of "*borrow*"

FROM-POSS-TEMP		TO-POSS-TEMP	
Role	Filler	Role	Filler
inh	-ANIMATE..FEELING +ACTION ..POSSESSION -DEST -PERM	inh	-ANIMATE..FEELING +ACTION ..POSSESSION +DEST -PERM
obj	^obj	obj	^obj
dat	^subj	dat	^dat

FIGURE 6.3. Relations in "*borrow*"

When a role filler of a relation is a symbol, it is taken as a role name referring to a role in the lexical entry (and subsequently the semantic entry) that refers to the relation. When the lexical entry definition is processed, an entry is added to the binding arrays of all roles which are referenced by any relations used by the lexical entry. Binding array entries are ordered pairs consisting of the role name that references the relation and the role within the relation.

As an example, consider the semantic information of the verb *"borrow"*, shown in Figure 6.2, and the two relations referred to, shown in Figure 6.3. The role `obj` will have two pairs in its binding array, while `subj` and `dat` each have one. The resulting binding arrays are shown in Figure 6.4. Semantic role values (other than the relations) are omitted for clarity.

When a role with a non-empty binding array has a filler assigned, the binding array is traversed. Each pair in the binding array is used to bind the new filler to a role (the

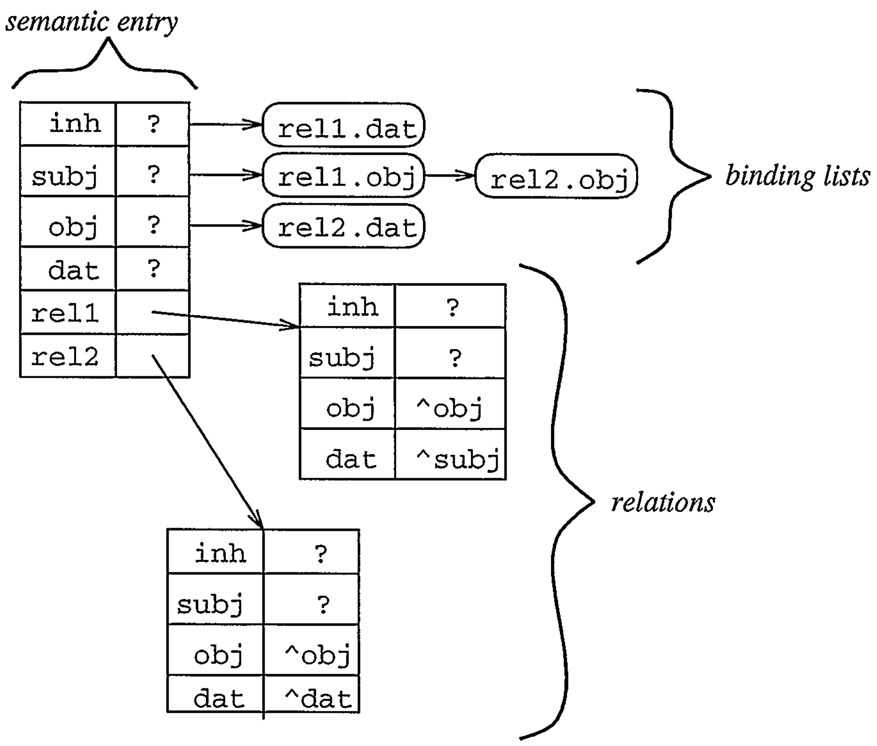


FIGURE 6.4. Binding Arrays

second element of the pair) within an outer role (the first element) of the semantic entry being updated. In this way the role fillers of relations are kept consistent with the semantic entry's roles which the relation refers to. The FILL-SLOT algorithm is Figure 6.5 shows how this is done.

```
FILL-SLOT (entry, slot, filler)

entry.slot.value is assigned filler
for each binding in entry.slot.bindings
   (entry.(binding.reference).value).(binding.role).value is assigned filler
```

FIGURE 6.5. Algorithm FILL-SLOT

6.3. The Parser

Having described the implementation of the lexical and the agreement/semantic aspects of the system, I now present the details of the parser implementation.

The algorithms that make up the parser engine have already been shown, in Figures 2.3, 2.4, and 2.5. The main data structures have been outlined already in Chapter 2 (specifically, see Figure 2.2).

6.3.1. Productions.

6.3.1.1. *Ternary Vectors.* The class `TernaryVector` is central to the parsing mechanism. Vector contents are implemented by two parallel bit vectors, called bit0 and bit1. A single element consists of one bit from each of these vectors. The two bits are combined to make a two bit code which corresponds to a ternary element, according to Table 6.1. The bit from bit1 is the MSB, and that from bit0 is the LSB.

This representation allows the ternary vector operations defined in §2.1 to be efficiently implemented using parallel bit vector operations: AND, OR, and XOR.

Code	Element
00	?
01	+
10	-
11	@

TABLE 6.1. Ternary Element Codes

6.3.1.2. *Actions.* There are five general types of actions, based on the number of arguments each takes: none, one, two, three, or four. These action types are implemented as hierarchically related abstract base classes rooted by Action (actions taking no arguments). Each subclass adds support for an additional action: `Action` is the superclass of `OneArgAction`, which is the superclass of `TwoArgAction`, and so on. Most of these abstract classes have subclasses representing specific actions. For example, `SaveAction` is a subclass of `OneArgAction`. The action hierarchy is shown in Figure 6.6.

6.3.2. Boundary Registers.

The main structure involved in the backtracking mechanism is the *boundary register*. In addition to an identifying name and a reference to a parser, boundary registers are made up of the following:

- A string containing the linear trace of the parse i.e. the history of production applications and input consumption (primarily for debugging purposes);
- A ternary vector that is a copy of the parser state;
- The embedding depth;
- The production being considered;
- The most recent production used for the current input token;
- The set of lexical productions that are being subcategorized; (This is used by the production selection mechanism as described in § 2.1.1.)

```
Action
    ReturnUpAction
    ShiftDownAction
    OneArgAction
        AdjoinAction
        AssignNewAction
        AssignNullAction
        LexAgreeAction
        LexPropAction
        ModLexAction
        MorphAgreeAction
        SaveAction
        TwoArgAction
            AddPropAction
            AssignAction
                AssignHigherAction
                AssignLowerAction
            GramAgreeAction
            ModAction
            SemAgreeAction
            ThreeArgAction
                FourArgAction
                    SemRoleAgreeAction
```

FIGURE 6.6. Action class hierarchy

- The set of production names (categories) that have already been used for this boundary register; (Keeping track of this prevents the reuse of productions. This avoids loops, both immediate and after backtracking.)
- The list of *Readings* (details are presented in the next section) containing all the possible interpretations of the input;
- The index, into list of readings, of the interpretation under consideration;
- The position in the input stream;
- A copy of the reference queue (see Figure 6.1);
- A mapping of grammatical roles to reference indexes (see Figure 6.1).

The parser state is saved in a boundary register, either by the parser engine itself for the boundaries `Curr` and `Word`, or by a `save` action for all other boundaries. When the state is saved, the current embedding level is appended to the name of the boundary, except in the case of the pseudo-boundaries `Curr` and `Word`. For example, saving in register `Clause` at level 2 results in the register name `Clause2`. This name is then pushed onto the resume stack. When the parser backtracks, a boundary name is popped from resume and the state is restored from that boundary register.

The resume stack is a stack that does not allow duplicates. When an item is pushed onto it, any duplicate on the stack is first removed. The reason for this is the central goal of achieving linear time complexity. Boundary registers are reused, overwriting any previous contents when the parser state is saved in them. This invalidates any previous use of the register, requiring it to be removed from the resume stack. While this may seem to be throwing away valuable information, it models the abilities of a native speaker. As an example, consider the sentence:

(17) The horse raced past the barn fell.

The reason that it is difficult to understand is that *"raced"* is used as an passive post-modifier of horse (meaning *the horse that was raced*), but the initial interpretation is generally as a verb, thus the MAIN-PREDICATE boundary is used. When *"fell"* is later encountered the MAIN-PREDICATE boundary is reused, losing the previous value. Thus it is difficult for a person to backtrack in order to reinterpret *"raced"*, and impossible for RV.

6.3.2.1. *Lexical Interpretations.* When the lexical analyzer finds an interpretation for the upcoming input it creates an interpretation record, called a *reading*, to represent it. All such readings are stored in a collection, sorted in descending order by the number of input characters it accounts for. This places interpretations that consume more input toward the first of the collection, giving them higher priority (by virtue of being considered first). This gives the parser its preference for idiomatic, rather than literal, interpretations.

Readings are comprised of three pieces of information:

(1) The entry in the lexicon that accounts for the input;
(2) The position in the input stream following that which is accounted for by the reading; (This is directly related to the number of characters accounted for, since all simultaneous readings start at the same position in the input. Thus, sorting is actually done on this value.)
(3) The set of morphosyntactic properties returned by the lexical trie search.

6.4. Summary

This chapter described my implementation of the RV parser engine, including the lexical and semantic components. The implementation of a parser engine was the necessary first step in developing an RV development environment.

CHAPTER 7

Development Tool Design

This chapter describes the design of the RV development tools. The first section outlines the design requirements, while the remainder of the chapter describes the design of the tools including how the requirements were met.

7.1. Design Requirements

A fundamental requirement of the development environment is that it be interactive and promote incremental parser development. The developer should be able to use any of the tools at any time, i.e. it should be a modeless environment. It follows from this that the editing tools must be available while a parser is being tested and changes made at that time should take effect immediately. This ability allows the developer to gradually refine a parser, without the discontinuity that a non-interactive environment imposes (e.g. waiting for a lengthy compile and changing tools constantly).

An important requirement is that parsers be convenient to edit. To accomplish this, a variety of structured editors are required, one for each structure in the system:

- productions;
- relations;
- paradigms; and
- lexical entries.

When one of these is to be created, a template is provided for the user to fill in. When a definition is to be edited, the current definition is placed in the template and presented to the user. As an example, Figures 7.1 and 7.2 show production creation and editing, respectively.

Name:	<*production name*>
Comment:	<*comment text*>
Type:	<*lexical type*>
Cond:	<*condition vector*>
Change:	<*change vector*>
Actions:	<*actions*>

FIGURE 7.1. Production creation.

```
Name:      <COMPAR>
Comment:   <>
Type:      <L>
Cond:      <?DET..NUM +ADJ..HEAD -NN -REL>
Change:    <-DET..NUM>
Actions:   <
           lexprop <compar>
           Word new
           Word = lex
           Word agree lex
           NP – > Word
           >
```

FIGURE 7.2. Production editing.

There are various identifiers that the user has to enter in the course of defining a structure (e.g. a production). Many of these refer to other items in the system and must be correctly spelled. This introduces a source of error into the development process. However, these critical identifiers always refer to one of the following items:

- a boundary;

- an ordering feature;
- a production;
- a semantic feature;
- a relation;
- a morphosyntactic property; or
- a paradigm.

All of these items are present in a list that is part of the parser definition. A requirement of the development environment design is that it automatically check the validity of these identifiers when the developer uses them. If one is invalid, the system must present the developer with a list of reasonable alternatives. If a correction is selected, the system should automatically make the correction in the definition being edited.

In the case of an error in the specification of an action, the system should identify the incorrect action and allow the developer to correct the error.

There are several sets of identifiers that are used in various places throughout an RV parser, boundary names for example. The parser editing tools should automatically update all uses of such an identifier when one is changed. For example, when a boundary is renamed, all productions that reference that boundary should be updated. Furthermore, when the developer asks to remove a boundary (or other such identifier) the system should check to see if it is being used anywhere in the parser. If so, the developer should not be allowed to remove it. References to it can be removed by using the appropriate cross-reference browsers (in this case, the "productions referencing a specified boundary" browser.

Specifying an RV parser involves many low-level details, specifically feature vectors (both ordering and semantic) and morphosyntactic property sets. To ease the work involved in this for the developer, a design requirement is that shorthand notations be available for doing these specifications. For feature vectors, this takes the form of macros, including a default condition and change vector for productions. For property sets, support for using simplest descriptions is required.

Another requirement is that the developer be able to view the parser from various viewpoints and freely cross-reference information[1]. To support this, several browsers will be needed to allow the user to access various facets of a parser:

- overall syntactic information;
- productions that reference a specified boundary;
- productions that reference a specified morphosyntactic property in `lexprop` or `addprop` actions;
- productions that categorize a specified lexical entry;
- non-lexical productions and their ordering;
- overall lexical information;
- lexical entries that are categorized by a specified production;
- lexical entries that reference a specified relation;
- lexical entries that reference a specified paradigm in their wordpath(es); and
- paradigms that reference a specified property.

It is sometimes useful to have the ability to examine an item in isolation. For example, if the developer wishes to compare two production definitions it would be convenient to be able to open a browser containing only a single production. This leads to another design requirement, specifically to provide browsers that operate at the following levels of granularity:

- overall syntactic information;

[1]Appendix F details how the cross-referencing operates and how the various browsers are invoked.

- single productions;
- overall lexical information;
- single relation;
- single paradigm; and
- single lexical entry.

Being able to conveniently specify a parser is of little value if it is awkward to test. Thus, another requirement is that parsers be convenient to test and debug. This requirement implies several more:

- There must be a facility for testing the parser with a testbed file of sentences, and a way of easily stepping through it;
- There must be a way to step through a parse, on a production by production basis;
- The developer must have access to the internal structures of the parser, during the parse;
- The debugger must provide useful information regarding the operations the parser is performing. The different types of information should be individually suppressible to allow the developer to concentrate on the information that is relevant. This information includes:
  - what productions are used, and when;
  - what productions are being considered for use each time a production is to be selected;
  - the reason that productions are not selected;
  - details of each action that is executed, and whether it succeeds or fails;

- an indication of when the parser backtracks, and what boundary register is involved; (There should be an option to suppress this information when the `Curr` register is being used, as this is so frequent.)
- a report of the current linear parser trace after each production application; (There should be an option of having backtracking points marked in the linear trace.)
- a report of the semantic structure representing the input's meaning, after each production application;
- a trace of the LOOKUP algorithm as it searches the lexical trie;
- reporting of any unrecognizable input, when the acquisition mechanism is disabled; and
- a trace of the lexical acquisition mechanism's operation, when it is enabled.

• There should be a convenient way to set the acquisition mechanism's operational parameters (e.g. the score at which a category become persistent, see Figure 8.4).

7.2. Background

The tools are designed using the Smalltalk-80 Model-View-Controller (MVC) paradigm (Systems, 1990; Goldberg, 1990). In this paradigm the model is responsible for all data storage and manipulation. The view handles the visual presentation of the data. Finally, the controller handles user input. Chapter 4 describes MVC in more detail.

The browser windows are subdivided into several component views, most of which contain lists. When a item in a list is selected, a message[2] is sent to the associated model[3]. Likewise whenever a view needs to update its contents, a message is sent to the model to retrieve the data to be presented. Both browsers have a text editing view. This view sends messages to retrieve its contents from the model and to signal when the user has accepted changes. The final type of subview is a button. Buttons send a message to the model indicating that they have been activated.

7.3. Syntax Browser

The layout of the syntax browser is shown in Figure 7.3 and consists of six areas:

Boundary View: A list of boundary names;

Feature View: A list of ordering features;

Category View: A list of production categories;

Production View: A list of production names;

Display Buttons: Two buttons controlling how vector definitions are displayed in production definitions;

Editing View: A text editor for editing production definitions.

Some of the views in the syntax browser are dependent on others in that a change in one causes a change in another. These dependencies are shown by the heavy arrows in Figure 7.3 and described below:

- Making a selection in the *boundary*, *feature*, or *production view*, causes the corresponding boundary comment, feature comment, or production definition, respectively, to be displayed in the *editing view*

[2]The message is specified by the programmer when the list view is created.
[3]An application usually has a single model but several view-controller pairs.

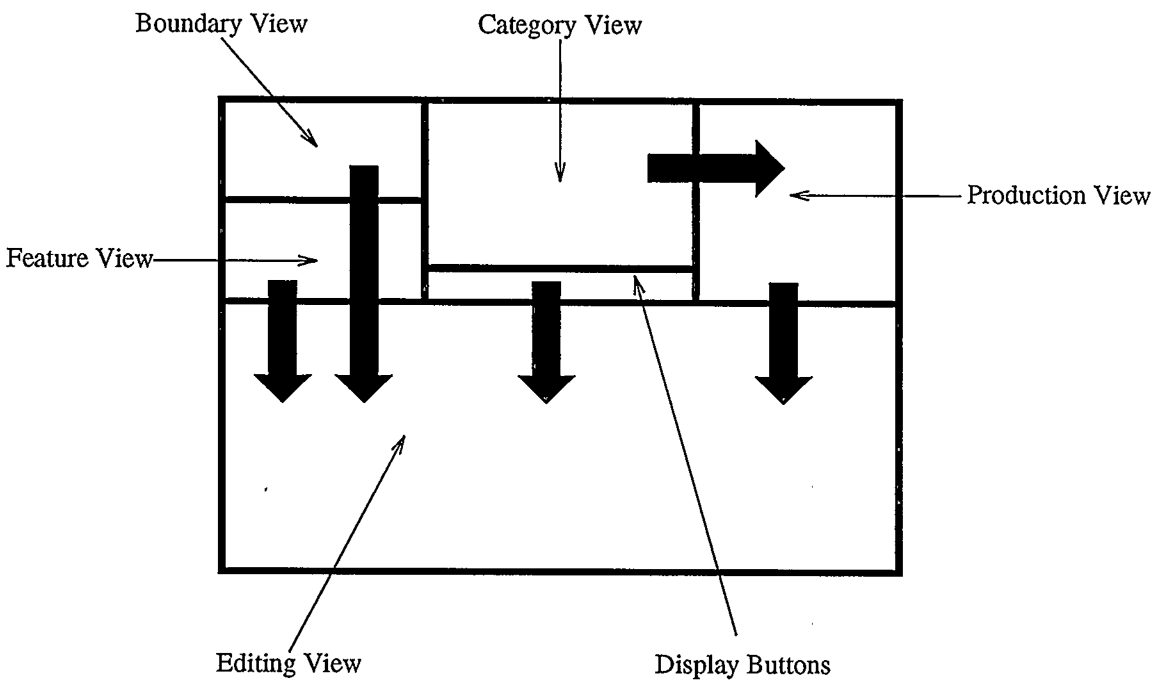


FIGURE 7.3. Layout of the syntax browser.

- Making a selection in the *category view* causes the names of productions in that category to be presented in production view;
- Setting the *display buttons* determines the way ternary vectors are formated.

The model–view structure of the syntax browser is presented in Figure 7.4. The central syntax browser is the model, and is connected to the *list*, *button*, and *editor views* to the left, right, and bottom. The parser is at the top and communicates with the browser. Connecting lines indicate messages that are sent between specific views and the browser model. The lines are labeled with the purpose of the message.

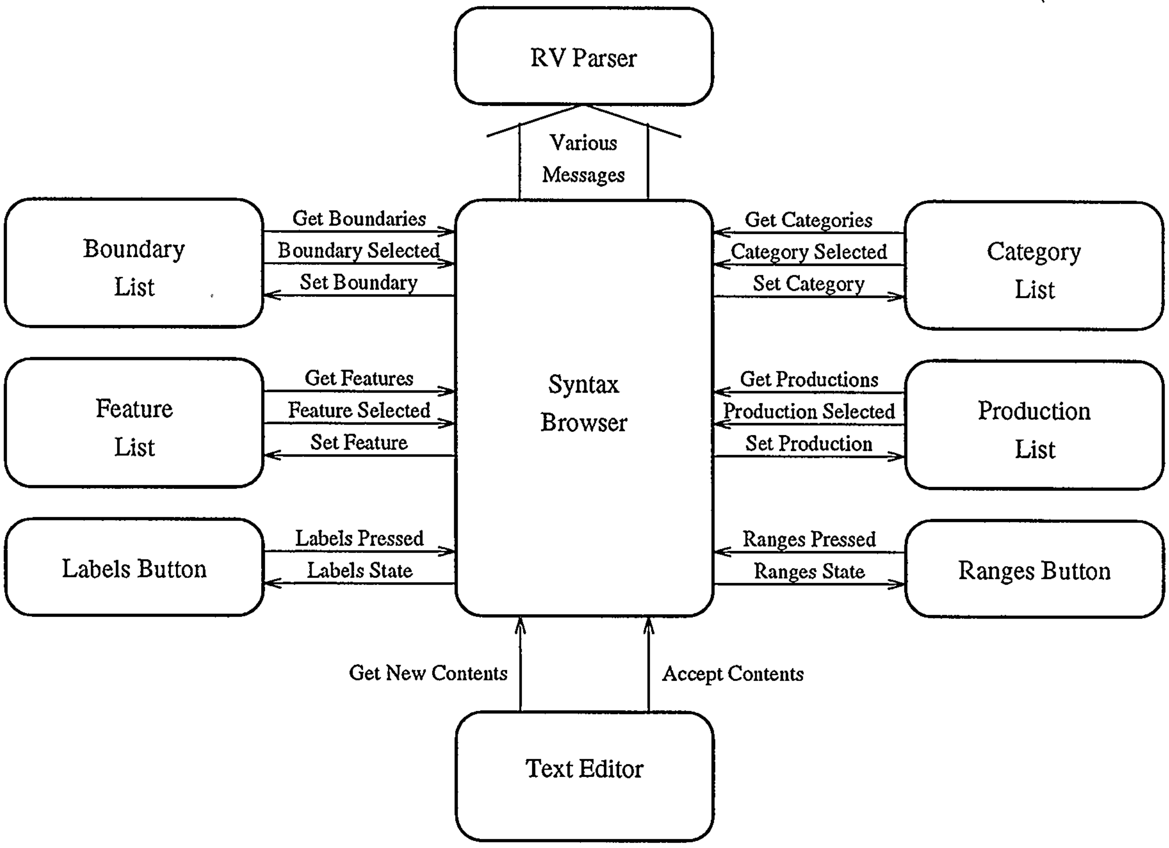


FIGURE 7.4. Syntax browser model–view structure.

7.4. Ternary Vector Editing

Part of the design goal was to make the low level of detail inherent in RV convenient for the developer to manage. To this end, Blank's notion of vector macros (Blank, 1989) has been designed into the browsers. These macros provide the ability for developer to define shorthands for commonly used feature configurations. For example, there is generally a large set of productions that are not usable while parsing noun phrases. In their condition vectors, they check for feature values that indicate that a noun phrase is not being parsed, e.g. `-DET..HEAD`. Instead of the developer having to specify `-DET..HEAD` in the condition vector for each of these productions, they can define a macro, called `NPoff` for example, which has the value `-DET..HEAD`. They can then use this macro in a vector definition, e.g. `+S +AUX #NPoff`.

This macro feature is also available in the lexicon browser for use with semantic feature vectors. An example of its use here is in constructing a hierarchy of features. For example, given the features `ANIMATE HUMAN PERMANENT STATE FEELING EVENT ACTION TRANSFER POSS LOC DEST`, some macros that could be used to construct a hierarchy are: (also shown are the resulting vectors)

ANIMATE: `+ANIMATE -STATE..DEST`

+ANIMATE -STATE..DEST

HUMAN: `#ANIMATE +HUMAN`

+ANIMATE..HUMAN -STATE..DEST

STATE: `-ANIMATE..PERMANENT +STATE -ACTION..TRANSFER`

-ANIMATE..PERMANENT +STATE -ACTION..TRANSFER

EMOTION: `#STATE +FEELING`

-ANIMATE..PERMANENT +STATE..FEELING -ACTION..TRANSFER

Notice that macros can be defined in terms of other macros. This applies to ordering feature macros in the syntax browser as well.

The syntax and lexicon browsers both provide access to macros through a popup dialog box that allows macros to be created, edited, and deleted.

7.5. Lexicon Browser

The layout of the lexicon browser is shown in Figure 7.5 and consists of eight areas:

Feature View: A list of semantic features;

Relation View: A list of relation names;

Property View: A list of morphosyntactic properties;

Paradigm View: A list of paradigm names;

Category View: A list of lexical entry categories;

Entry View: A list of lexical entry names;

Display Buttons: Two buttons which control the format of the entry display, specifically:

(1) whether categories removed by the aging mechanism are displayed,

(2) whether category scores are shown.

Editing View: A text editor for editing entry, paradigm, and relation definitions, agreement sets, and feature comments.

As with the syntax browser, some of the views in the lexicon browser are dependent on others. These dependencies are shown by the heavy arrows in Figure 7.5 and described below:

- Making a selection in the *feature*, *relation*, *paradigm*, or *entry view* causes the corresponding feature comment, relation definition, paradigm definition, or entry definition, respectively, to be displayed in the *editing view*;

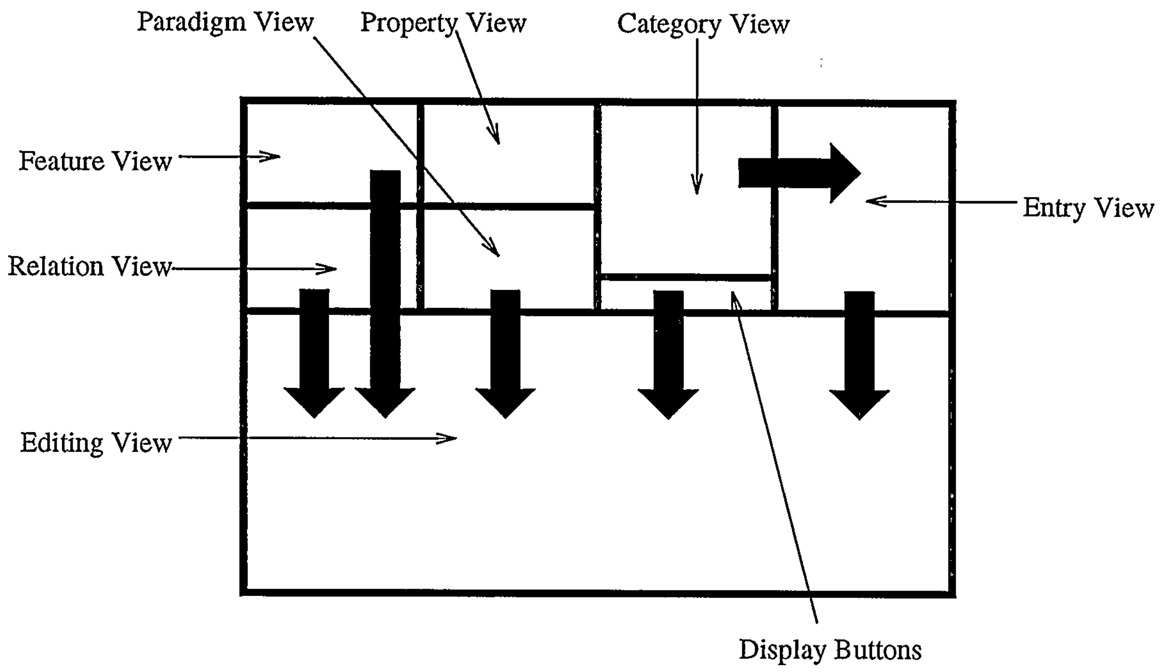


FIGURE 7.5. Layout of the lexicon browser.

- Making a selection in the *category view* causes the names of lexical entries in that category to be presented in the *entry view*;
- Setting the display buttons enables or disables the display of the category aging information.

The model–view structure of the lexicon browser is presented in Figure 7.6.

7.5.1. MorphoSyntactic Property Editing. To make editing of agreement properties more convenient for the parser developer, the browser provides a shorthand notation when editing paradigm definitions.

Specific properties can be entered, e.g. *first:plural:past* from the example in §2.7.2.1. However, suppose the only thing of interest is the first-person-plural property. The

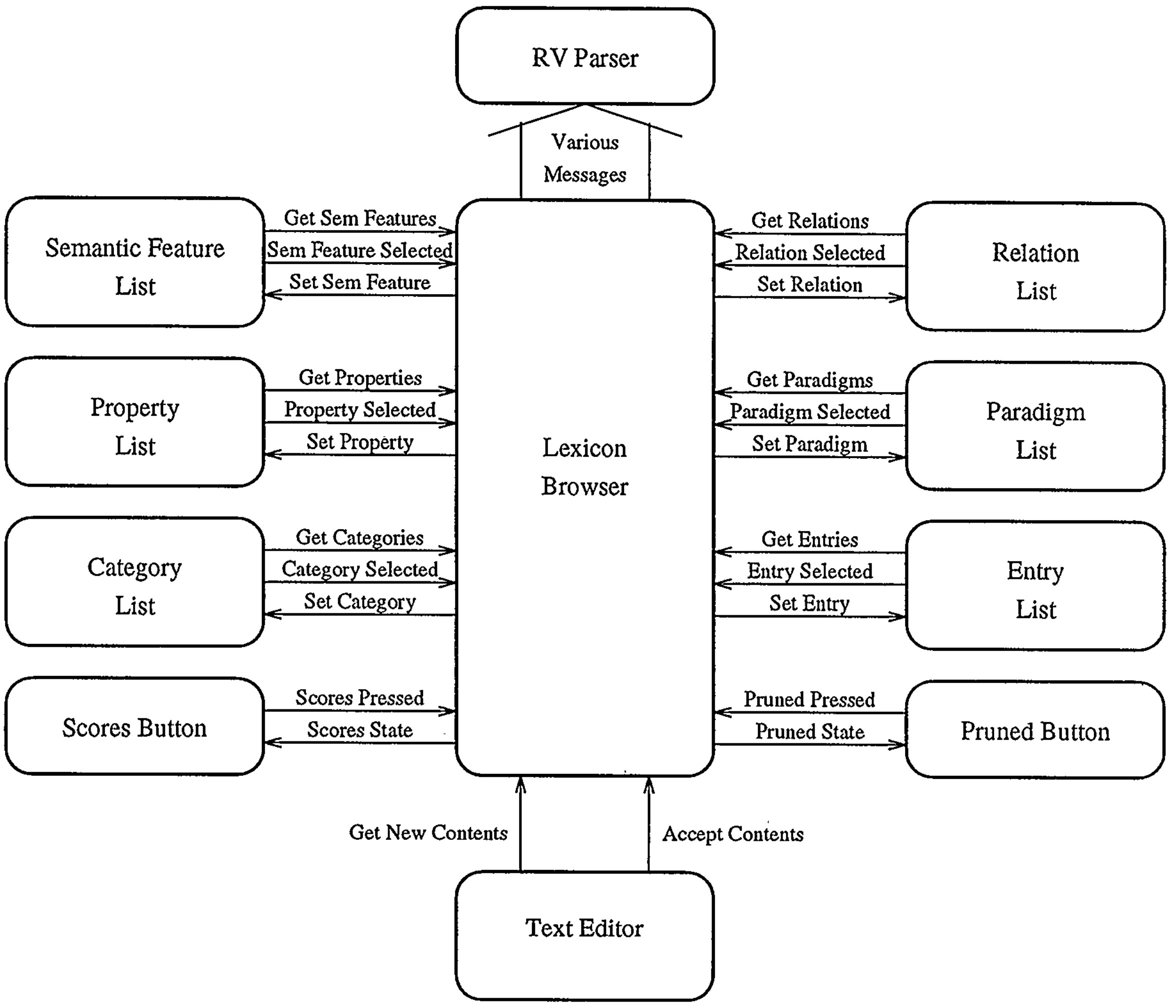


FIGURE 7.6. Lexicon browser model–view structure.

shorthand allows the designer to specify *first:plural*. This denotes the subset of agreement properties that contain *first* and *plural*. In this case the full set of specified properties is {*first:plural:past*, *first:plural:present*}. When a paradigm definition is displayed, the property sets are simplified as much as possible to provide the developer with the simplest property set description. The algorithm for doing this is detailed in Figure 7.7. The following example illustrates how the algorithm works.

Assume three agreement sets[4]: $A = \{a_1\ a_2\ a_3\}$, $B = \{b_1\ b_2\ b_3\}$, and $C = \{c_1\ c_2\}$. The cross product of these would result in eighteen agreement properties. Now suppose we want to specify the set of agreement properties: $S = \{a_1{:}b_1{:}c_2\ a_1{:}b_3{:}c_2\ a_1{:}b_2{:}c_1\ a_1{:}b_2{:}c_2\ a_2{:}b_2{:}c_1\ a_2{:}b_2{:}c_2\ a_3{:}b_2{:}c_1\ a_3{:}b_2{:}c_2\}$. A simplest description of S is $\{a_1{:}c_2\ b_2\}$. A simplest description specifies only members of those agreement sets that are constrained, any sets that have no members specified imply all members of that set. In our example, the description means:

$$\{a_1\} \times B \times \{c_2\} \cup A \times \{b_2\} \times C$$

Referring to Figure 7.7, **props** is initialized to S. We go through each agreement set (A, B, and C) beginning with A, so **aSet** $= A$. Looking for an element of S containing a_1 (i.e. **p**) we find $\{a_1\ b_1\ c_2\}$. This becomes the first instantiation of **propSet**. Next, we check if $\{a_2\ b_1\ c_2\}$ and $\{a_3\ b_1\ c_2\}$ are also members of S. We do this since **rest** $= \{b_1\ c_2\}$ and we are looking for all sets that are supersets of **rest** and whose elements other than **rest** are proper subsets of **aSet**, in this case A. In this example, this means that we are looking for sets that are made up of $\{b_1\ c_2\}$ and something from A. If so, mark all three and add $\{b_1\ c_2\}$ to S. Further, if all elements found (e.g. all three containing $\{b_1\ c_2\}$) were already marked the element

[4]An agreement set is a set of property values for an agreement feature.

that is added is marked as well. Note that a marked element can still be used in the search. We continue by searching for the next element of S containing a_1. This is repeated until all elements of S have been examined. This results in $S = \{\{a_1\ b_1\ c_2\}\ \{a_1\ b_3\ c_2\}\ \{a_1\ b_2\ c_1\}^*\ \{a_1\ b_2\ c_2\}\ \{a_2\ b_2\ c_1\}^*\ \{a_2\ b_2\ c_2\}\ \{a_3\ b_2\ c_1\}^*\ \{a_3\ b_2\ c_2\}\ \{b_2\ c_1\}\}$.

We repeat this for sets B and C. At the end of this process $S = \{\{a_1\ b_1\ c_2\}^*\ \{a_1\ b_3\ c_2\}^*\ \{a_1\ b_2\ c_1\}^*\ \{a_1\ b_2\ c_2\}^*\ \{a_2\ b_2\ c_1\}^*\ \{a_2\ b_2\ c_2\}^*\ \{a_3\ b_2\ c_1\}^*\ \{a_3\ b_2\ c_2\}^*\ \{b_2\ c_1\}^*\ \{b_2 c_2\}^*\ \{a_1\ c_2\}\ \{a_1\ b_2\}^*\ \{a_2\ b_2\}^*\ \{a_3\ b_2\}^*\ \{b_2\}\}$.

The minimal description of S consists of those elements that remain unmarked: $\{\{a_1\ c_2\}\{b_2\}\}$.

```
SIMPLIFY (someProps)

props is assigned the contents of someProps
for each agreement set (aSet)
   p is assigned the first element of aSet
   for each propSet in props containing p
      rest is assigned propSet − p
      group is assigned all ps in props such that
         rest is a subset of ps and (ps − rest) is a proper subset of aSet
      if each member of group contains a unique member of aSet
         rest is marked if all members of group are
         mark all members of group
         add rest to props
```

FIGURE 7.7. Property simplification algorithm

7.6. Restricted browsers

In addition to the two general browsers described above, the system includes several more limited browsers. These allow access to the parser at various levels of granularity. Typically these limited browsers are opened in response to an operate menu command from the related pane in another browser. These browsers are described below:

- Browse productions that refer to a specified boundary; (Opened from the *boundary view* of the syntax browser.)
- Browse productions that are named in the category set of a specified lexical entry; (Opened from the *entry view* of the lexicon browser.)
- Browse a single production; (Opened from the *production view* of the syntax browser. This is of use when a production is desired for comparison with others. This saves keeping a full syntax browser open just for this.)
- Browse non-lexical productions; (Opened from the *category view* of the syntax browser.)
- Browse a single lexical entry; (This serves a similar purpose to the single production browser.)
- Browse all lexical entries that name a specified production in their category set; (Opened from the *production view* of the syntax browser.)
- Browse all paradigms that reference a specified property. (Opened from the *property view* of the lexicon browser.)

7.7. Graph Browsers

The previously described browsers are textual. The development environment also provides two graphical browsers: one for the lexical trie, and one for semantic structures. Both browsers have identical structure and so are discussed together. They consist of two views, shown in Figure 7.8. The *graph view* presents a graphical representation of a graph structure: the lexical trie or a semantic structure. Clicking the left mouse button on a node in the displayed graph causes the textual definition of the contents of that node to be displayed in the *data view*. Thus, there is a single inter-view dependency in these browsers, as shown by the heavy arrows in Figure 7.8.

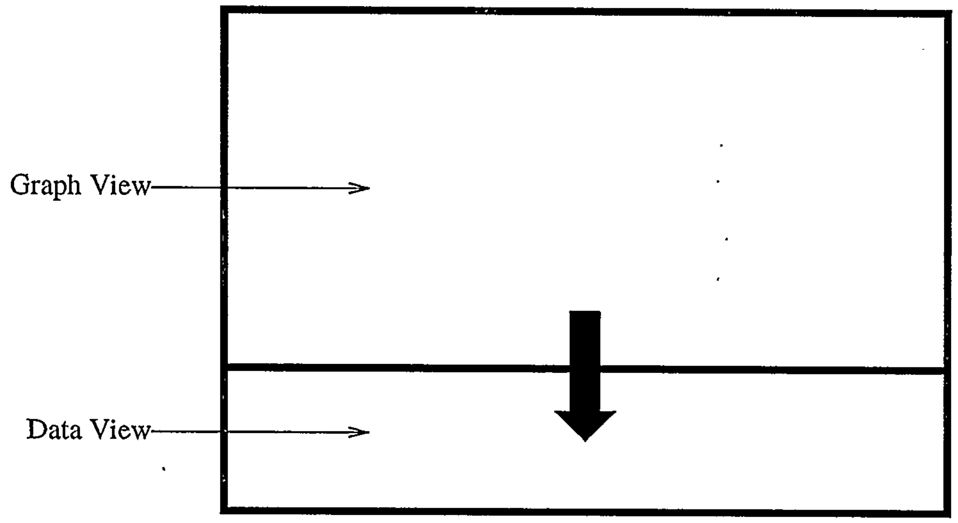


FIGURE 7.8. Layout of the graph browsers.

The graph browsers require facilities for the construction, layout and presentation of simple, directed graphs with the option of labelling edges[5]. A easy-to-incorporate, public domain package was found that met these requirements. It is the Grapher package written by Mario Wolczko at The University of Manchester (Wolczko, 1992). I decided to use this package to meet the graph display requirements of the development tools.

Having now discussed general issues of the graph browsers, I now discuss design issues particular to each.

7.7.1. Lexical Trie Browser. Figure 7.9 shows an example trie as the browser would present it.

Any node can be selected and its contents shown in the *data view*. This can then be edited and accepted. For internal nodes this value is what is displayed in the node.

[5]This requirement is explained in §7.7.2.

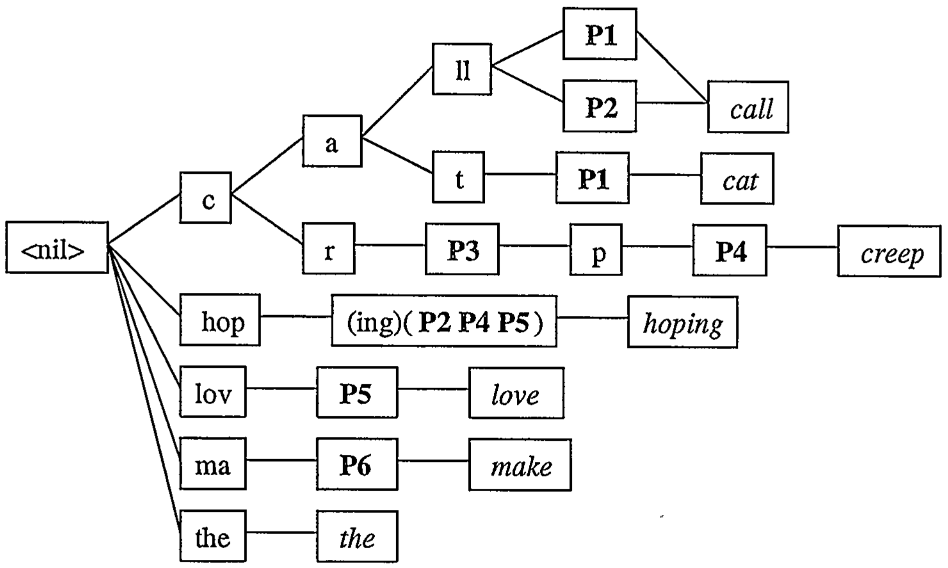


FIGURE 7.9. Example trie presentation.

For a lexical entry (i.e. the leaf nodes) the full definition of the node is presented, as in the lexicon browser.

In addition to this display-and-edit capability, there is a menu associated with each node in the trie. This menu allows the developer to inspect the internal structure of the node or to remove the node and its descendants from the trie.

A final feature of the trie browser is the ability to select text in the *data view* and have the trie perform a **LOOKUP** of it. If the lookup is successful (i.e. the selected text was recognised) the path(es) through the trie that resulted in the recognition are highlighted. This is done by using heavier lines for arcs and node outlines.

7.7.2. Semantic Structure Browser. Nodes in the semantic structure are represented by displaying the name of the associated lexical entry. Relationships between nodes are more involved than in the trie. In the trie, the only relationship is a parent-child one. In a semantic structure, the exact relationship is based on what semantic role the child is bound to. This requires a method of denoting these relationships in the graphical representation of the structure. The graphing package's edge labelling capabilities were made use of here, enabling the name of the relationship (i.e. the semantic role) to be used as a label on the arcs connecting nodes. Figure 7.10 shows an example of a semantic structure as the browser would present it.

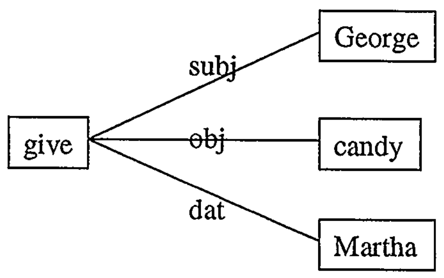


FIGURE 7.10. Example semantic structure presentation.

7.8. Debugger Design

As with the browsers, the debugger window consists of several views as shown in Figure 7.11:

Input View: Contains text to be parsed.

Control Panel: Contains various buttons and switches that control the operation of the debugger.

Trace View: Displays parser and debugger output.

Boundary List: Contains the list of boundary register names.

Boundary Display: Displays the contents of a boundary register.

Gramrole List: Contains the list of grammatical role names.

Gramrole Display: Displays the contents of a grammatical role.

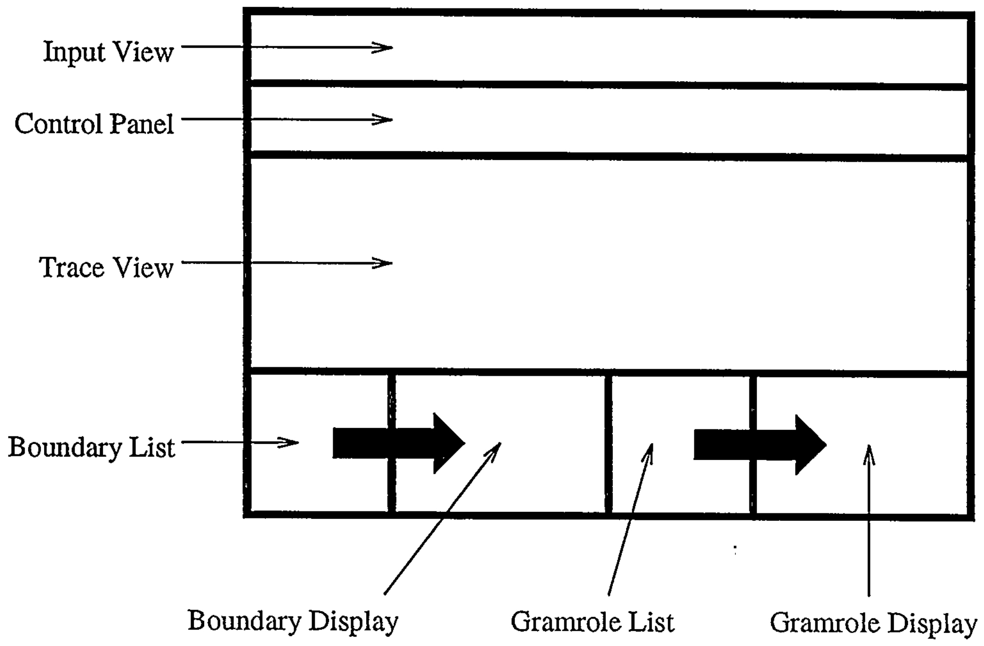


FIGURE 7.11. Debugger layout.

Inter-view dependencies are much simpler in the debugger than in either textual browser: selecting an item from the boundary or grammatical role list causes the corresponding *display view* to display the contents of the selected item. This is shown by the heavy arrows in Figure 7.11.

The *trace view* is a transcript that the parser writes its output to. The operate menu for the *trace view* gives the user access to dialog boxes that allow them to determine the level of detail in the parser's output. Other than that, only a **Clear** operation is available which empties the *trace view*.

The debugger operates in one of two modes: single-stepping or free-running. The mode is controlled by the switch at the right end of the control panel. The central part of the control panel contains buttons for controlling the parser. **Parse** and **Parse to...** are used in free-running mode, while **Step**, **Stop at...**, and **Continue** are used in single-stepping mode. When the parser is started, it is given the selection in the *input view* as its input stream. The parser controls are described below:

- Free running mode:

  Parse: Starts the parser in free-running mode.

  Parse to...: As **Parse** but stops when a specified production has fired (the user is given a dialog box for choosing the production).

- Single stepping mode

  Step: Causes the parser to execute the **CYCLE** algorithm once (see §2.1.1).

  Stop at...: Switches to free-running mode until a specified production is fired.

  Continue: Switches to free-running mode and continues the parse to completion.

To the left of the parser controls there is a button that passes the *input view*'s selection to the lexical acquisition module. This is mainly useful in testing of the learning mechanism.

To meet the design requirement that there be convenient access to sentence testbeds, the *input view* may contain any amount of text, each sentence terminated by a carriage-return character. Lines may be made comments by prepending them with a '#' character. This provides a method of annotating testbeds. At the far left of the control panel are two buttons for moving through the *input view*. The top button causes the first sentence in the *input view* to be selected. The other button moves the selection to the next sentence.

The debugger was designed to provide the parser developer access to all pertinent information. This is accomplished by the *trace view*, the boundary list and the grammatical role list. The *trace view* shows step by step operations of the parser in a selectable level of detail. The boundary list allows access to all boundary registers, including CURR which contains the parser's state. When a boundary register is being displayed, the operate menu of the *display view* contains an option which allows the developer to compare the condition vector of any production to the state that is stored in the displayed register. The result of the comparison is a vector that could be used to **change** the state in the register to allow the selected production to **match** it.

The grammatical role list provides access to the semantic structure that is being built, during the parse. When the parse has completed, the resulting semantic structures can be accessed using the semantic structure browser.

7.9. Launcher

The final component of the RV development environment is the launcher. This is a small tool that is used to manage parsers. It allows several parsers to be loaded and easily accessible. This is done by presenting the names of the parsers in a list. It allows parsers to be loaded from files, written to files (in both human and machine readable formats), renamed, copied, and removed from the system. Finally it provides a set of buttons that allow the browsers and debugger to be opened on the parser selected in the list.

7.10. Summary

This chapter began by presenting the design requirements for a set of RV development tools. Put concisely, these requirements are:

- The tools should allow and encourage interactive and incremental development;
- Editing should be convenient for the developer, minimizing the sources of operator error (e.g. spelling errors);

- The system under development should be able to be viewed from various organizational standpoints and at various levels of granularity;
- Adequate information should be provided (in controllable detail) to the developer during parser testing.

The bulk of the chapter described the design of the tools. The result of this design is a comprehensive set of tools allowing incremental, online browsing, editing, and debugging of all aspects of an RV system.

Chapter 8
Vocabulary Learner

A broad coverage lexicon has to be large. There are two alternatives in constructing a large lexicon: 1) by hand or 2) (semi-)automatically. The latter reduces the load on the parser developer, and also reduces the possibility of making errors while entering lexical information. The lexicon acquisition mechanism described here, is semi-automatic. The developer must create a skeletal lexicon, containing definitions of paradigmatic variation and representative open class words. The acquisition mechanism (referred to as the learner) can then acquire more open class words on the fly, based on information in the skeletal lexicon.

As the learner sees different forms of a word, it tries to recognise known suffixes. The learner maintains associations between suffix sets (implemented by paradigms (see §2.7.2)) and categories (see §2.7.1). This allows it to infer the categories of a word from the possible sets of suffixes for the word. This results in an overly general categorization of the word. Unused categories are later removed by an aging mechanism.

The learner makes assumptions about prior knowledge. Specificly, it assumes that all suffix sets are known, and that for each suffix set there is at least one word in the initial lexicon that uses it. The learner is intended to extend a lexicon, not generate one from scratch. Also, the initial lexicon should contain all closed class words (e.g. prepositions and pronouns) and words that do not have a regular suffix structure.

8.1. Learning the Lexicon

This section explains the learning mechanism. There are two stages to the mechanism's operation: 1) incorporating a new word into the lexicon, and 2) refining the new word's categorization. The learning algorithm is shown in Figure 8.1, and explained below.

8.1.1. Learning a new word. When **LOOKUP** can not recognise a word, the word is extracted from the input stream. For this purpose, a word is defined as being terminated by whitespace or punctuation. The learner must then create both a lexical entry and trie-path for the new word. Literal nodes are first added to the trie. The final internal node in the path (i.e. the parent of the lexical entry) is then processed with the goal of converting a recognised suffix or suffixes to one or more paradigm-set nodes. The new lexical entry is then assigned categories based on the paradigms contained in its immediate parent(s).

The final step is to organize the trie for optimum efficiency. During processing of the node(s) containing the possible suffix, some rearrangement of that part of the trie may need to be undone. Once the new word has been incorporated into the lexicon, parsing can resume at the point where the new word was encountered. This time it will be recognised and parsing can proceed.

8.1.1.1. *Initial trie path creation.* The **LEARN** algorithm creates a path in the trie for the new word by using only literal nodes. This can involve both the insertion of new nodes and the splitting of existing ones. Figure 8.2a shows the string "*communication*" being added to the trie of Figure 2.11. In this case a literal node is added.

```
LEARN (word)

Add a template for word to the lexicon dictionary
Add literal nodes to the lexical trie so that word will be matched
        (If necessary split existing literal nodes when there is a partial match)
Add new–leaf to the end of the new word path, for the new lexical entry
p is assigned the parent node of new–leaf
potential–suffixes is assigned:
   the set of all known suffixes partially matching p such that
        there is a node containing a vowel between the root and new-leaf,
        excluding the suffix being considered
if potential–suffixes is empty
   flatten the trie in the region of p (see text)
   p is again assigned the parent node of new–leaf
   potential–suffixes is again assigned:
        the set of all known suffixes partially matching p such that
           there is a node containing a vowel between the root and new-leaf,
           excluding the suffix being considered
      if potential–suffixes is not empty
        parent is assigned the parent of p
        remove p from parent
        add a subtrie to parent for each member of potential–suffixes
        group these new subtries within parent
for each parent of new–leaf
   invoke LEARN–WITH–SUFFIX (parent, new–leaf)
refactor the trie to restore the correct trie structure


LEARN–WITH–SUFFIX (suffix–node, leaf)

1. Create a new paradigm–set node
if suffix–node.string is known
   replace suffix–node with a new paradigm-set node
      that includes all paradigms with suffix–node.string as a string
   add suffix–node.string to the paradigm–set suffixes

2. Assign categories
Give leaf the set of all categories associated with its parents' paradigms
if the lexical entry now has an empty category set
   (i.e. no paradigm or paradigm–set parents; the suffix is not known)
   give it all possible lexical and semi–lexical categories
```

FIGURE 8.1. Algorithm LEARN

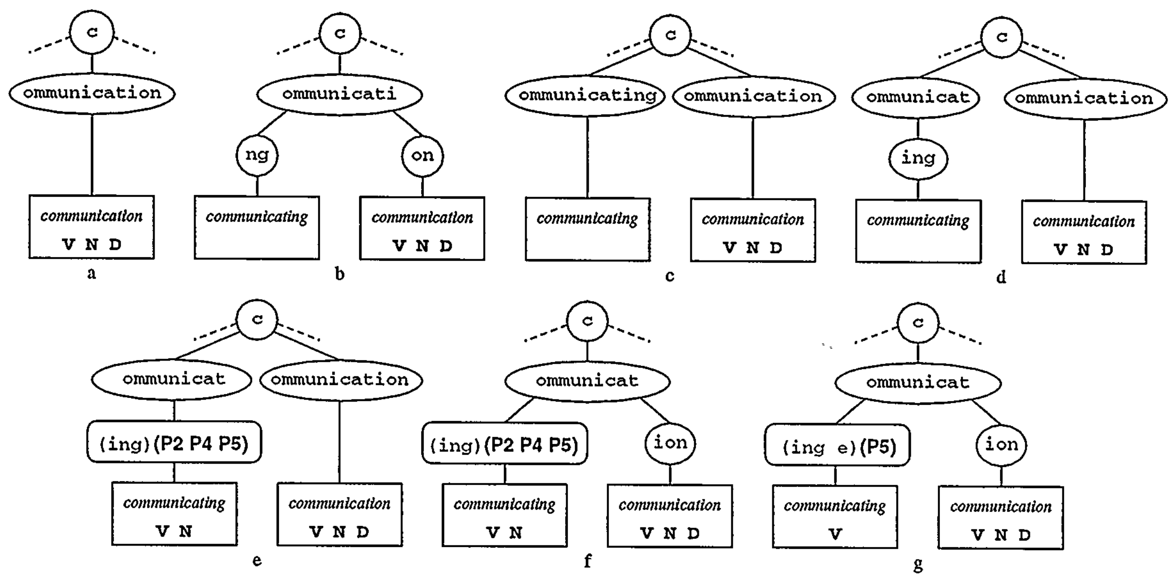


FIGURE 8.2. Learning "*communication*" (a), "*communicating*" (b–f) and "*communicate*" (g).

The word "*communicating*" is added in Figure 8.2b. This causes the "*ommunication*" node to be split.

8.1.1.2. *Rearranging the suffix.* The goal of the learner is to recognise a suffix[1] in the parent node of the new lexical entry. It is unlikely that the node contains exactly a suffix. In most cases, there will be extra characters in the node, or the suffix may be split between two nodes.

The first case is handled by creating all right-justified substrings from the contents of the node. A branch is made in the trie for each suffix found. For example, the string "*oking*" added in Figure 8.3, results in the strings: "*oking*", "*king*", "*ing*", "*ng*", "*g*". Using the paradigms in Table 2.4 we see that the strings "*king*" (from P6) and "*ing*" (from P2, P4, and P5) are valid suffixes. In this case the "*o*" is split

[1]Note that for my purposes in this chapter, a suffix is defined as a string that occurs in a paradigm which is found in parent nodes of lexical entries.

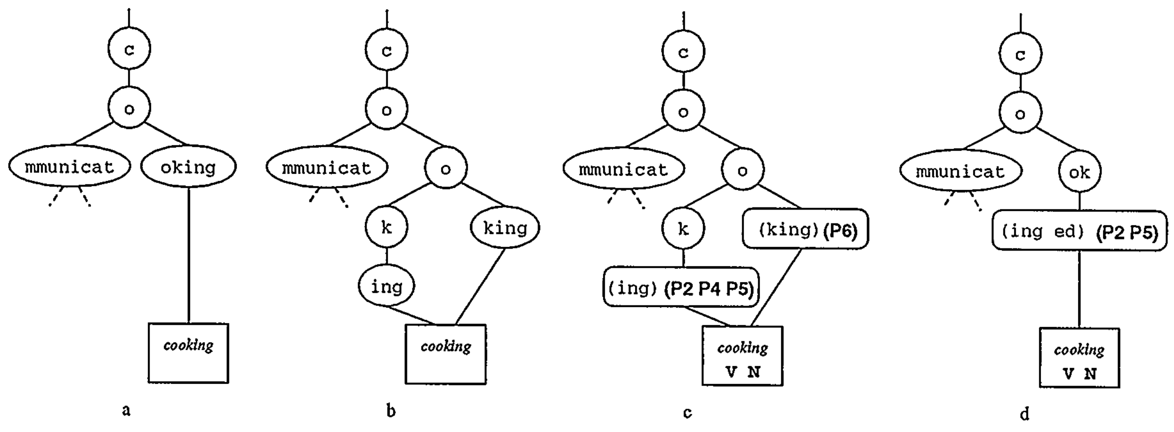


FIGURE 8.3. Learning "*cooking*" (a–c) and "*cooked*" (d).

off into a separate node and two branches are created: one containing "*k*" and "*ing*", and the other containing "*king*".

There is a constraint on what suffixes can be used: they must result in a reasonable word stem. For English-like languages, stipulating that the stem contain a vowel seems valid. This means that if the suffix is removed from the last internal node on the new word path, at least one of the nodes on the path contains a vowel. For a literal node, it is simply a matter of checking for a vowel in the string. For a paradigm node, each substring must contain a vowel. Finally, for a paradigm–set node at least one candidate paradigm must have a vowel in all substrings. This constraint keeps the mechanism from missegmenting in some cases. Without this constraint, the learner would interpret "*red*", and "*ring*" as the stem "*r*" with suffixes "*ed*" and "*ing*".

If the lexical entry's parent does not contain a suffix, the suffix may be split between that node and its parent. The trie is flattened locally (only the lexical entry's grandparent, parent, and parent's siblings are involved). A suffix is again searched for as described above.

Figures 8.2b and 8.2c show *"communicating"* being added to the trie. The second *"i"* is placed in the stem string by the trie-building mechanism, leaving "*ng*" in a node by itself. The *"ommunicati"* node and its children must now be flattened. The string from this node is prepended to each child's string. The children are then added as children of their grandparent and their original parent node is removed. Figure 8.2c shows the result of doing this. The "*ing*" suffix can now be found in the "*ommunicating*" node, as shown in Figure 8.2d.

8.1.1.3. *Paradigm–set node creation.* If there is now a known suffix in the lexical-entry's parent node, it is replaced with a paradigm-set node containing all paradigms that cover the suffix. In Figure 8.2e the "*ing*" from "*communicating*" is replaced by a paradigm–set node containing paradigms P2, P4 and P5. Also, Figure 8.3c shows this for "*ing*" in P2, P4 and P5, as well as "*king*" in P6.

The set of new branches containing paradigm–set nodes is remembered by their common parent. This is used by a later learning mechanism. For example, in Figure 8.3c the two paths leading to "*cooking*" are grouped together and remembered.

8.1.1.4. *Initial category assignment.* The set of initial categories for a new lexical entry is computed by first collecting all paradigms from its paradigm-set parents. Each paradigm has associated with it a set of categories. The union is taken of all these category sets. This is the lexical entry's initial category set.

The paradigm→categories association is computed by taking the union of the category sets of all lexical entries that have one or more parents containing the paradigm. For efficiency, this association is cached by storing the set union in the paradigm.

If the new lexical entry has only literal parents (i.e. no suffixes were found) its initial category set contains all lexical and semi-lexical categories. Figure 8.2a shows this for "*communication*".

Paradigm	Lexical Entries	Categories
P2	*call*	V, N
P4	*creep*	V
P5	*communicating*, *love*	V
P6	*make*	V

TABLE 8.1. Paradigm – Lexical Entry correspondences

An example of initial category assignment occurs when "*cooking*" is learned. Its parents contain P2, P4, P5, and P6 between them. The category associations for these paradigms are shown in Table 8.1. The categories from column three of the table are placed in a set, resulting in {V, N} being the initial category set of "*cooking*".

8.1.2. Adjustment of Categories. The initial category set assigned to a new lexical entry is always a superset of the correct category set. When paradigms are removed from a paradigm-set parent, categories that are no longer supported by the parents' paradigms are removed. This helps reduce the generality of the category sets. Only productions named in the category sets will be examined when the lexical entry is encountered. Because of this, it is in the best interest of efficiency to have the category set as small as possible. To this end, there is an aging mechanism that keeps track of how frequently each category is used. Frequently used categories become persistent and rarely used ones are removed. If the system is provided with sentences representative of the accepted usage of the new lexical entry, the category set should converge on an appropriate, stable value.

Refer to the lexical entry "*cooking*" from the above examples. As it is used in sentences (for purposes of this example, I assume that "*cook*" is used predominately as a verb and not as a noun) the scores of the V and N categories are adjusted. As V is used frequently and N infrequently (if at all), V will become permanent and N will

```
UPDATE (update-info)
for each (lex-entry, cats-used) in update-info
  for each used-cat in cats-used
    if lex-entry.used-cat is not persistent
      increase lex-entry.used-cat.score
    if lex-entry.used-cat.score > persistence-threshold
      make lex-entry.used-cat persistent
    assign -1 to lex-entry.used-cat.age
  for each cat in the category set of lex-entry
    increase lex-entry.used-cat.age
    if lex-entry.used-cat.age > the old-age threshold
      decrease lex-entry.used-cat.score
      if lex-entry.used-cat.score = 0
        remove used-cat from lex-entry's category list
```

FIGURE 8.4. Algorithm UPDATE

be removed. If "*cooking*" (i.e. "*cook*") is also frequently used a noun both categories will eventually become permanent. The aging mechanism works as shown in the UPDATE algorithm in Figure 8.4. The input to UPDATE is a dictionary that maps lexical entries to sets of categories. This mapping is constructed throughout the parse. UPDATE is called whenever a successful parse is found.

8.2. Changes to the LOOKUP Algorithm

The LOOKUP algorithm (see Figure 2.10) has to be extended to handle the new paradigm-set node. The new algorithm is shown in Figure 8.5. The added lines are marked on the left.

When different forms of the word are seen later, their suffixes are added to the paradigm set. This causes the removal of paradigms that no longer cover all suffixes encountered. Figure 8.3d shows the result of encountering "*cooked*". The string "*ked*" is not included in paradigm P4, nor in P6, so when LOOKUP adds "*ked*" to suffixes

```
LOOKUP (input, node, props)

interpretations is initialized empty
case node
  leaf
    if input is empty or the first character is non-alphabetic
      add (node, props) to interpretations
  string pattern
    if node.string matches a prefix of input
      add to interpretations:
        $\bigcup_{\text{child}}$ LOOKUP (remaining input, node.child, props)
  paradigm pattern
    for each (string, properties) of the paradigm
      if string matches a prefix of input and props $\cap$ properties $\neq \phi$
        add to interpretations:
          $\bigcup_{\text{child}}$ LOOKUP (remaining input, node.child, props $\cap$ properties)
        return interpretations
| paradigm–set pattern
|   for each (string, properties) of each paradigm in (suffixes, paradigms)
|     if string matches a prefix pref of input and props $\cap$ properties $\neq \phi$
|       let temp–interpretations be
|         $\bigcup_{\text{child}}$ LOOKUP (remaining input, node.child, props $\cap$ properties)
|       if temp–interpretations is not empty
|         add to interpretations: temp–interpretations
|         add pref to suffixes
|         delete any member of paradigms that does not contain pref
|         if paradigms is now empty
|           remove the trie branch containing node
| for each group of children, cg, which contains a path to a leaf
|   for each child in cg which did not lead to a leaf
|     remove the subtrie rooted at child (except the leaf) form the trie
return interpretations
```

FIGURE 8.5. The revised algorithm LOOKUP with support for vocabulary acquisition.

in the paradigm–set node that includes P6, P6 is deleted. Since the node now has no paradigms, that branch of the trie can be removed. Also P4 is removed from the other node since it does not cover "*ed*". Figure 8.2g shows the removal of the paradigms P2 and P4 from the new paradigm–set node, when "*communicate*" is encountered. Recall that when children were added by the acquisition mechanism (see Figure 8.1) they were grouped together. This grouping information is now used: trie branches are removed only if at least one other member of the group provides a path to a leaf.

8.3. Limitations and Problems

The vocabulary learner described in this section is the initial attempt at adding automated lexical acquisition to the RV parser and development system. In its current form it has several problems.

Of major concern is the removal of paradigms from paradigm–set nodes. If the word being learned has a single set of endings, (i.e. the paradigm–set node should ideally reduce to a single paradigm) then there is no problem. However, if the word can validly take more than one set of endings there is a problem. Consider the lexical entry for "*call*" in Figure 2.11: it takes two sets of endings as described by paradigms P1 ("*call*" as a noun) and P2 ("*call*" as a verb). If call was being learned and both forms were being encountered, two things could happen:

(1) If a common suffix is encountered first, the paradigm-set node will contain both paradigms (P1 and P2). The next form that does not contain one of the common suffixes will dominate, causing the other paradigm (and hence wordsense) to be removed and lost. If "*call*" is then used in the other sense, there are two possibilities:

(a) If a common suffix is used the word will be recognized using the existing paradigm and miscategorized. This will very likely cause the parse to fail.

(b) If a form-specific suffix is used, a new lexical entry will be created for this sense of the word.

(2) If, for example, a noun-specific suffix is first encountered the first occurrence of a verb-specific suffix will cause a new lexical entry to be created.

A possible approach to correcting this deficiency of the current algorithm would be not to remove paradigms (and trie branches) based on what suffixes they cover versus what suffixes have been encountered. Rather, paradigms (and branches) would only be removed when a category is removed from the lexical entry by the category aging mechanism. Any paradigms that only supported the removed category would be removed. This could be done since usage of the word indicates that that category is inappropriate, and thus paradigms specific to that category are also not appropriate.

Another approach that would avoid this problem to some extent is to have paradigms marked by the parser designer as being general or special purpose[2]. The learner would then only consider general purpose paradigms when evaluating the validity of suffixes and constructing paradigm-set nodes.

8.4. Summary

This chapter described a method for acquiring vocabulary on-the-fly in RV systems. The learner incorporates new words into the lexicon and trie, and assigns an initial set of categories. As more forms of the word are encountered, a paradigm will eventually be selected to account for the set of suffixes seen, and a more precise categorization

[2]An example of this is P6, which is designed for use with the irregular verb "*make*".

will be assigned as useful categories are repeatedly seen. The learning system runs together with the parser, so that vocabulary learning does not require extra effort from the user.

CHAPTER 9

Evaluation

This chapter evaluates the parser engine, development tools, and lexical acquisition mechanism. Limitations of the system and directions for future work are also discussed.

The goal of the work described in this thesis is to design and develop a set of tools to be used for developing RV parsers. The criteria that these tools must meet are:

(1) convenient editing of parsers, through the use of structured editors;

(2) automatic validity checking of identifiers, and suggestion of spelling corrections;

(3) identification of incorrect actions;

(4) automatic updating of all uses of an identifier when it is changed (i.e. renamed);

(5) protection against removal of referenced identifiers;

(6) the use of shorthand notations where useful;

(7) the ability to view the parser from various viewpoints;

(8) support for freely cross referencing information;

(9) the ability to view the parser at various levels of granularity;

(10) convenient testing and debugging of parsers, which involves:

- sentence testbeds;

- single stepping;
- access to internal structures;
- provision of relevant, useful information;
- convenient alteration of the acquisition mechanism's operational parameters.

These tools can be divided into three parts;

- an RV parser engine;
- parser definition editing tools; and
- parser debugging tools.

Each of these parts will be discussed and evaluated separately, followed by an evaluation of the system as a whole and the integration of the three parts.

9.1. Evaluating the Parser Engine

The best way to evaluate the parser engine is to use it, along with an appropriate parser specification, to parse sentences which test various aspects of the RV design. This is what has been done. Appendix B lists several parsers that were developed by Blank for testing and illustrating various aspects the parser engine's operation.

The development system was used to enter several parsers written by Blank. These performed successfully on sentences used by Blank with his implementation. The definition of these parsers (both syntax and lexicon) and the sentences each was tested with appear in Appendix B. Each of these parsers will now be discussed briefly.

9.1.1. Subject-Verb-Object. This parser (§B.1) is meant as a illustration of the basic RV components. It tests the functioning of the basic parser control algorithms, ternary vector operations, and lexical lookup.

9.1.2. Noun Phrases. This parser (§B.2) tests the ability to restrict productions to a specific type of clause, specificly noun phrases. It shows processing at the sentence level being suspended while the noun phrase is begin processed. This is accomplished by specifying the `-HEAD` feature in sentences level productions.

9.1.3. WH Questions. This parser (§B.3) adds support for WH questions (the lexicon only contains *who*). This is done by adding the concept of a gap, which is represented by the `GAP` feature. When the WH-question word is encountered, a gap is created. The gap has to be accounted for before the sentence can be successfully parsed. The gap can be accounted for by being used in place of a missing noun phrase. This tests the ability of the parser to handle discontinuous constraints.

9.1.4. Relative Clauses. This parser (§B.4) tests/shows the use of embedding, both right and center. For example, sentence (31) shows the use of both center embedding ("*hate men that eat quiche love pizza*", introduced by *who*) and, within that, right embedding ("*eat quiche*", introduced by *that*).

9.1.5. Subcategorization. This parser (§B.5) demonstrates how subcategorization is implemented using semi-lexical productions/categories. For example, consider the subcategorization of *believe*. Sentence (35) makes use of no semi-lexical categories, while (36) and (37) use `THAT_`, and `XO_`, respectively. Sentence (39) is rejected because *believe* does not subcategorize with `INF_` (which enable the `INF` production that handles infinitive clauses).

9.1.6. MorphoSyntactic Properties. This parser (§B.6) shows how morphosyntactic properties can be used to constrain production use. This constraint is implemented by `lexprop` actions (e.g. see the `QUES` production).

9.1.7. Boundary Registers. This parser (§B.7) exercises the boundary backtracking mechanism. Sentence (46) ("*The horse raced past the barn fell.*" which is equivalent to "*The horse* ***that was*** *raced past the barn fell.*") shows what happens when a boundary register is reused during a parse, particularly the Pred boundary at embedding level one (i.e. Pred1). This boundary is saved by the V production which is responsible for processing and consuming main predicates. In this sentence "*raced*" is initially recognised as the main predicate. Later in the parse "*fell*" is encountered. This is also recognised as the main predicate. Now there is a problem and the parser has to backtrack to search for an alternate interpretation that avoids this conflict. However, since "*fell*" was processed as a main predicate by the V production, the boundary Pred1 was reused. As a result the original contents of Pred1 (saved when "*raced*" was processed) is no longer available and the parser can not backtrack to that point so as to search for alternate interpretations of "*raced*".

The remaining test sentences show how the backtracking mechanism is used to find alternate interpretations.

9.2. Evaluating the Development Tools

The goal of this thesis is to design a set of interactive RV development tools which meet the criteria listed at the beginning of this chapter. This section will examine the degree of success achieved in meeting this goal.

The browsers present the syntax and lexicon in a much more structured way than the linear textual representation of Blank's system. Structure is imposed by initially providing a template for the developer to fill in, and later placing an existing definition in a template. This makes editing easier since the user always knows what information is expected, and in what order (satisfying criteria 1). This capability is not the same as a full language sensitive editor, in that there is no support for automatic formatting other than that done initially and there is no attempt made to keep the user from editing text other than that in the fields.

To make editing more convenient when working with feature vectors and property sets, shorthands are provided: macros (§7.4) and simplest descriptions (§7.5.1), respectively. This satisfies criteria 6. The macro editing facility is one way: when a vector is specified macros can be entered, but when an existing vector is displayed, there is no attempt made to replace its literal contents with macros. The simplest property description facility is bidirectional, however. Abbreviated descriptions can be entered and simplest descriptions are displayed for existing property sets. There is, however, no attempt made to preserve the actual property set specification: the system might generate a different specification than that which the user entered.

The information is cross-referenced, satisfying criteria 8. Not all possible cross references are maintained, just those that promised to be useful (e.g. finding all properties referenced by a paradigm is not useful since properties are fixed early in the development process, but finding all paradigms that reference a specified property is useful, so the latter is supported but not the former). This means that the developer can immediately call for a list of, for example, all productions referencing the selected boundary. This, along with the graphical semantic and lexical trie browsers, provides various views of the parser, satisfying criteria 7. For example, the developer can view all the productions, those that reference a specified boundary or property, and those that categorize a specified lexical entry. Also, they can view all lexical entries, those that are categorized by a specified production, and those that reference a specified paradigm. Also, a browser can be opened on a single production or lexical entry, which goes most of the way to satisfying criteria 9. The satisfaction of criteria 7 and 9 are naturally limited. There are very few ways to look at the parser specification other than different groupings (both in terms of the relationship between the members and group size) of productions, lexical entries, relations, and paradigms.

The browsers also perform consistency checks when information is accepted. This is done by automatic spell-checking of categories (production names), properties, boundaries, features (both ordering and semantic), and paradigm names. When one of these items is referenced and can not be found in the system, the user is presented with the most likely candidates if any are close enough to the specified item. One of these can be selected to automatically replace the offending item. This capability satisfies criteria 2. The validity of production actions is also verified. Actions are very simple and there are few of them. This makes it quite easy for the system to validate action keywords (e.g. `addprop`, `->`). Action parameters are all textual identifiers and are checked for validity at run-time, but each action has a fixed structure so the number of arguments is also checked at entry time. This functionality satisfies criteria 3.

Whenever an identifier (as listed above) is changed, all references to it are updated, i.e. any item that references the changed identifier is updated to refer to the new one (criteria 4). This includes embedded identifiers like feature names.

Finally, the user is not allowed to remove an item (property, boundary, etc.) which is referenced (criteria 5). The user can then use the cross referencing capabilities to find all references to the identifier and deal with them appropriately.

The interactiveness and modelessness of the tools make for a responsive debugging environment: when an error occurs during a parse, the browsers can be used to fix the error immediately. The debugger limits this capability in that it does not allow an operation to be restarted. The debugger almost completely satisfies criteria 10. Three things are missing:

(1) a parse cannot be stopped on demand;

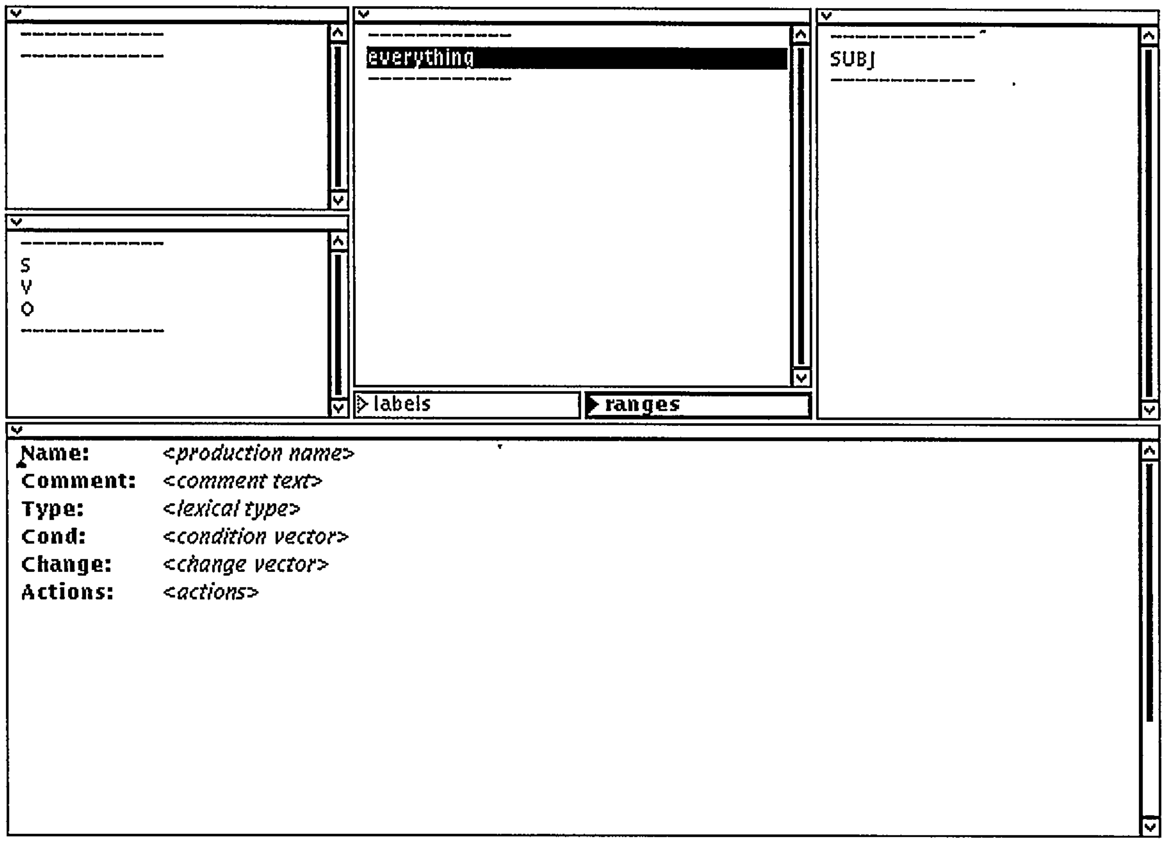
(2) a parser cannot be restarted at a specified point;

(3) all internal structures are displayable, but boundary registers, the register stack (*resume*), and grammatical roles are not editable (all others are).

The remainder of this section is a brief example of the operation of the development tools, specifically entering a production definition and subsequently debugging it. Figure 9.1 shows the syntax browser with a production definition template ready to be filled in. Figure 9.2 shows the the ADJ production being defined in the parser of §B.2. Note that the feature HEAD has been misspelled. Once complete, the definition is accepted by the developer. Since HEAD was misspelled, the system will notify the developer and ask if he want the system to find a correction. After the developer responds affirmatively, the system finds a single acceptable correction: HEAD. The developer is asked to confirm the correction. This is shown in Figure 9.3. If there were more than a single candidate, the system would provide a menu of the alternatives. Once the correction has been accepted, the production definition is redisplayed, without the omitted optional fields. Figure 9.4 shows this.

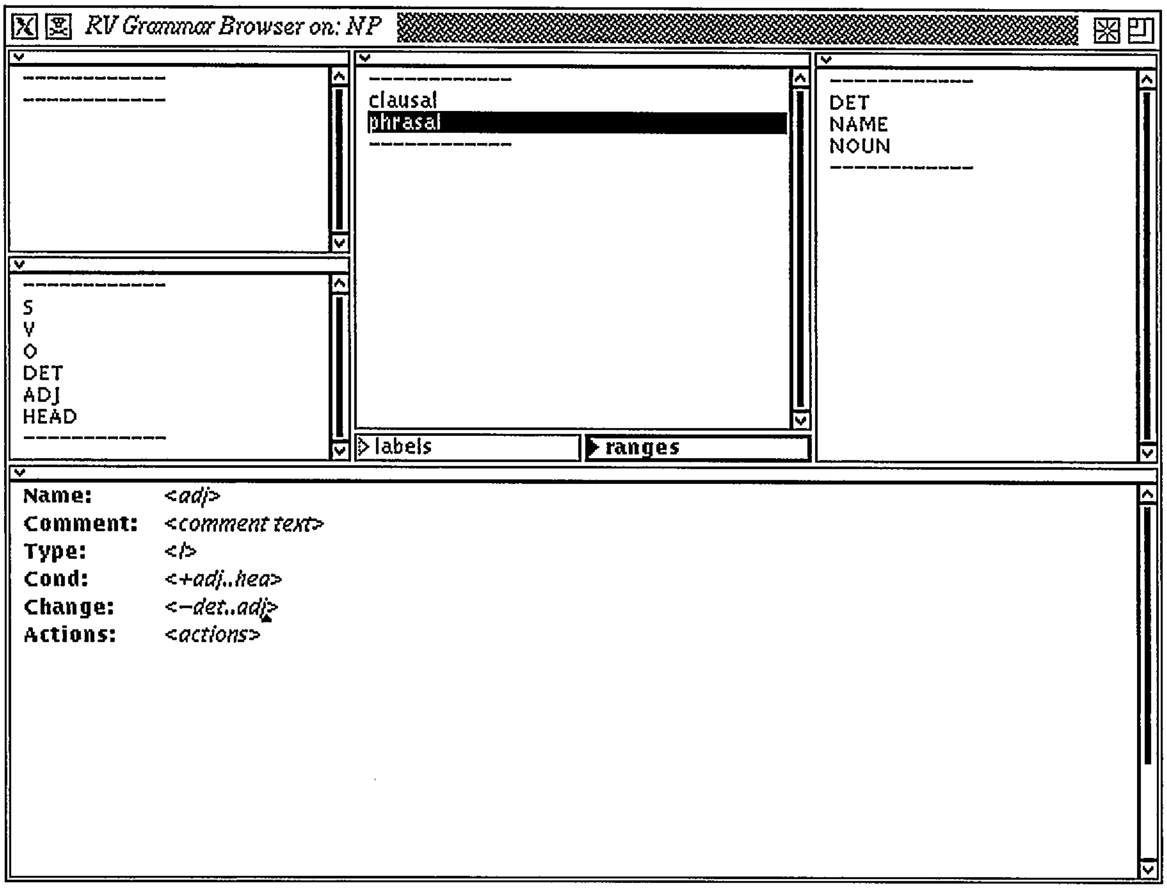
In the traditional RV development system, production definitions are placed into a linear text file using a text editor. No checking is performed until the definition is converted into a format usable by the RV parser engine. In contrast to this, RV-Tools provides a skeletal production definition to be completed, and immediate feedback that aids in correcting common errors before they can create a problem.

I now turn to the parser debugging and testing facilities. As an example, assume that the developer wishes to extend the parser to allow multiple adjectives. By enabling information regarding the reason productions are not used, it can be seen



Name:	<production name>
Comment:	<comment text>
Type:	<lexical type>
Cond:	<condition vector>
Change:	<change vector>
Actions:	<actions>

FIGURE 9.1. Production Definition Template



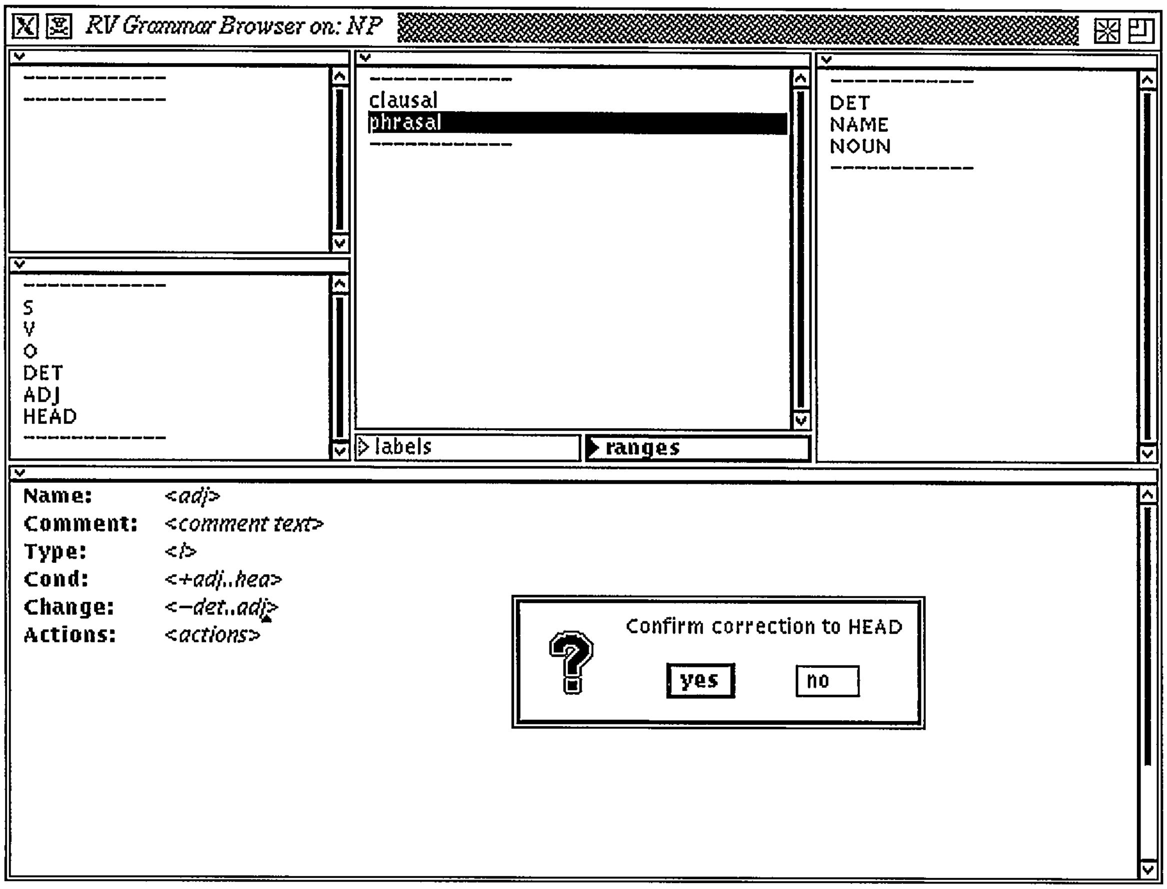
```
S
V
O
DET
ADJ
HEAD
```

```
clausal
phrasal
```

```
DET
NAME
NOUN
```

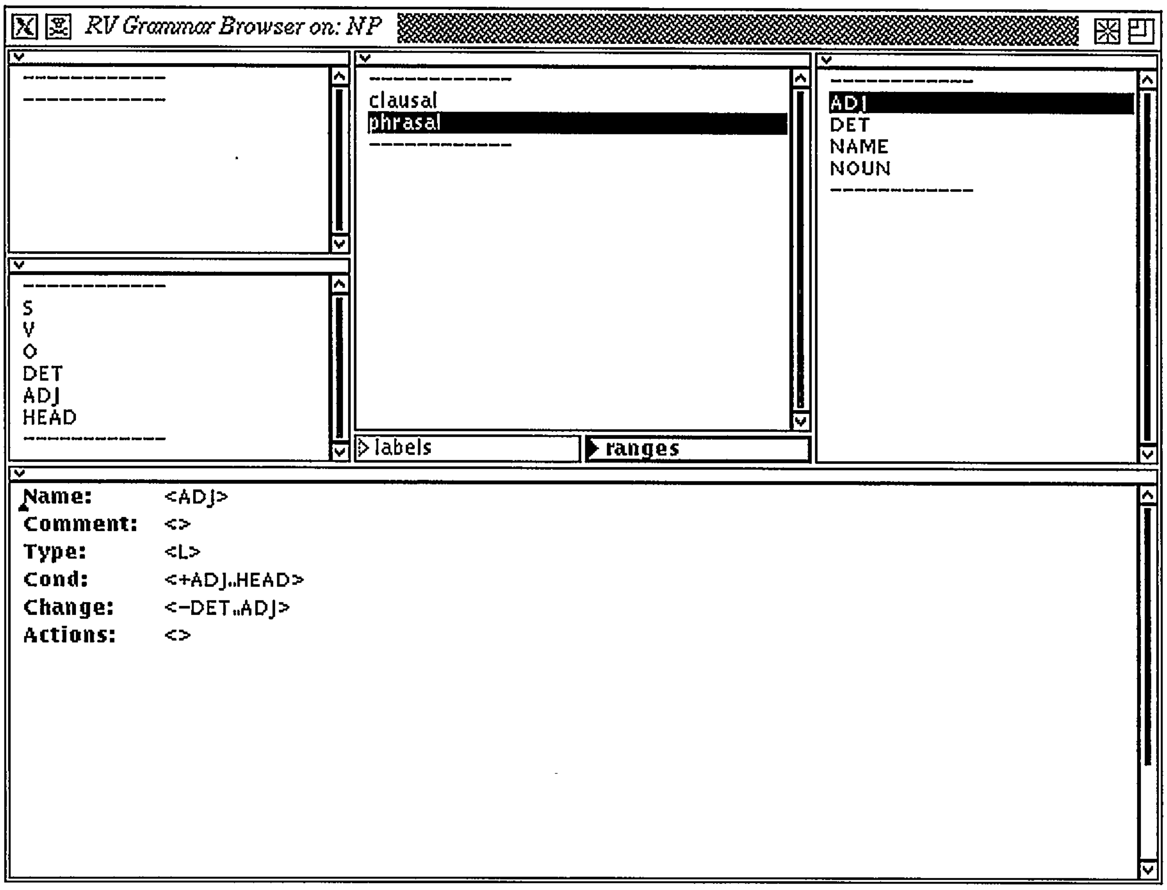
```
Name:      <adj>
Comment:   <comment text>
Type:      </>
Cond:      <+adj..hea>
Change:    <-det..adj>
Actions:   <actions>
```

FIGURE 9.2. Definition of the Production ADJ



Name: <adj>
Comment: <comment text>
Type: <i>
Cond: <+adj..hea>
Change: <–det..adj>
Actions: <actions>

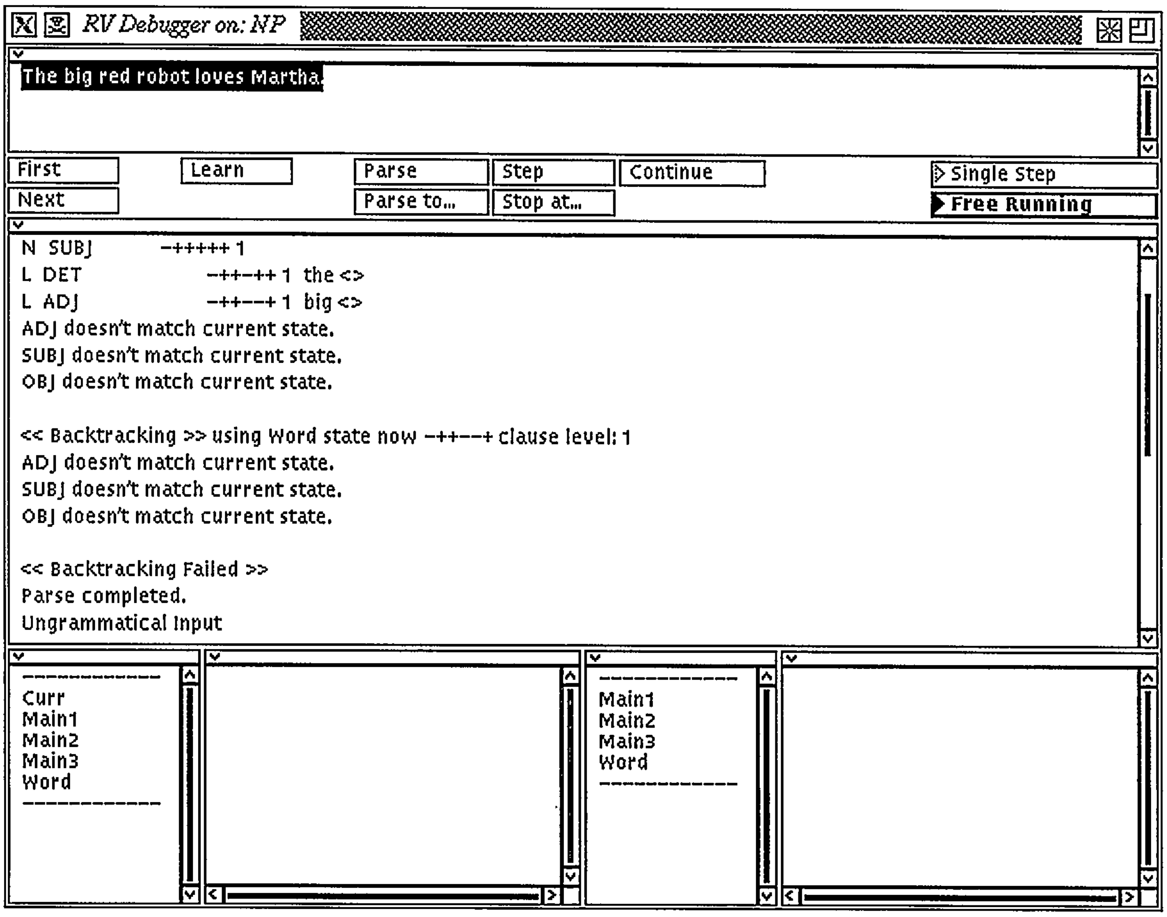
FIGURE 9.3. Spell Checking Mechanism at Work



```
Name:      <ADJ>
Comment:   <>
Type:      <L>
Cond:      <+ADJ..HEAD>
Change:    <-DET..ADJ>
Actions:   <>
```

FIGURE 9.4. Final Definition of ADJ

that the condition vector of the ADJ production does not match the parser state vector once ADJ has been used once. Figure 9.5 shows this.



```
The big red robot loves Martha.
```

```
N SUBJ        -+++++ 1
L DET              -++-++ 1  the <>
L ADJ              -++--+ 1  big <>
ADJ doesn't match current state.
SUBJ doesn't match current state.
OBJ doesn't match current state.

<< Backtracking >> using Word state now -++--+ clause level: 1
ADJ doesn't match current state.
SUBJ doesn't match current state.
OBJ doesn't match current state.

<< Backtracking Failed >>
Parse completed.
Ungrammatical Input
```

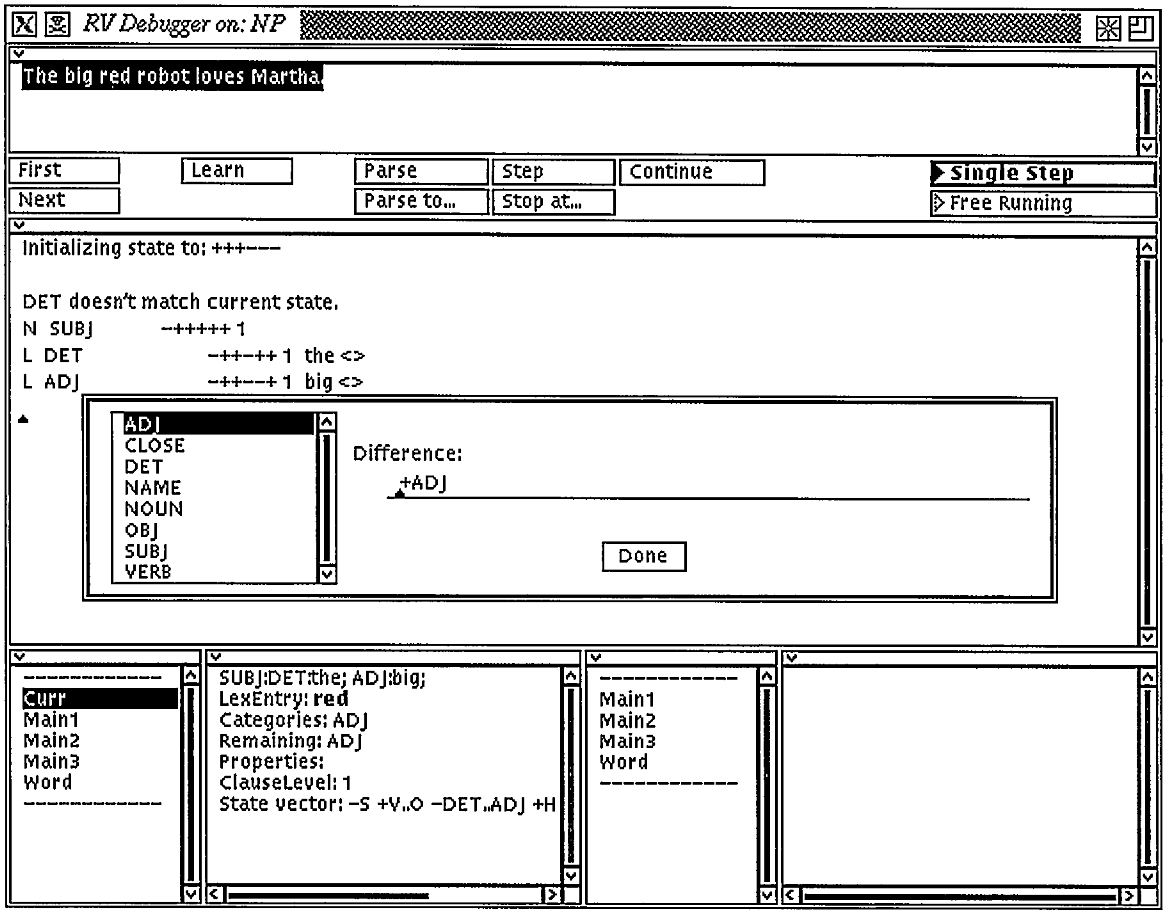
```
------------
Curr
Main1
Main2
Main3
Word
------------
```

```
------------
Main1
Main2
Main3
Word
------------
```

FIGURE 9.5. Failure of ADJ the second time.

By having the parser stop after ADJ is used the developer can examine the parser state and compare it with ADJ's condition vector. Figure 9.6 shows this being done. The developer can see that ADJ requires the ordering feature value +ADJ. By examining the definition of ADJ (see Figure 9.4), it is found that -ADJ is set by ADJ's change vector. It is decided that +ADJ should be removed from ADJ's condition vector. The

-ADJ value is left in the change vector so as to provide an indication that adjectives were encountered. Multiple adjectives can now be processed, as shown in Figure 9.7.



```
The big red robot loves Martha.
```

```
Initializing state to: +++---

DET doesn't match current state.
N  SUBJ       -+++++ 1
L  DET             -++-++ 1  the <>
L  ADJ             -++--+ 1  big <>
```

```
Difference:
  +ADJ
```

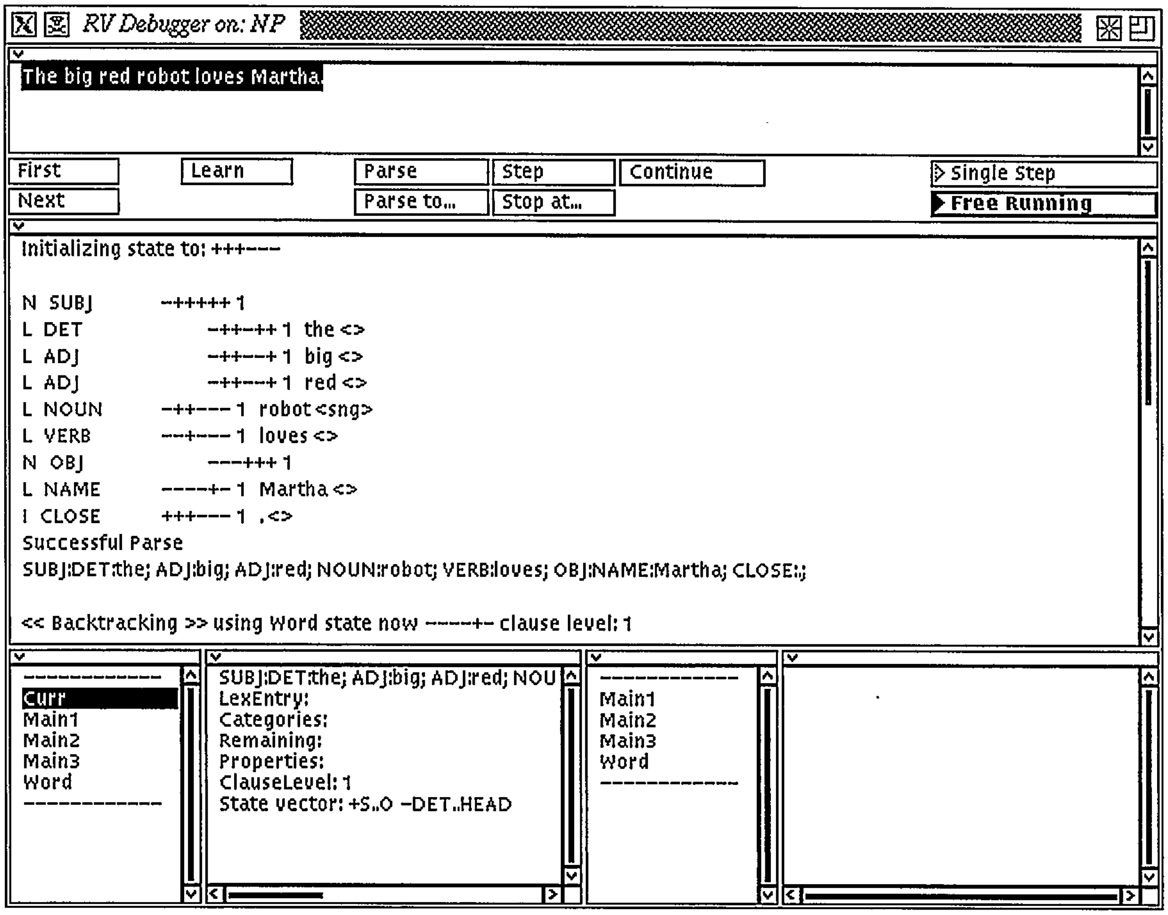
```
------------
Curr
Main1
Main2
Main3
Word
------------
```

```
SUBJ:DET:the; ADJ:big;
LexEntry: red
Categories: ADJ
Remaining: ADJ
Properties:
ClauseLevel: 1
State vector: -S +V..O -DET..ADJ +H
```

```
------------
Main1
Main2
Main3
Word
------------
```

FIGURE 9.6. Comparing ADJ with the Parser State.

As a demonstration of the tools, I developed a parser that accepted simple commands relating to UNIX file management (Appendix D contains the parser definition). The parser communicated with a small application (see source code in Appendix E) via a UNIX socket. The application accepted a command from the user (e.g. "*Text files end with txt.*", "*List all the text files.*") and passed it to the RV parser. The parser converted the sentence into a semantic structure and returned the structure



```
The big red robot loves Martha.
```

```
Initializing state to: +++---

N  SUBJ      -+++++ 1
L  DET           -++-++ 1  the <>
L  ADJ           -++--+ 1  big <>
L  ADJ           -++--+ 1  red <>
L  NOUN      -++--- 1  robot <sng>
L  VERB      --+--- 1  loves <>
N  OBJ           ---+++ 1
L  NAME      ----+- 1  Martha <>
I  CLOSE     +++--- 1  .<>
Successful Parse
SUBJ:DET:the; ADJ:big; ADJ:red; NOUN:robot; VERB:loves; OBJ:NAME:Martha; CLOSE:.;

<< Backtracking >> using Word state now ----+- clause level: 1
```

```
------------
Curr
Main1
Main2
Main3
Word
------------
```

```
SUBJ:DET:the; ADJ:big; ADJ:red; NOU
LexEntry:
Categories:
Remaining:
Properties:
ClauseLevel: 1
State vector: +S..O -DET..HEAD
```

```
------------
Main1
Main2
Main3
Word
------------
```

FIGURE 9.7. Parsing Multiple Adjectives.

to the application. The application then built a UNIX command string based on the information in the structure. The command string was then executed.

Also, in the course of entering Blank's parsers some discrepancies were encountered. These consisted mostly of missing ordering feature settings in condition and change vectors. Whether these were omissions of Blanks, or symptomatic of slight differences between my engine and his, they provided a chance to rigorously test the debugger as well as the browsers.

9.3. Evaluating the Lexical Acquisition Mechanism

To test the lexical acquisition mechanism, I used a parser with a lexicon containing 326 words and 37 paradigms, shown in §C.1. Six words were removed from the lexicon: "address," "follow," "instruction," "intelligence," "process," and "program." The parser was then used to parse the 40 sentences from Tomita's testbed drawn from computer science texts (Tomita, 1987, Appendix G) (as distributed by Blank, see §C.2).

The spelling and categorization of "process" was learned as correctly as possible: two categories remained which could not be resolved by examining ending sets, the incorrect category would eventually be removed by the aging mechanism. The spelling of "intelligence" was left with two alternatives, and the categorization needed significant pruning. The remaining four words were learned to varying degrees: four extraneous categories for two of them, six for one, and thirteen for one. In all cases, the category aging mechanism would eventually converge the categorizations to the correct ones (i.e. the categorizations that prove useful).

9.4. Implementation Status

I have implemented the development tools, RV parser engine, and lexical acquisition mechanism described in this thesis using Objectworks\Smalltalk-80 release 4.0 on a Sun 3/50 workstation. One reason for using Smalltalk-80 was its portability. This was tested by running the RV development system, without modification or recompilation, on SparcStations (the only other St-80 r4.0 platform that was available).

9.5. Limitations and Future Directions

There are several directions in which future work could progress. All of these require a considerable amount of work, and in some cases major reworking of the existing system. They are described in the following paragraphs.

Complete the implementation of the RV engine. The only feature of Blank's definition of RV that is not implemented is support for discontinuous idioms (Blank & Kasson, 1989). In the system described here, multi-word idioms must be continuous. This means that the idiomatic senses of sentences (18) and (20) are recognisable but those of sentences (19) and (21) are not. Future work should address the lack of support for discontinuous idioms.

(18) He *kicked the bucket.*

(19) He *kicked the* proverbial *bucket.*

(20) I feel like *throwing in the towel.*

(21) Once *the towel* has been *thrown in*, there is no going back.

Extension of the acquisition mechanism. The lexical acquisition mechanism is very simple. It needs to be extended to handle more than just suffixes and made more robust. Another avenue would be to explore grammar acquisition.

Generation capabilities. The system as described here contains only a parser. An RV generation mechanism would make the system more useful. Morphological generation is quite simple to add: given a lexical entry and a set of properties, it is simply a matter of walking up the trie to the root, selecting the appropriate string from each paradigm on the way (literal nodes are used verbatim). Syntactic generation presents more of a problem. What would be desired is a system that could take a semantic structure and generate a textual sentence for it. It would be preferable if the generator could make use of the same productions[1] that the parser uses. Approaches to this have not been explored to my knowledge.

Extension of the debugger's functionality. There are several deficiencies in the design of the debugger:

- There is no way to interrupt the parser when it is operating in free-running mode;
- There is no way to examine the backtracking stack and set the parser to any state stored on it;
- There is no way to edit the boundary register and grammatical role that are displayed;
- There is very little in the way of support for post-mortem debugging of a parse. There needs to be work done on adding facilities for examining a parse after it has completed, successfully or otherwise. At the moment all that is available to the developer is the contents of the trace view. What is needed is a way to browse the tree of decisions made during the parse, especially branches that

[1]The structure of productions will likely have to be extended to include generation related information.

led to a dead end. A useful feature would be the ability to restart the parser at a specific point.

Better interaction in the lexical trie browsers. The capabilities of the trie browser are very limited. Capabilities should be added allowing individual nodes to be added, deleted, and moved. The browser should allow complete subtries to be moved as well as deleted.

Improve the formalism. In the current definition of RV, non-lexical productions are order sensitive: When non-lexical productions are checked for applicability, the first to pass all the tests (see §2.1.1) is used. This creates another, non-obvious issue that parser developers must deal with: parser operation is dependent on the ordering of non-lexical productions. The possibility of modifying the RV formalism to remove this should be explored. One approach to this would be to design a generality metric for productions, and enforce an ordering based on that. This would guarantee that more specific productions would be checked before more general ones.

Development of a runtime engine. A development system is of little use without a delivery mechanism. To this end, future work on this system should include the design and development of a runtime RV engine. The design of this engine would be significantly different than that of the development engine. The runtime engine would have to be designed with efficiency and economy of storage as the main criteria. This redesign applies to the fundamental data structures used by the engine as well.

9.6. Summary

This chapter provided an evaluation of the RV parser engine, the development tools, and the lexical acquisition mechanism. All design requirements were met, most to the full extent. The system has been implemented using Smalltalk-80 on Sun 3 and SparcStation platforms. This work is the first to try to develop a convenient, interactive environment for developing RV parsers. As such, there are several areas of future work that have been identified. Foremost among these are the need for more extensive debugging facilities and the improvement of some facets of the RV formalism, particularly the order sensitivity of non-lexical productions.

Chapter 10
Conclusions

This thesis presents the design and implementation of an RV parser and a set of parser development tools. Traditional approaches to parser development employ an edit–compile–test paradigm. This thesis presents a set of RV parser development tools that provide an interactive, modeless environment for constructing and testing RV parsers. These tools satisfied the criteria put forth in Chapter 1, specifically:

- convenient editing of parsers, through the use of structured editors;
- automatic validity checking of identifiers, and suggestion of spelling corrections;
- identification of incorrect actions;
- automatic updating of all uses of an identifier when it is changed (i.e. renamed);
- protection against removal of referenced identifiers;
- the use of shorthand notations where useful;
- the ability to view the parser from various viewpoints;
- support for freely cross referencing information;
- the ability to view the parser at various levels of granularity;
- convenient testing and debugging of parsers, which involves:
  - sentence testbeds;

- single stepping;
- access to internal structures;
- provision of relevant, useful information;
- convenient alteration of the acquisition mechanism's operational parameters.

The first contribution of this thesis is the development of an RV parser engine in Smalltalk-80. This is a valid contribution because the replication of scientific work is an important endeavor.

The second contribution is the design and implementation of a set of interactive RV development tools. Two general browsers were developed, one each for syntax and lexicon related information. In addition to these, several more specific browsers were developed which present more focussed information. Also developed was a debugger which allows the user to step through a parse and examine the parser's internal structures at any point. Together, these tools provide a responsive, productive environment for developing RV parsers. This is a significant improvement over the traditional RV development systems, similar to the improvement of the Smalltalk programming environment over traditional programming environments (e.g. C programming using separate editor, compiler, and debugger).

A final contribution of this thesis is the initial attempt at developing an automated mechanism for vocabulary acquisition. The value of this line of research is that it has the potential of removing the requirement that the parser developer create a *complete* lexicon.

It was stated in Chapter 1 that natural language understanding is an important area of research in artificial intelligence. The work described in this thesis contributes to this area of scientific endeavor by providing a set of tools that researchers can use when they are building RV parsers. The system is open-ended enough to allow easy modification. This enables researchers to modify the tools to accommodate extensions to the RV formalism itself. RV has the potential of being useful in relating to natural language parsing, and RV-Tools makes it easier to work with RV.

This thesis began by briefly introducing RV, pointing out its advantages and disadvantages. The most important of the latter was its awkwardness for parser developers. My approach to countering this disadvantage was described, specifically the design and implementation of a set of development tools that make it easier and more convenient to develop RV parsers.

Chapter 2 presented a detailed description of the design and operation of the RV formalism.

This was followed in Chapter 3 by a description of the two existing, documented systems which support development of RV parsers: Blank's and Reed's. RV-Tools' philosophical differences with these were described briefly:

- Blank's system has a traditional edit–compile–test development paradigm whereas RV-Tools provides various views of the systems under development, is interactive, and allows modification of the parser at any time;
- Reed's system was designed to use RV as an efficient mechanism for implementing phrase structure grammars, and does not address direct development of RV parsers. RV-Tools, on the other hand, is targeted at the parser developer who is working directly with the RV formalism.

Finally, related work in the area of automated lexical acquisition was briefly described. Most of the systems examined strive to be psychologically valid, while those that are primarily pragmatic are generally not interactive or incremental. This contrasts to my system, described in Chapter 8, which is designed for interactive, incremental operation.

Chapter 4 very briefly described object-oriented programming as it applies to Smalltalk-80. This chapter also introduced the Model-View-Controller paradigm that forms the basis of the implementation of the RV development environment.

The next chapter documents the extensions I have made to the RV formalism defined in Chapter 2:

- the **diff:** vector operation and the '@' element value;
- a lexical acquisition mechanism and support for it, both the parser's handling of unknown words and the addition of a new type of node for the lexical trie; and
- an extension of the production selection algorithm to use information about which lexical productions are associated with each semi-lexical production, in order to reduce the number of impossible production sequences that are considered.

This is followed by a chapter describing my implementation of the RV parser engine, including the lexical and semantic components.

Chapter 7 began by presenting the design requirements for a set of RV development tools. Put concisely, these requirements are:

- The tools should allow and encourage interactive and incremental development;
- Editing should be convenient for the developer, minimizing the sources of operator error (e.g. spelling errors);
- The system under development should be able to be viewed from various organizational standpoints and at various levels of granularity;

- Adequate information should be provided (in controllable detail) to the developer during parser testing.

The remainder of the chapter described the design of a set tools that meet the stipulated requirements. The core of the system consists of the syntax browser, lexicon browser, and debugger. In addition to these core tools are several specialized browsers, including:

- browse a single production;
- browse all production which refer to a specified boundary;
- browse all lexical entries which are categorized by a specified production.

Chapter 8 described a method for acquiring vocabulary on-the-fly in RV systems. The learner incorporates new words into the lexicon and trie, and assigns an initial set of categories. As more forms of the word are encountered, a paradigm will eventually be selected to account for the set of suffixes seen, and a more precise categorization will be assigned as useful categories are repeatedly seen. The learning system runs together with the parser, so that vocabulary learning does not require extra effort from the user.

Chapter 9 provided an evaluation of the RV parser engine, the development tools, and the lexical acquisition mechanism. All design requirements were met, most to the full extent. The system has been implemented using Smalltalk-80 on Sun 3 and SparcStation platforms. This work is the first to try to develop a convenient, interactive environment for developing RV parsers. As such, there are several areas of future work that have been identified. Foremost among these are the need for more extensive debugging facilities and the improvement of some facets of the RV formalism, particularly the order sensitivity of non-lexical productions.

In conclusion, RV has proven to be a powerful, efficient processing formalism. Its main problem lies in its low level of abstraction. The work described in this thesis is an attempt to make it easier to develop RV parsers by providing a set of interactive tools. These tools allow non-linear browsing/editing of parsers, immediate error feedback, as well as interactive debugging. These tools do make it easier to create RV parsers in that they provide low-level support for working with productions, lexical entries, and other components of a parser. What is still missing is a tool to aid with the interrelationships between productions, i.e. something to aid with constructing *the big picture* of a parser. Even so, the tools described herein go a long way to making the development of RV parsers faster and easier.

References

Aho, A., Sethi, R., & Ullman, J. (1986). *Compilers. Principles, Techniques and Tools.* Addison-Wesley.

Astels, D. R. (1991). An object-oriented implementation of a register vector parser. Honours thesis, Acadia University.

Astels, D. R. & MacDonald, B. A. (1993). Learning vocabulary for a register vector parser. In *Proc. of the* 1st *Conference of the Pacific Association for Computational Linguistics*, pages 92–101, Vancouver. PACLING, Simon Fraser University.

Bar, A. & Feigenbaum, E. A. (1981). *The Handbook of Artificial Intelligence*, volume 1. Los Altos, CA: Morgan Kaufmann.

Blank, G. D. (1985). A new kind of finite-state automaton: Register vector grammar. In *Proceedings of The Ninth International Joint Conference on Artificial Intelligence*, pages 749–755.

Blank, G. D. (1989a). A finite and real-time processor for natural language. *Communications of the ACM*, *32*(10), 1174–1189.

Blank, G. D. (1989b). *Register Vector Grammar Version 3.2.1 Tutorial and User Manual.* CSEE Dept., Packard Lab 19, Lehigh University.

Blank, G. D. (1991a). *Register Vector Grammar Version 3.9.1 Tutorial and User Manual.* CSEE Dept., Packard Lab 19, Lehigh University.

Blank, G. D. (1991b). Semantic interpretation and agreement in linear time. In *Proc. of the 4th University of New Brunsick Symposium in Artificial Intelligence.*

Blank, G. D. et al. (1992). Rvg source code. Objective-C Source code for RVG.

Blank, G. D. & Kasson, M. S. (1989). Inflections and idioms: Lexical access using tries. This paper is a synopsis of an earlier Master's Thesis by Mr. Kasson.

Blank, G. D. & Labuda, E. J. (1991). Agreement and directionality. This paper is a synopsis of an earlier Master's Thesis by Mr. Labuda.

Blank, G. D. & Owens, C. J. (1990). Subcategorization in register vector grammar. In *SPIE Applications of Artificial Intelligence, VIII*, Orlando, FL. This paper is a synopsis of an earlier Master's Thesis by Ms. Owens.

Carrithers, C. & Bever, T. (1984). Eye-fixation patterns during reading confirm theories of language comprehension. *Cog. Sci.*, *8*(2), 157–172.

Chomsky, N. (1957). *Syntactic Structures.* The Hague: Mouton.

Church, K. (1982). On memory limitations in natural language processing. IU Linguistics Club, Bloomington, Ind.

Garrett, M. & Bever, T. (1970). The perceptual segmentation of sentences. In Bever, T. & Weksel, W. (Eds.), *The Structure and Psychology of Language.* Holt, Rinehart & Winston.

Goldberg, A. (1990). Information models, views, and controllers. *Dr. Dobb's Journal.* Reprinted by ParcPlace.

Harris, L. R. (1977). A system for primative natural language acquisition. *Int'l Jnl of Man-machine Studies*, *9*, 153–206.

Hastings, P. M., Lytinen, S. L., & Lindsay, R. K. (1991). Learning words from context. In Birnbaum, L. A. & Collins, G. C. (Eds.), *Machine Learning, Proc. of the 8th Int. Workshop*, pages 55–59. Morgan Kaufmann.

Kazman, R. (1991). Babel: A psychologically plausible cross-linguistic model of lexical and syntactic acquisition. In Birnbaum, L. A. & Collins, G. C. (Eds.), *Machine Learning, Proc. of the 8^{th} Int. Workshop*, pages 75–79. Morgan Kaufmann.

Kogut, P. (1992). Acquiring lexical semantics from wordnet and text corpora. Draft of a paper.

LaLonde, W. & Pugh, J. R. (1990a). *Inside Smalltalk*, volume 1. Prentice-Hall, Inc.

LaLonde, W. & Pugh, J. R. (1990b). *Inside Smalltalk*, volume 2. Prentice-Hall, Inc.

Miller, G. & Chomsky, N. (1963). Finitary models of language users. In et al., R. L. (Ed.), *Handbook of Mathematical Psychology.* New York: Wiley.

Reed, J. H. (1987). An efficient contex-free parsing algorithm based on register vector grammars. In *Proceedings of The Third Annual Expert Systems in Government Conference*, pages 34–40.

Reed, J. H. (1989). Compiling phrase structure grammar rules into register vector grammar. In *Proceedings of The Fifth Annual AI Systems in Government Conference*, pages 244–249.

Systems, P. (1990). *Objectworks\Smalltalk Release 4 User's Guide.* Sunnyvale, CA.

Tomita, M. (1987). *Efficient Parsing for Natural Language.* Norwell, Mass.: Kluwer Academic Publishers.

Wolczko, M. (1992). `grapher`. Manchester Smalltalk Goodies Archive. Graph layout and display classes.

Woods, W. A. (1970). Transition network grammars for natural language analysis. *Communications of the ACM*, *13*(10), 591–606.

Appendix A

Comparison with Finite State Machines

RV is an improvement on finite state machines because it is functionally more powerful. In a simple finite state machine, **match** is symbol identity and **change** does a complete state replacement. In RV, however, an operation is allowed to involve several items at once. This allows RV to be more compact than an equivalent finite state machine. Figures A.1 and A.2 and Tables A.1 and A.2 constitute an example for parsing simple Subject-Verb-Object languages used by Blank (Blank, 1989). The advantage of RV can be seen. When just one constraint is relaxed (the requirement that the subject precede the verb) the size and complexity of the finite state machine increases by two nodes (a 40% increase) and four transitions[1] (an 80% increase), while the size of the RV parser remains constant. The only change is that of the first element in the *condition* vector of the VERB production from - to ?.

[1]Noted by heavier lines in Figure A.2.

cat	condition	change
SUBJ	+??	-??
VERB	-+?	?-?
OBJ	?-+	??-
CLOSE	--?	+++

Table A.1. The RV Productions for Subject-Verb-Object Language

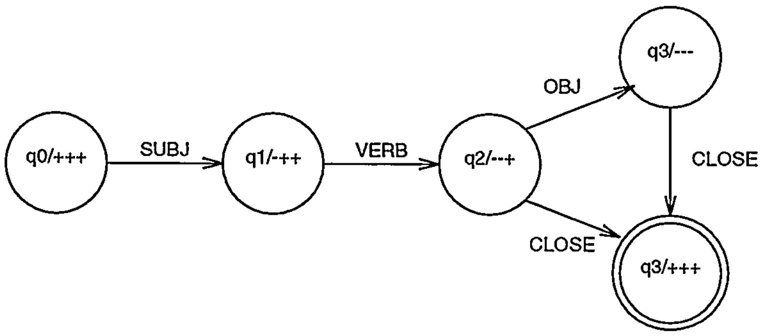


FIGURE A.1. The Finite State Machine Transition Diagram for the Subject-Verb-Object Language

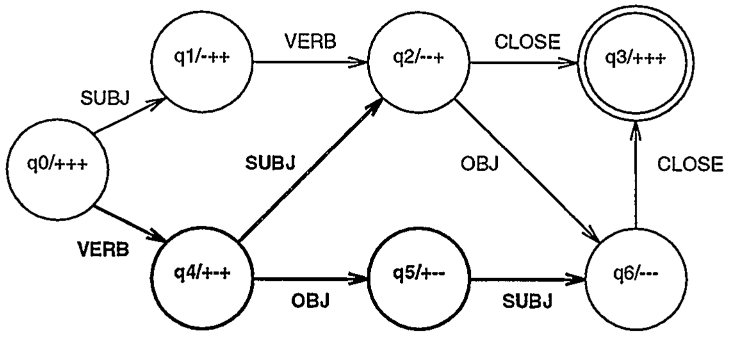


FIGURE A.2. The Finite State Machine Transition Diagram for Partially Free-Order Language

cat	condition	change
SUBJ	+??	-??
VERB	?+?	?-?
OBJ	?-+	??-
CLOSE	--?	+++

TABLE A.2. The RV Productions for Partially Free-Order Language

In both the finite state machine and the RV implementations, all resources are preallocated and do not grow during the operation of the machine. This is due to the fact that RV has only one production per category. The effect of this property becomes more significant as the grammar size increases. The result of this is that an RV implementation will be more efficient than the equivalent finite state machine in terms of both space and recognition time.

The overall efficiency of RV is a result of two characteristics that are not present in functionally more primitive mechanisms such as finite state machines: multiplicity and masking.

Multiplicity refers to the capability of one operation to test multiple constraints simultaneously or have multiple effects. This capability is inherent in the ternary feature vectors that are the central component of RV. In the above example `OBJ` will only fire when the second feature is - and the third is +. The *change* vector of the `CLOSE` production gives an example of the multiplicity of effect. When `CLOSE` fires, it will set all features to +.

Masking (allowing constraints to be unaffected or ignored by an operation) is made possible by the third allowable element value in ternary vectors: ?. This value matches anything and changes nothing, thus it can be used to let constraints pass from state to state. An example of this is present in the `SUBJ` production. This production will fire when the first feature is +. It ignores all other features. When it fires, it changes only the first feature, leaving the others as they were.

Appendix B

Test Parsers

This appendix contains the parsers that the RV development environment has been used to implement. These parsers appear in various works by Blank.

B.1. Subject-Verb-Object

Ordering Features: S V O

Productions:

```
CLOSE  I   cond -S..V    change +S..O
OBJ    L   cond -V +O    change -O
SUBJ   L   cond +S       change -S
VERB   L   cond -S +V    change -V
```

Lexicon:

```
.        cat CLOSE      morph '.'
George   cat SUBJ OBJ   morph 'george'
loves    cat VERB       morph 'loves'
Martha   cat SUBJ OBJ   morph 'martha'
```

(22) *George loves Martha.*

```
Initializing state to: +++

LSUBJ   -++ 1 George <>
LVERB   --+ 1 loves <>
LOBJ    --- 1 Martha <>
I CLOSE +++ 1 . <>
Successful Parse
SUBJ:George; VERB:loves; OBJ:Martha; CLOSE:.;

<< Backtracking >> using Word state now --- clause level: 1

<< Backtracking Failed >>
Parse completed.
```

(23) *Martha loves.*

```
Initializing state to: +++

LSUBJ   -++ 1 Martha <>
LVERB  --+ 1 loves <>
I CLOSE +++ 1 . <>
Successful Parse
SUBJ:Martha; VERB:loves; CLOSE:.;

<< Backtracking >> using Word state now --+ clause level: 1

<< Backtracking Failed >>
Parse completed.
```

(24) **George Martha.*

```
Initializing state to: +++

LSUBJ   -++ 1 George <>

<< Backtracking >> using Word state now -++ clause level: 1

<< Backtracking Failed >>
Parse completed.
Ungrammatical Input
```

B.2. Noun Phrases

Ordering Features: S V O DET ADJ HEAD
Properties: pl sng
Productions:

```
clausal
CLOSE  I  cond -S..V -HEAD     change +S..O -DET..HEAD
OBJ    N  cond -V +O -HEAD     change -O +DET..HEAD
SUBJ   N  cond +S -HEAD        change -S +DET..HEAD
VERB   L  cond -S +V -HEAD     change -V
phrasal
ADJ    L  cond +ADJ..HEAD      change -DET..ADJ
DET    L  cond +DET..ADJ       change -DET
NAME   L  cond +DET +HEAD      change -DET -HEAD
NOUN   L  cond +HEAD           change -DET -HEAD
```

Non-lexical Ordering: SUBJ OBJ
Paradigms:

```
BED  s <pl>
     $ <sng>
```

Lexicon:

```
.        cat CLOSE  morph '.'
a        cat DET    morph 'a'
big      cat ADJ    morph 'big'
George   cat NAME   morph 'george'
loves    cat VERB   morph 'loves'
Martha   cat NAME   morph 'martha'
red      cat ADJ    morph 'red'
robot    cat NOUN   morph 'robot_BED_'
the      cat DET    morph 'the'
```

(25) *George loves Martha.*

```
Initializing state to: +++---

NSUBJ   -+++++ 1
LNAME   -++-+- 1 George <>
LVERB   --+-+- 1 loves <>
NOBJ    ---+++ 1
LNAME   ----+- 1 Martha <>
I CLOSE +++--- 1 . <>
Successful Parse
SUBJ:NAME:George; VERB:loves; OBJ:NAME:Martha; CLOSE:.;

<< Backtracking >> using Word state now ----+- clause level: 1

<< Backtracking Failed >>
Parse completed.
```

(26) *George loves the red robot.*

```
Initializing state to: +++---

NSUBJ   -+++++ 1
LNAME   -++-+- 1 George <>
LVERB   --+-+- 1 loves <>
NOBJ    ---+++ 1
LDET    ----++ 1 the <>
LADJ    -----+ 1 red <>
LNOUN   ------ 1 robot <sng>
I CLOSE +++--- 1 . <>
Successful Parse
SUBJ:NAME:George; VERB:loves; OBJ:DET:the; ADJ:red; NOUN:robot; CLOSE:.;

<< Backtracking >> using Word state now ------ clause level: 1

<< Backtracking Failed >>
Parse completed.
```

(27) *A robot loves Martha.*

```
Initializing state to: +++---

NSUBJ   -+++++ 1
```

```
LDET    -++-++ 1 a <>
LNOUN  -++-+- 1 robot <sng>
LVERB  --+-+- 1 loves <>
NOBJ    ---+++ 1
LNAME  ----+- 1 Martha <>
I CLOSE +++--- 1 . <>
Successful Parse
SUBJ:DET:a; NOUN:robot; VERB:loves; OBJ:NAME:Martha; CLOSE:.;

<< Backtracking >> using Word state now ----+- clause level: 1

<< Backtracking Failed >>
Parse completed.
```

B.3. WH Questions

Ordering Features: S V O AUX GAP DET HEAD

Productions:

clausal

CLOSE	I	cond -S..V -GAP -HEAD	change +S..AUX -GAP..HEAD
OBJ	N	cond -V +O -HEAD	change -O +DET..HEAD
SUBJ	N	cond +S -HEAD	change -S +DET..HEAD
VERB	L	cond -S +V -HEAD	change -V -AUX

phrasal

DET	L	cond +DET	change -DET
NAME	L	cond +DET..HEAD	change -DET..HEAD
NOUN	L	cond +HEAD	change -DET..HEAD

wh

NGAP	N	cond +GAP..HEAD	change -GAP..HEAD
QUES	L	cond +S +AUX -HEAD	change -AUX
WH	L	cond +S -GAP -HEAD	change +GAP

Non-lexical Ordering: SUBJ OBJ NGAP

Lexicon

names

George	cat NAME	morph 'george'
Martha	cat NAME	morph 'martha'
Pam	cat NAME	morph 'pam'
Pamela	cat NAME	morph 'pamela'

nouns

men	cat NOUN	morph 'men'
quiche	cat NOUN	morph 'quiche'
robot	cat NOUN	morph 'robot'

determiners

a	cat DET	morph 'a'
the	cat DET	morph 'the'

verbs

love	cat VERB	morph 'love'
loves	cat VERB	morph 'loves'
sighed	cat VERB	morph 'sighed'

wh words

does	cat QUES	morph 'does'
who	cat WH	morph 'who'

punctuation

.	cat CLOSE	morph '.'
?	cat CLOSE	morph '?'

(28) *Who loves Pamela?*

```
Initializing state to: ++++---

LWH     +++++-- 1 who <>
NSUBJ   -++++++ 1
NNGAP   -+++--- 1
LVERB   --+---- 1 loves <>
NOBJ    -----++ 1
LNAME   ------- 1 Pamela <>
I CLOSE ++++--- 1 ? <>
Successful Parse
WH:who; SUBJ:NGAP:VERB:loves; OBJ:NAME:Pamela; CLOSE:?;

<< Backtracking >> using Word state now ------- clause level: 1

<< Backtracking Failed >>
Parse completed.
```

(29) *Who does Pamela love?*

```
Initializing state to: ++++---

LWH     +++++-- 1 who <>
LQUES   +++-+-- 1 does <>
NSUBJ   -++-+++ 1
LNAME   -++-+-- 1 Pamela <>
LVERB   --+-+-- 1 love <>
NOBJ    ----+++ 1
NNGAP   ------- 1
I CLOSE ++++--- 1 ? <>
Successful Parse
WH:who; QUES:does; SUBJ:NAME:Pamela; VERB:love; OBJ:NGAP:CLOSE:?;

<< Backtracking >> using Word state now --+-+-- clause level: 1

<< Backtracking Failed >>
Parse completed.
```

(30) * *Who does George love Pamela?*

```
Initializing state to: ++++---
```

```
LWH    +++++-- 1 who <>
LQUES  +++-+-- 1 does <>
NSUBJ  -++-+++ 1
LNAME  -++-+-- 1 George <>
LVERB  --+-+-- 1 love <>
NOBJ   ----+++ 1
LNAME  ----+-- 1 Pamela <>

<< Backtracking >> using Word state now ----+-- clause level: 1

<< Backtracking Failed >>
Parse completed.
Ungrammatical Input
```

B.4. Relative Clauses

Ordering Features: S V O AUX GAP DET HEAD NTERM REL
Properties: genpl gensng inf past pastpart pl pres pres3 prespart sng
Productions:

```
terminators
NPEND    N  cond -DET..HEAD +NTERM                  change -NTERM
RELEND   N  cond -S..V -AUX -DET..NTERM +REL        change -NTERM..REL
            returnup
NP post-modifiers
RELC  L  cond +V -DET..HEAD +NTERM   change +S..GAP -DET..HEAD +REL
         shiftdown
RELR  L  cond -V -DET..HEAD +NTERM   change +S..GAP -DET..HEAD
phrasal
ART    L  cond +DET..NTERM    change -DET
NAME   L  cond +DET..NTERM    change -DET..NTERM
NGAP   N  cond +GAP..NTERM    change -GAP..NTERM
NOUN   L  cond +HEAD..NTERM   change -DET..HEAD
general
CLOSE   I  cond -S..V -GAP..NTERM       change +S..AUX -GAP..REL
CTHAT   L  cond -V +O -DET..NTERM       change +S..AUX +DET..NTERM
OBJ     N  cond -V +O -DET..NTERM       change -O +DET..NTERM
QUES    L  cond +S +AUX -DET..NTERM     change -AUX -REL
SUBJ    N  cond +S -DET..NTERM          change -S +DET..NTERM
VERB    L  cond -S +V -DET..NTERM       change -V -AUX
WH      L  cond +S -GAP..NTERM          change +GAP
```

Non-lexical Ordering: SUBJ OBJ NGAP NPEND RELEND
Paradigms:

```
BED s' <genpl>
    's <gensng>
    s <pl>
    $ <sng>
```

```
dO o <inf pastpart pres pres3 prespart>
   i <past>

Do ing <prespart>
   es <pres3>
   ne <pastpart>
   d <past>
   $ <inf pres>

EAt en <pastpart>
    s <pres3>
    $ <inf pres>

eAT ate <past>
    eat <inf pastpart pres pres3 prespart>

LOVE ing <prespart>
     es <pl pres3>
     ed <past pastpart>
     e <inf pres sng>

MaN a <gensng sng>
    e <genpl pl>

MAn s' <genpl>
    's <gensng>
    $ <pl sng>

PULL ing <prespart>
     ed <past pastpart>
     s <pl pres3>
     $ <inf pres sng>
```

Lexicon:

articles

```
a     cat ART                       morph 'a'
that  cat RELR RELC CTHAT ART       morph 'that'
the   cat ART                       morph 'the'
```

close

```
.  cat CLOSE  morph '.'
?  cat CLOSE  morph '?'
```

nouns

```
man      cat NOUN  morph 'm_MaN_n_MAn_'
pizza    cat NOUN  morph 'pizza_BED_'
quiche   cat NOUN  morph 'quiche_BED_'
```

wh words

```
what  cat WH              morph 'what'
who   cat WH RELR RELC    morph 'who'
```

verbs
```
do     cat VERB QUES   morph 'd_dO__Do_'
eat    cat VERB        morph '_eAT__EAt_'
hate   cat VERB        morph 'hat_LOVE_'
love   cat VERB        morph 'lov_LOVE_'
think  cat VERB        morph 'think_PULL_'
```
names
```
george   cat NAME   morph 'george_BED_'
martha   cat NAME   morph 'martha_BED_'
pamela   cat NAME   morph 'pamela_BED_'
```

(31) *Men who hate men that eat quiche love pizza.*

```
Initializing state to: ++++-----

N SUBJ    -+++-+++- 1
L NOUN    -+++---+- 1 man <pl>
L RELC    +++++--++ 2 who <>
N NPEND   +++++---+ 2
N SUBJ    -++++++++ 2
N NGAP    -+++----+ 2
L VERB    --+-----+ 2 hate <inf pres sng>
N OBJ     -----++++ 2
L NOUN    -------++ 2 man <pl>
L RELR    +++++--++ 2 that <>
N NPEND   +++++---+ 2
N SUBJ    -++++++++ 2
N NGAP    -+++----+ 2
L VERB    --+-----+ 2 eat <inf pres>
N OBJ     -----++++ 2
L NOUN    -------++ 2 quiche <sng>
N NPEND   --------+ 2
N RELEND -+++----- 1
L VERB    --+------ 1 love <inf pres sng>
N OBJ     -----+++- 1
L NOUN    -------+- 1 pizza <sng>
N NPEND   --------- 1
I CLOSE   ++++----- 1 . <>
Successful Parse
```
SUBJ:NOUN:*man*; RELC:*who*; NPEND:SUBJ:NGAP:VERB:*hate*; OBJ:NOUN:*man*; RELR:*that*; NPEND:SUBJ:NGAP:VERB:*eat*; OBJ:NOUN:*quiche*; NPEND:RELEND: VERB:*love*; OBJ:NOUN:*pizza*; NPEND:CLOSE:.;

```
<< Backtracking >> using Word state now -------+- clause level: 1

<< Backtracking Failed >>
Parse completed.
```

(32) *Who do the men think that Pamela loves?*

```
Initializing state to: ++++-----

L WH      +++++---- 1 who <>
```

```
L QUES     +++-+---- 1 do <inf pres>
N SUBJ     -++-++++- 1
L ART      -++-+-++- 1 the <>
L NOUN     -++-+--+- 1 man <pl>
N NPEND    -++-+---- 1
L VERB     --+-+---- 1 think <inf pres sng>
L CTHAT    ++++++++- 1 that <>
L NAME     +++++---- 1 pamela <sng>
N SUBJ     -+++++++- 1
N NGAP     -+++----- 1
L VERB     --+------ 1 love <pl pres3>
I CLOSE    ++++----- 1 ? <>
Successful Parse
WH:who; QUES:do; SUBJ:ART:the; NOUN:man; NPEND:VERB:think; CTHAT:that;
  NAME:pamela; SUBJ:NGAP:VERB:love; CLOSE:?;

<< Backtracking >> using Word state now --+------ clause level: 1
N OBJ      -----+++- 1

<< Backtracking Failed >>
Parse completed.
```

B.5. Subcategorization

Boundaries: Obj Pred
Ordering Features: S TR V O DET HEAD XO THAT INF BE
Properties: genpl gensng inf past pastpart pl pres pres3 prespart sng

Productions
phrasal

```
ADJ    L  cond -V..O +BE      change -S..HEAD -BE
ART    L  cond +DET..HEAD     change -DET
NAME   L  cond +DET..HEAD     change -DET..HEAD
NOUN   L  cond +HEAD          change -DET..HEAD
```

general			
BE	L	cond +V -HEAD	change -V..O +BE
CLOSE	I	cond -S..HEAD	change +S..O -DET..BE
CTHAT	N	cond -V -HEAD +THAT	change +S..O -DET..INF
INF	L	cond -V -HEAD +INF	change -S +TR..O -DET..INF
INF_	S VERB	cond -TR -HEAD	change +INF
INTRANS	S VERB	cond -S +TR -HEAD save Pred	change -TR -O
OBJ	N	cond -V +O -HEAD save Obj	change -O +DET..HEAD
SUBJ	N	cond +S -HEAD	change -S +DET..HEAD
THAT	L	cond -V -HEAD +THAT	change +S..O -DET..INF
THAT_	S VERB	cond -TR -HEAD -INF	change +THAT
TRANS	S VERB	cond -S +TR -HEAD save Pred	change -TR
VERB	L	cond -TR +V -HEAD	change -V
XO	S VERB	cond -TR +O -HEAD	change +XO
XOBJ	N	cond -V +O -HEAD +XO	change -O -XO

Non-lexical Ordering: SUBJ OBJ XOBJ CTHAT

Paradigms

```
BE were <past pl>
   are <pres>
   was <past sng>
   am <pres>
   is <pres3>

BED s' <genpl>
    's <gensng>
    s <pl>
    $ <sng>

dO o <inf pastpart pres pres3 prespart>
   i <past>

Do ing <prespart>
   es <pres3>
   ne <pastpart>
   d <past>
   $ <inf pres>

eAT ate <past>
    eat <inf pastpart pres pres3 prespart>

EAt en <pastpart>
    s <pres3>
    $ <inf pres>
```

```
GiVE i <inf pastpart pres pres3 prespart>
      a <past>

GIVen ing <prespart>
      ves <pres3>
      en <pastpart>
      e <inf past pres>

IforY i <past pastpart pl pres3>
      y <inf pres prespart sng>

lEAVE av <inf pl pres pres3 prespart sng>
      ft <past pastpart>

LEAVe ing <prespart>
      es <pl pres3>
      e <inf pres sng>
      $ <past pastpart>

LOVE ing <prespart>
     es <pl pres3>
     ed <past pastpart>
     e <inf pres sng>

MAn s' <genpl>
    's <gensng>
    $ <pl sng>

MaN a <gensng sng>
    e <genpl pl>

PP p <past pastpart prespart>
   $ <inf pres pres3>

PULL ing <prespart>
     ed <past pastpart>
     s <pl pres3>
     $ <inf pres sng>

TRy ying <prespart>
    ies <pl pres3>
    ied <past pastpart>
    y <inf pres sng>
```

Lexicon

punctuation

```
. cat CLOSE   morph '.'
```

articles

a	cat ART	morph 'a'
the	cat ART	morph 'the'

nouns

robot	cat NOUN	morph 'robot_BED_'

adjectives

cold	cat ADJ	morph 'cold'
good	cat ADJ	morph 'good'

names

george	cat NAME	morph 'george_BED_'
john	cat NAME	morph 'john_BED_'
martha	cat NAME	morph 'martha_BED_'
mary	cat NAME	morph 'mary_BED_'
tom	cat NAME	morph 'tom_BED_'

misc

that	cat THAT	morph 'that'
to	cat INF	morph 'to'

verbs

be	cat BE	morph '_BE_'
believe	cat THAT1 XO TRANS VERB	morph 'believ_LOVE_'
eat	cat TRANS INTRANS VERB	morph '_eAT__EAt_'
give	cat TRANS VERB	morph 'g_GiVE_v_GIVen_'
giveaway	cat TRANS VERB	morph 'g_GiVE_v_GIVen_+away'
hope	cat INTRANS THAT1 INF1 VERB	morph 'hop_LOVE_'
hurry	cat TRANS INTRANS VERB	morph 'hurr_TRy_'
hurryup	cat INTRANS TRANS VERB	morph 'hurr_TRy_ up'
leave	cat INTRANS VERB	morph 'le_lEAVE__LEAVe_'
love	cat TRANS VERB	morph 'lov_LOVE_'

(33) *George hopes to leave.*

```
Initializing state to: ++++------

N SUBJ      -+++++---- 1
L NAME      -+++------ 1 george <sng>
S INTRANS   --+------- 1
S INF1      --+-----+- 1
L VERB      --------+- 1 hope <pl pres3>
L INF       -+++------ 1 to <>
S INTRANS   --+------- 1
L VERB      ---------- 1 leave <inf pres sng>
I CLOSE     ++++------ 1 . <>
Successful Parse
SUBJ:NAME:george; INTRANS:INF1:VERB:hope; INF:to; INTRANS:VERB:leave; CLOSE:.;

<< Backtracking >> using Word state now ---------- clause level: 1

<< Backtracking >> using Pred1 state now -+++------ clause level: 1

<< Backtracking Failed >>
Parse completed.
```

(34) *George hopes that Martha left.*

```
Initializing state to: ++++------

NSUBJ      -+++++---- 1
L NAME     -+++------ 1 george <sng>
S INTRANS  --+------- 1
S THAT1    --+----+-- 1
S INF1     --+----++- 1
L VERB     -------++- 1 hope <pl pres3>
L THAT     ++++------ 1 that <>
NSUBJ      -+++++---- 1
L NAME     -+++------ 1 martha <sng>
S INTRANS  --+------- 1
L VERB     ---------- 1 leave <past pastpart>
I CLOSE    ++++------ 1 . <>
Successful Parse
SUBJ:NAME:george; INTRANS:THAT1:INF1:VERB:hope; THAT:that;
  SUBJ:NAME:martha; INTRANS:VERB:leave; CLOSE:.;

<< Backtracking >> using Word state now ---------- clause level: 1

<< Backtracking >> using Pred1 state now -+++------ clause level: 1

<< Backtracking Failed >>
Parse completed.
```

(35) *George believes Martha.*

```
Initializing state to: ++++------

NSUBJ      -+++++---- 1
L NAME     -+++------ 1 george <sng>
S TRANS    --++------ 1
S THAT1    --++---+-- 1
S XO       --++--++-- 1
L VERB     ---+--++-- 1 believe <pl pres3>
NOBJ       ----++++-- 1
L NAME     ------++-- 1 martha <sng>
I CLOSE    ++++------ 1 . <>
Successful Parse
SUBJ:NAME:george; TRANS:THAT1:XO:VERB:believe; OBJ:NAME:martha; CLOSE:.;

<< Backtracking >> using Word state now ------++-- clause level: 1
NCTHAT     ++++------ 1
NSUBJ      -+++++---- 1

<< Backtracking >> using Obj1 state now ---+--++-- clause level: 1
NXOBJ      -------+-- 1
NCTHAT     ++++------ 1
NSUBJ      -+++++---- 1
```

```
L NAME     -+++------ 1 martha <sng>

<< Backtracking >> using Word state now -+++------ clause level: 1

<< Backtracking >> using Pred1 state now -+++------ clause level: 1

<< Backtracking Failed >>
Parse completed.
```

(36) *George believes that Martha left.*

```
Initializing state to: ++++------

N SUBJ     -+++++---- 1
L NAME     -+++------ 1 george <sng>
S TRANS    --++------ 1
S THAT1    --++---+-- 1
S XO       --++--++-- 1
L VERB     ---+--++-- 1 believe <pl pres3>
L THAT     ++++------ 1 that <>
N SUBJ     -+++++---- 1
L NAME     -+++------ 1 martha <sng>
S INTRANS  --+------- 1
L VERB     ---------- 1 leave <past pastpart>
I CLOSE    ++++------ 1 . <>
Successful Parse
SUBJ:NAME:george; TRANS:THAT1:XO:VERB:believe; THAT:that;
  SUBJ:NAME:martha; INTRANS:VERB:leave; CLOSE:.;

<< Backtracking >> using Word state now ---------- clause level: 1

<< Backtracking >> using Pred1 state now -+++------ clause level: 1

<< Backtracking Failed >>
Parse completed.
```

(37) *George believes.*

```
Initializing state to: ++++------

N SUBJ     -+++++---- 1
L NAME     -+++------ 1 george <sng>
S TRANS    --++------ 1
S THAT1    --++---+-- 1
S XO       --++--++-- 1
L VERB     ---+--++-- 1 believe <pl pres3>
N OBJ      ----++++-- 1

<< Backtracking >> using Obj1 state now ---+--++-- clause level: 1
N XOBJ     -------+-- 1
I CLOSE    ++++------ 1 . <>
Successful Parse
SUBJ:NAME:george; TRANS:THAT1:XO:VERB:believe; XOBJ:CLOSE:.;
```

```
<< Backtracking >> using Word state now ---+--++-- clause level: 1
N CTHAT    ++++------ 1
N SUBJ     -+++++---- 1

<< Backtracking >> using Pred1 state now -+++------ clause level: 1

<< Backtracking Failed >>
Parse completed.
```

(38) *George hopes.*

```
Initializing state to: ++++------

N SUBJ     -+++++---- 1
L NAME     -+++------ 1 george <sng>
S INTRANS  --+------- 1
S THAT1    --+----+-- 1
S INF1     --+----++- 1
L VERB     -------++- 1 hope <pl pres3>
I CLOSE    ++++------ 1 . <>
Successful Parse
SUBJ:NAME:george; INTRANS:THAT1:INF1:VERB:hope; CLOSE:.;

<< Backtracking >> using Word state now -------++- clause level: 1
N CTHAT    ++++------ 1
N SUBJ     -+++++---- 1

<< Backtracking >> using Pred1 state now -+++------ clause level: 1

<< Backtracking Failed >>
Parse completed.
```

(39) **George believes to leave.*

```
Initializing state to: ++++------

N SUBJ     -+++++---- 1
L NAME     -+++------ 1 george <sng>
S TRANS    --++------ 1
S THAT1    --++---+-- 1
S XO       --++--++-- 1
L VERB     ---+--++-- 1 believe <pl pres3>
N OBJ      ----++++-- 1

<< Backtracking >> using Obj1 state now ---+--++-- clause level: 1
N XOBJ     -------+-- 1
N CTHAT    ++++------ 1
N SUBJ     -+++++---- 1

<< Backtracking >> using Word state now ---+--++-- clause level: 1

<< Backtracking >> using Pred1 state now -+++------ clause level: 1
```

```
     << Backtracking Failed >>
     Parse completed.
     Ungrammatical Input
```

(40) **George hopes believes.*

```
     Initializing state to: ++++------

     N SUBJ      -+++++---- 1
     L NAME      -+++------ 1 george <sng>
     S INTRANS   --+------- 1
     S THAT1     --+----+-- 1
     S INF1      --+----++- 1
     L VERB      -------++- 1 hope <pl pres3>
     N CTHAT     ++++------ 1
     N SUBJ      -+++++---- 1

     << Backtracking >> using Word state now -------++- clause level: 1

     << Backtracking >> using Pred1 state now -+++------ clause level: 1

     << Backtracking Failed >>
     Parse completed.
     Ungrammatical Input
```

(41) **George hopes Martha.*

```
     Initializing state to: ++++------

     N SUBJ      -+++++---- 1
     L NAME      -+++------ 1 george <sng>
     S INTRANS   --+------- 1
     S THAT1     --+----+-- 1
     S INF1      --+----++- 1
     L VERB      -------++- 1 hope <pl pres3>
     N CTHAT     ++++------ 1
     N SUBJ      -+++++---- 1
     L NAME      -+++------ 1 martha <sng>

     << Backtracking >> using Word state now -+++------ clause level: 1

     << Backtracking >> using Pred1 state now -+++------ clause level: 1

     << Backtracking Failed >>
     Parse completed.
     Ungrammatical Input
```

B.6. MorphoSyntactic Properties

Ordering Features: S V O GAP DET ADJ HEAD NTERM REL COMP AF BE TENS PASSIVE

Default cond: -DET..NTERM

Properties: <first second third> <pl sg> <past pres> inf nom pastpart prespart

Productions

verb categories

CADJ	L	cond -S +V -BE	change -V..O
VCOMP	L	cond -S +V +AF	change -V..O +COMP -AF
VINTRANS	L	cond -S +V +AF	change -V..O -AF..BE
VTRANS	L	cond -S +V +AF	change -V -AF..BE

terminators

NPEND	N	cond +NTERM	change -NTERM
RELEND	N	cond -S..V -GAP +REL -BE returnup	change -NTERM

relative pronouns

RELC	L	cond +V +NTERM shiftdown	change +S..GAP -DET..NTERM +REL
RELR	L	cond -V +NTERM	change +S..GAP -DET..NTERM

phrasal

ADJ	L	cond ?DET +ADJ..NTERM	change -DET
DET	L	cond +DET..NTERM	change -DET
NAME	L	cond +DET..NTERM	change -DET..NTERM -AF
NGAP	N	cond +GAP..NTERM	change -GAP..NTERM
NOUN	L	cond ?DET..ADJ +HEAD..NTERM lexprop <nom>	change -DET..HEAD -AF

misc

BE	L	cond +AF..BE	change -AF..BE
CLOSE	I	cond -S..GAP -COMP	change +S..O -GAP..COMP +AF..TENS -PASSIVE
CTHAT	L	cond +COMP	change +S..GAP -DET..COMP +AF..BE
OBJ	N	cond -V +O	change -O +DET..NTERM
PASSIVE	N	cond -BE -PASSIVE lexprop <pastpart>	change -O +AF +PASSIVE
PASSIVEBY	L	cond -V +PASSIVE	change +DET..NTERM
PROG	N	cond -AF..BE -PASSIVE lexprop <prespart>	change +AF..BE
QUES	N	cond +S +TENS lexprop <past pres>	change +AF -TENS
SUBJ	N	cond +S	change -S +DET..NTERM
TENSE	N	cond -S +TENS lexprop <past pres>	change +AF -TENS
WH	L	cond +S -GAP	change +GAP

Non-lexical Ordering: SUBJ TENSE QUES PROG PASSIVE OBJ NGAP NPEND RELEND

Paradigms:

AM $ <sg:pres:first>

```
ARE $ <pres:pl pres:second>

BE ing <prespart>
   en <pastpart>
   $ <inf>

BED s <nom third:pl>
    $ <nom sg:third>

BOX es <nom third:pl>
    $ <nom sg:third>

Do oing <prespart>
   oes <sg:third:pres>
   id <past pastpart>
   o <inf pres:first pres:pl pres:second>

FiND ou <past pastpart>
     i <inf pres prespart>

FINDing ing <prespart>
        s <sg:third:pres>
        $ <first inf past pastpart pl second>

HAve ving <prespart>
     ve <inf pres:first pres:pl pres:second>
     s <sg:third:pres>
     d <past pastpart>

IS $ <sg:third:pres>

LOVE ing <prespart>
     ed <past pastpart>
     es <sg:third:pres>
     e <inf pres:first pres:pl pres:second>

PULL ing <prespart>
     ed <past pastpart>
     s <sg:third:pres>
     $ <inf pres:first pres:pl pres:second>

SHEEP $ <nom third>

SiNG i <inf pres prespart>
     a <past>
     u <pastpart>
```

```
WAS ere <past:pl second:past>
     as <sg:past:first sg:third:past>
```

Lexicon:

```
misc
by    cat PASSIVEBY   morph 'by'
wh words
what   cat WH RELC RELR   morph 'what'
who    cat WH RELC RELR   morph 'who'
aux
be   cat BE   morph 'are_ARE_'
              m 'is_IS_'
              m 'w_WAS_'
              m 'am_AM_'
              m 'be_BE_'
verbs
find      cat NOUN VTRANS      morph 'f_FiND_nd_FINDing_'
                               m 'find_BED_'
kick      cat NOUN VTRANS      morph 'kick_PULL_'
                               m 'kick_BED_'
sing      cat NOUN VINTRANS    morph 's_SiNG_ng_FINDing_'
squeak    cat NOUN VINTRANS    morph 'squeak_PULL_'
                               m 'squeak_BED_'
nouns
barn      cat NOUN   morph 'barn_BED_'
bird      cat NOUN   morph 'bird_BED_'
block     cat NOUN   morph 'block_BED_'
bucket    cat NOUN   morph 'bucket_BED_'
fox       cat NOUN   morph 'fox_BOX_'
horse     cat NOUN   morph 'horse_BED_'
robot     cat NOUN   morph 'robot_BED_'
adjectives
large   cat ADJ CADJ   morph 'large'
red     cat ADJ CADJ   morph 'red'
silly   cat ADJ CADJ   morph 'silly'
articles
a     cat DET   morph 'a'
the   cat DET   morph 'the'
names
George   cat NAME   morph 'george_BED_'
Martha   cat NAME   morph 'martha_BED_'
punctuation
.   cat CLOSE   morph '.'
?   cat CLOSE   morph '?'
```

(42) *The robot is being kicked.*

```
Initializing state to: +++---------+++-

N SUBJ        -++-++++--+++- 1
L DET         -++--+++--+++- 1 the <>
```

```
L NOUN       -++-----+---++- 1 robot <nom sg:third>
N NPEND      -++---------++- 1
N TENSE      -++--------++-- 1
L BE         -++------------ 1 be <sg:third:pres>
N PROG       -++--------++-- 1
L BE         -++------------ 1 be <past pres prespart>
N PASSIVE    -+---------+--+ 1
L VTRANS     --------------+ 1 kick <past pastpart>
I  CLOSE     +++--------+++- 1 . <>
Successful Parse
SUBJ:DET:the; NOUN:robot; NPEND:TENSE:BE:be; PROG:BE:be; PASSIVE:
  VTRANS:kick; CLOSE:.;

<< Backtracking >> using Word state now --------------+ clause level: 1

<< Backtracking Failed >>
Parse completed.
```

(43) *George was kicked by Martha.*

```
Initializing state to: +++--------+++-

N SUBJ       -++-++++---+++- 1
L NAME       -++---------++- 1 George <nom sg:third>
N TENSE      -++--------++-- 1
L BE         -++------------ 1 be <sg:past:first sg:third:past>
N PASSIVE    -+---------+--+ 1
L VTRANS     --------------+ 1 kick <past pastpart>
L PASSIVEBY  ----++++------+ 1 by <>
L NAME       --------------+ 1 Martha <nom sg:third>
I  CLOSE     +++--------+++- 1 . <>
Successful Parse
SUBJ:NAME:George; TENSE:BE:be; PASSIVE:VTRANS:kick; PASSIVEBY:by;
  NAME:Martha; CLOSE:.;

<< Backtracking >> using Word state now --------------+ clause level: 1

<< Backtracking Failed >>
Parse completed.
```

(44) *George is being silly.*

```
Initializing state to: +++--------+++-

N SUBJ       -++-++++---+++- 1
L NAME       -++---------++- 1 George <nom sg:third>
N TENSE      -++--------++-- 1
L BE         -++------------ 1 be <sg:third:pres>
N PROG       -++--------++-- 1
L BE         -++------------ 1 be <past pres prespart>
L CADJ       --------------- 1 silly <>
I  CLOSE     +++--------+++- 1 . <>
Successful Parse
```

```
SUBJ:NAME:George; TENSE:BE:be; PROG:BE:be; CADJ:silly; CLOSE:.;

<< Backtracking >> using Word state now -------------- clause level: 1

<< Backtracking Failed >>
Parse completed.
```

(45) *The red robot squeaked.*

```
Initializing state to: +++--------+++-

N SUBJ       -++-++++--+++- 1
L DET        -++--+++--+++- 1 the <>
L ADJ        -++--+++--+++- 1 red <>
L NOUN       -++----+---++- 1 robot <nom sg:third>
N NPEND      -++---------++- 1
N TENSE      -++--------++-- 1
L VINTRANS   -------------- 1 squeak <past pastpart>
I  CLOSE     +++--------+++- 1 . <>
Successful Parse
SUBJ:DET:the; ADJ:red; NOUN:robot; NPEND:TENSE:VINTRANS:squeak; CLOSE:.;

<< Backtracking >> using Word state now -------------- clause level: 1

<< Backtracking Failed >>
Parse completed.
```

B.7. Boundary Registers

Boundaries: NP NPmod Obj Pred Prep Subj Tense Topic
Ordering Features: S TENS BE AF V O PASS THAT VSUB NP DET HEAD NEND REL ROLE PREP GAP RELEND
Default cond: -DET..HEAD
Properties: <first second third> <pl sg> <past pres> inf nom pastpart prespart
Productions:

clausal

BE	L	cond +BE -AF	change -BE +AF +NP
CLOSE	I	cond -S..O -ROLE -GAP..RELEND	change +S..O -PASS..VSUB +NP -DET..RELEND
IMP	N	cond +TENS -REL -GAP	change -S..AF
		lexprop <past pres>	save Tense
PASS	N	cond -S -BE +AF..V	change -AF -O +PASS -NP
		lexprop <pastpart>	
PROG	N	cond -S -BE +AF..V -PASS	change -AF +NP
		lexprop <prespart>	
QUES	N	cond +TENS -REL..ROLE	change -TENS -AF -NP
		lexprop <past pres>	save Tense
SUBJ	N	cond +S -TENS +ROLE	change -S -DET..ROLE
		save Subj	
TENS	N	cond +TENS -NEND +ROLE	change -TENS..AF
		lexprop <past pres>	save Tense

verb subcategories

INTRAN	S V	cond -S -PASS	change -O +VSUB
THAT_	S V	cond -S	change +THAT..VSUB
TRANS	S V	cond -S	change +VSUB..NP

verbs and predicates

CADJ	L	cond -S -BE +AF..V	change -AF..O -NP
PP	S PREP	cond -S..BE +AF..V	change -AF..O -NP +PREP
		save Prep	
V	L	cond -S -AF +V +VSUB	change -BE -V -VSUB
		save Pred	

verb compliments

CTHAT	L	cond -V +THAT	change +S..O -PASS..VSUB +NP -DET..PREP -RELEND
OBJ	N	cond -V +O +ROLE	change -O -DET..ROLE
		save Obj	
PASSBY	L	cond -V..O +PASS	change +NP..HEAD +PREP
PPEND	N	cond +ROLE..PREP	change -DET..PREP
		save NPmod	
PREP	L	cond +PREP	change +DET..HEAD
VREL1	S PREP	cond -V..O -PREP	change -THAT +PREP
		save Prep	

noun phrase productions

```
DET      L  cond +DET..HEAD               change -DET
NAME     L  cond +DET..HEAD               change -DET..HEAD +NEND..ROLE
NOUN     L  cond ?DET +HEAD               change -DET..HEAD +NEND..ROLE
            lexprop <nom>
NP       N  cond -VSUB +NP -PREP          change -NP +DET..HEAD
            save NP
NPEND    N  cond +NEND -PREP              change -DET..NEND +ROLE
            save NPmod
REDREL   N  cond +S..O +REL +GAP          change -S..BE -NP -REL +ROLE -GAP
RELC0    N  cond +O +NEND -RELEND         change +S..O -PASS..VSUB +NP
                                                -DET..NEND +REL -ROLE..PREP
                                                +GAP..RELEND
            shiftdown
RELEND   N  cond -S..O -ROLE -GAP +RELEND change -DET..REL
            returnup
RELR0    N  cond -S..O +NEND -GAP         change +S..O -PASS..VSUB +NP
                                                -DET..NEND +REL -ROLE..PREP
                                                +GAP
```

questions

```
NGAP  N  cond +NP ?DET..HEAD -REL +GAP  change -NP..REL +ROLE -GAP
WH    L  cond +S -GAP                   change -NP +GAP
         save Topic
```

Non-lexical Ordering: SUBJ QUES TENS IMP PROG PASS PPEND OBJ NP NPEND RELR0 RELC0 REDREL RELEND NGAP

Paradigms:

```
AM $ <sg:pres:first>

ARE $ <pres:pl pres:second>

BE ing <prespart>
   en <pastpart>
   $ <inf>

BED s <nom third:pl>
    $ <nom third:sg>

BODY ies <nom third:pl>
     y <nom third:sg>

BOX es <nom third:pl>
    $ <nom third:sg>

BUILT ding <prespart>
      ds <pres:third:sg>
      d <inf pres:first pres:pl pres:second>
      t <past pastpart>
```

```
Do oing <prespart>
   oes <pres:third:sg>
   id <past pastpart>
   o <inf pres:first pres:pl pres:second>

FaLL a <inf pres prespart>
     e <past pastpart>

FALLen ing <prespart>
       en <pastpart>
       s <pres:third:sg>
       $ <first inf past pl second>

FiND ou <past pastpart>
     i <inf pres prespart>

FINDing ing <prespart>
        s <pres:third:sg>
        $ <first inf past pastpart pl second>

FLy ying <prespart>
    ies <pres:third:sg>
    own <pastpart>
    ew <past>
    y <inf pres:first pres:pl pres:second>

GiVen iving <prespart>
      ives <pres:third:sg>
      iven <pastpart>
      ive <inf pres:first pres:pl pres:second>
      ave <past>

IS $ <pres:third:sg>

LOVE ing <prespart>
     ed <past pastpart>
     es <pres:third:sg>
     e <inf pres:first pres:pl pres:second>

PULL ing <prespart>
     ed <past pastpart>
     s <pres:third:sg>
     $ <inf pres:first pres:pl pres:second>

SHEEP $ <nom third>
```

```
SiT i <inf pres prespart>
    a <past>

TT t <prespart>
    $ <inf pastpart prespart>

WAS ere <past:pl second:past>
      s <sg:past:first third:past:sg>
```

Lexicon:

prepositions

by	cat VREL1 PP PREP PASSBY	morph 'by'
from	cat VREL1 PP PREP	morph 'from'
in	cat VREL1 PP PREP	morph 'in'
on	cat VREL1 PP PREP	morph 'on'
past	cat VREL1 PP PREP	morph 'past'

common nouns

arrow	cat NOUN	morph 'arrow_BED_'
barn	cat NOUN	morph 'barn_BED_'
block	cat NOUN	morph 'block_BED_'
box	cat NOUN	morph 'box_BOX_'
flower	cat NOUN	morph 'flower_BED_'
horse	cat NOUN	morph 'horse_BED_'
sheep	cat NOUN	morph 'sheep_SHEEP_'
table	cat NOUN	morph 'table_BED_'

noun/verbs

fly	cat NOUN INTRAN TRANS V	morph 'fl_FLy_'
		m 'fl_BODY_'
like	cat VREL1 PP PREP NOUN TRANS V	morph 'like_BED_'
		m 'lik_LOVE_'
love	cat NOUN TRANS V	morph 'love_BED_'
		m 'lov_LOVE_'
race	cat TRANS INTRAN V NOUN	morph 'race_BED_'
		m 'rac_LOVE_'
time	cat TRANS INTRAN V NOUN	morph 'tim_LOVE_'
		m 'time_BED_'

verbs

be	cat BE	morph 'w_WAS_'
		m 'be_BE_'
		m 'is_IS_'
		m 'am_AM_'
		m 'are_ARE_'
borrow	cat TRANS V	morph 'borrow_PULL_'
fall	cat INTRAN V	morph 'f_FaLL_ll_FALLen_'
find	cat TRANS V	morph 'f_FiND_nd_FINDing_'
open	cat TRANS INTRAN V	morph 'open_PULL_'
receive	cat TRANS V	morph 'receiv_LOVE_'
sit	cat INTRAN V	morph 's_SiT_t_TT__FINDing_'
think	cat THAT_ TRANS V	morph 'think_PULL_'

wh words

what	cat WH	morph 'what'
who	cat WH	morph 'who'

punctuation

.	cat CLOSE	morph '.'
?	cat CLOSE	morph '?'

adjectives

fat	cat CADJ	morph 'fat'
heavy	cat CADJ	morph 'heavy'
red	cat CADJ	morph 'red'

names

George	cat NAME	morph 'george_BED_'
Martha	cat NAME	morph 'martha_BED_'
Mary	cat NAME	morph 'mary_BED_'

determiners

a	cat DET	morph 'a'
an	cat DET	morph 'an'
that	cat DET CTHAT	morph 'that'
the	cat DET	morph 'the'

(46) *The horse raced past the barn fell.

```
Initializing state to: ++++++---+---------

N NP      ++++++-----++------ 1
L DET     ++++++------+------ 1 the <>
L NOUN    ++++++-------+++--- 1 horse <nom sg:third>
N NPEND   ++++++--------++--- 1
N TENS    +---++--------++--- 1
N SUBJ    ----++------------- 1
S TRANS   ----++---++-------- 1
S INTRAN  ----+----++-------- 1
L V       ----------+-------- 1 race <past pastpart>
S VREL1   ----------+-----+-- 1
L PREP    ----------+++---+-- 1 past <>
L DET     ----------+-+---+-- 1 the <>
L NOUN    ----------+--++++-- 1 barn <nom sg:third>
S INTRAN  ---------++--++++-- 1

<< Backtracking >> using Word state now ----------+--++++-- clause level: 1
N PPEND   ----------+-------- 1
N NP      -----------++------ 1

<< Backtracking >> using NP1 state now ----------+-------- clause level: 1

<< Backtracking >> using NPmod1 state now ----------+--++++-- clause level: 1
N RELR0   ++++++----+---+--+- 1
N NP      ++++++----++--+--+- 1

<< Backtracking >> using NP1 state now ++++++----+---+--+- clause level: 1
N REDREL  ---+++---------+--- 1
```

```
<< Backtracking >> using Prep1 state now ---------+-------- clause level: 1
N NP         ----------++------ 1

<< Backtracking >> using NP1 state now ---------+-------- clause level: 1

<< Backtracking >> using Pred1 state now ----+---++-------- clause level: 1

<< Backtracking >> using Subj1 state now +---++-------++--- clause level: 1

<< Backtracking >> using Tense1 state now ++++++-------++--- clause level: 1

<< Backtracking Failed >>
Parse completed.
Ungrammatical Input
```

(47) Is the block on the table red?

```
Initializing state to: ++++++---+--------

N QUES       +-+-++------------ 1
L BE         +--+++---+-------- 1 be <sg:third:pres>
N NP         +--+++----++------ 1
L DET        +--+++-----+------ 1 the <>
L NOUN       +--+++------+++--- 1 block <nom sg:third>
N SUBJ       ---+++------------ 1
S PP         ---------------+-- 1
L PREP       ----------++---+-- 1 on <>
L DET        -----------+---+-- 1 the <>
L NOUN       ------------++++-- 1 table <nom sg:third>
N PPEND      ------------------ 1

<< Backtracking >> using NPmod1 state now ------------++++-- clause level: 1
N RELR0      ++++++---+---+--+- 1
N NP         ++++++----++-+--+- 1

<< Backtracking >> using NP1 state now ++++++---+---+--+- clause level: 1
N REDREL ---+++--------+--- 1
L CADJ       --------------+--- 1 red <>

<< Backtracking >> using Word state now --------------+--- clause level: 1

<< Backtracking >> using Prep1 state now ---+++------------ clause level: 1

<< Backtracking >> using Subj1 state now +--+++------+++--- clause level: 1
N NPEND  +--+++-------++--- 1

<< Backtracking >> using NPmod1 state now +--+++------+++--- clause level: 1
N RELC0      ++++++---+---+--++ 2
N NP         ++++++----++-+--++ 2
```

```
<< Backtracking >> using NP2 state now ++++++---+---+--++ clause level: 2
N REDREL ---+++--------+--+ 2
S PP     --------------++-+ 2
L PREP   ----------++--++-+ 2 on <>
L DET    -----------+--++-+ 2 the <>
L NOUN   ------------++++-+ 2 table <nom sg:third>
N PPEND  -----------------+ 2
N RELEND +--+++--------+--- 1
N SUBJ   ---+++------------ 1
L CADJ   ------------------ 1 red <>
I CLOSE  ++++++---+-------- 1 ? <>
Successful Parse
QUES:BE:be; NP:DET:the; NOUN:block; RELC0:REDREL:PP:PREP:on; DET:the;
  NOUN:table; PPEND:RELEND:SUBJ:CADJ:red; CLOSE:?;

<< Backtracking >> using Word state now ------------------ clause level: 1

<< Backtracking >> using Subj1 state now +--+++--------+--- clause level: 1

<< Backtracking >> using NPmod2 state now ------------++++-+ clause level: 2
N RELR0  ++++++---+---+--++ 2
N NP     ++++++----++-+--++ 2

<< Backtracking >> using NP2 state now ++++++---+---+--++ clause level: 2
N REDREL ---+++--------+--+ 2
L CADJ   --------------+--+ 2 red <>

<< Backtracking >> using Word state now --------------+--+ clause level: 2

<< Backtracking >> using Prep2 state now ---+++--------+--+ clause level: 2

<< Backtracking >> using Tense1 state now ++++++---+-------- clause level: 1
N IMP    ----++---+-------- 1
N NP     ----++----++------ 1

<< Backtracking >> using NP1 state now ----++---+-------- clause level: 1

<< Backtracking >> using Tense1 state now ++++++---+-------- clause level: 1
N NP     ++++++----++------ 1

<< Backtracking >> using NP1 state now ++++++---+-------- clause level: 1

<< Backtracking Failed >>
Parse completed.
```

(48) Is the block on the table?

```
Initializing state to: ++++++---+--------

N QUES      +-+-++------------ 1
L BE        +--+++---+-------- 1 be <sg:third:pres>
N NP        +--+++----++------ 1
L DET       +--+++-----+------ 1 the <>
L NOUN      +--+++------+++--- 1 block <nom sg:third>
N SUBJ      ---+++------------ 1
S PP        ---------------+-- 1
L PREP      ----------++---+-- 1 on <>
L DET       -----------+---+-- 1 the <>
L NOUN      ------------++++-- 1 table <nom sg:third>
N PPEND     ------------------ 1
I CLOSE     ++++++---+-------- 1 ? <>
Successful Parse
QUES:BE:be; NP:DET:the; NOUN:block; SUBJ:PP:PREP:on; DET:the; NOUN:table;
  PPEND:CLOSE:?;

<< Backtracking >> using NPmod1 state now ------------++++-- clause level: 1
N RELR0     ++++++---+---+--+- 1
N NP        ++++++----++-+--+- 1

<< Backtracking >> using NP1 state now ++++++---+---+--+- clause level: 1
N REDREL ---+++--------+--- 1

<< Backtracking >> using Word state now ------------++++-- clause level: 1

<< Backtracking >> using Prep1 state now ---+++------------ clause level: 1

<< Backtracking >> using Subj1 state now +--+++------+++--- clause level: 1
N NPEND     +--+++----++------ 1

<< Backtracking >> using NPmod1 state now +--+++------+++--- clause level: 1
N RELC0     ++++++---+---+--++ 2
N NP        ++++++----++-+--++ 2

<< Backtracking >> using NP2 state now ++++++---+---+--++ clause level: 2
N REDREL ---+++--------+--+ 2
S PP        --------------++-+ 2
L PREP      ----------++--++-+ 2 on <>
L DET       -----------+--++-+ 2 the <>
L NOUN      ------------++++-+ 2 table <nom sg:third>
N PPEND     -----------------+ 2
N RELEND +--+++--------+--- 1
N SUBJ      ---+++------------ 1

<< Backtracking >> using Subj1 state now +--+++--------+--- clause level: 1

<< Backtracking >> using NPmod2 state now ------------++++-+ clause level: 2
N RELR0     ++++++---+---+--++ 2
```

```
N NP       ++++++----++-+--++ 2

<< Backtracking >> using NP2 state now ++++++---+---+--++ clause level: 2
N REDREL ---+++--------+--+ 2

<< Backtracking >> using Word state now ------------++++-+ clause level: 2

<< Backtracking >> using Prep2 state now ---+++--------+--+ clause level: 2

<< Backtracking >> using Tense1 state now ++++++---+-------- clause level: 1
N IMP      ----++---+-------- 1
N NP       ----++----++------ 1

<< Backtracking >> using NP1 state now ----++---+-------- clause level: 1

<< Backtracking >> using Tense1 state now ++++++---+-------- clause level: 1
N NP       ++++++----++------ 1

<< Backtracking >> using NP1 state now ++++++---+-------- clause level: 1

<< Backtracking Failed >>
Parse completed.
```

(49) Time flies like an arrow.

```
Initializing state to: ++++++---+--------

N QUES     +-+-++------------ 1

<< Backtracking >> using Tense1 state now ++++++---+-------- clause level: 1
N IMP      ----++---+-------- 1
S TRANS    ----++--++-------- 1
S INTRAN   ----+---++-------- 1
L V        ---------+-------- 1 time <nom sg:third>
S INTRAN   --------++-------- 1
S TRANS    --------++-------- 1

<< Backtracking >> using Word state now ---------+-------- clause level: 1
N NP       ----------++------ 1
L NOUN     -------------+++-- 1 fly <nom third:pl>
S VREL1    ------------++++-- 1
L PREP     ----------++++++-- 1 like <nom sg:third>
L DET      -----------+++++-- 1 an <>
L NOUN     ------------++++-- 1 arrow <nom sg:third>
N PPEND    ------------------ 1
I CLOSE    ++++++---+-------- 1 . <>
Successful Parse
IMP:TRANS:INTRAN:V:time; NP:NOUN:fly; VREL1:PREP:like; DET:an; NOUN:arrow;
  PPEND:CLOSE:.;
```

```
<< Backtracking >> using NPmod1 state now ------------++++-- clause level: 1
N RELR0    ++++++---+---+--+- 1
N NP       ++++++----++-+--+- 1

<< Backtracking >> using NP1 state now ++++++---+---+--+- clause level: 1
N REDREL ---+++--------+--- 1

<< Backtracking >> using Word state now ------------++++-- clause level: 1

<< Backtracking >> using Prep1 state now ------------+++--- clause level: 1
S TRANS    --------++--+++--- 1

<< Backtracking >> using Pred1 state now ----+---++-------- clause level: 1

<< Backtracking >> using Tense1 state now ++++++---+-------- clause level: 1
N NP       ++++++----++------ 1
L NOUN     ++++++------+++--- 1 time <nom sg:third>
N NPEND    ++++++-------++--- 1
N TENS     +---++-------++--- 1
N SUBJ     ----++------------ 1
S INTRAN   ----+---+--------- 1
S TRANS    ----+---++-------- 1
L V        ---------+-------- 1 fly <nom third:pl>
S VREL1    ---------+-----+-- 1
L PREP     ---------+++---+-- 1 like <nom sg:third>
L DET      ---------+-+---+-- 1 an <>
L NOUN     ---------+--++++-- 1 arrow <nom sg:third>
N PPEND    ---------+-------- 1
I CLOSE    ++++++---+-------- 1 . <>
Successful Parse
NP:NOUN:time; NPEND:TENS:SUBJ:INTRAN:TRANS:V:fly; VREL1:PREP:like; DET:an;
  NOUN:arrow; PPEND:CLOSE:.;

<< Backtracking >> using NPmod1 state now ---------+--++++-- clause level: 1
N RELR0    ++++++---+---+--+- 1
N NP       ++++++----++-+--+- 1

<< Backtracking >> using NP1 state now ++++++---+---+--+- clause level: 1
N REDREL ---+++--------+--- 1

<< Backtracking >> using Word state now ---------+--++++-- clause level: 1

<< Backtracking >> using Prep1 state now ---------+-------- clause level: 1
S TRANS    --------++-------- 1

<< Backtracking >> using Pred1 state now ----+---++-------- clause level: 1

<< Backtracking >> using Subj1 state now +---++-------++--- clause level: 1

<< Backtracking >> using Tense1 state now ++++++-------++--- clause level: 1
<< Backtracking Failed >>
Parse completed.
```

(50) Borrow flies like an arrow.

```
Initializing state to: ++++++---+--------

N QUES     +-+-++------------ 1

<< Backtracking >> using Tense1 state now ++++++---+-------- clause level: 1
N IMP      ----++---+-------- 1
S TRANS    ----++--++-------- 1
L V        -----+---+-------- 1 borrow <inf pres:first pres:pl pres:second>
S INTRAN   --------++-------- 1
S TRANS    --------++-------- 1

<< Backtracking >> using Word state now -----+---+-------- clause level: 1
N NP       -----+----++------ 1
L NOUN     -----+------+++--- 1 fly <nom third:pl>
S TRANS    -----+--++--+++--- 1

<< Backtracking >> using Word state now -----+------+++--- clause level: 1
N OBJ      ------------------ 1
S VREL1    ---------------+-- 1
L PREP     ----------++---+-- 1 like <nom sg:third>
L DET      -----------+---+-- 1 an <>
L NOUN     ------------++++-- 1 arrow <nom sg:third>
N PPEND    ------------------ 1
I CLOSE    ++++++---+-------- 1 . <>
Successful Parse
IMP:TRANS:V:borrow; NP:NOUN:fly; OBJ:VREL1:PREP:like; DET:an; NOUN:arrow;
  PPEND:CLOSE:.;

<< Backtracking >> using NPmod1 state now ------------++++-- clause level: 1
N RELR0    ++++++---+---+--+- 1
N NP       ++++++----++-+--+- 1

<< Backtracking >> using NP1 state now ++++++---+---+--+- clause level: 1
N REDREL ---+++--------+--- 1

<< Backtracking >> using Word state now ------------++++-- clause level: 1

<< Backtracking >> using Prep1 state now ------------------ clause level: 1

<< Backtracking >> using Obj1 state now -----+------+++--- clause level: 1
N NPEND    -----+-------++--- 1

<< Backtracking >> using NPmod1 state now -----+------+++--- clause level: 1
N RELC0    ++++++---+---+--++ 2
```

```
N NP      ++++++----++-+--++ 2
L NOUN    ++++++------+++-++ 2 like <nom sg:third>
N NPEND   ++++++-------++-++ 2
N REDREL ---+++--------+--+ 2

<< Backtracking >> using NPmod2 state now ++++++------+++-++ clause level: 2
N REDREL ---+++------+-+--+ 2

<< Backtracking >> using Word state now ++++++------+++-++ clause level: 2

<< Backtracking >> using NP2 state now ++++++---+---+--++ clause level: 2
N REDREL ---+++--------+--+ 2
S PP      ---------------++-+ 2
L PREP    ----------++--++-+ 2 like <nom sg:third>
L DET     -----------+--++-+ 2 an <>
L NOUN    ------------++++-+ 2 arrow <nom sg:third>
N PPEND   -----------------+ 2
N RELEND -----+--------+--- 1
N OBJ     ------------------ 1
I CLOSE   ++++++---+-------- 1 . <>
Successful Parse
IMP:TRANS:V:borrow; NP:NOUN:fly; RELC0:REDREL:PP:PREP:like; DET:an;
  NOUN:arrow; PPEND:RELEND:OBJ:CLOSE:.;

<< Backtracking >> using Obj1 state now -----+--------+--- clause level: 1

<< Backtracking >> using NPmod2 state now ------------++++-+ clause level: 2
N RELR0   ++++++---+---+--++ 2
N NP      ++++++----++-+--++ 2

<< Backtracking >> using NP2 state now ++++++---+---+--++ clause level: 2
N REDREL ---+++--------+--+ 2

<< Backtracking >> using Word state now ------------++++-+ clause level: 2

<< Backtracking >> using Prep2 state now ---+++--------+--+ clause level: 2

<< Backtracking >> using Pred1 state now ----++--++-------- clause level: 1

<< Backtracking >> using Tense1 state now ++++++---+-------- clause level: 1
N NP      ++++++----++------ 1

<< Backtracking >> using NP1 state now ++++++---+-------- clause level: 1

<< Backtracking Failed >>
Parse completed.
```

Three other of Blank's parsers were implemented using my RV development environment. However, These are too large to include. They are named `agree`, `jul90`, and `apr92`, and are available from the author or Blank's FTP site (pluto.csee.lehigh.edu) as part of the file /rvg/sun4.tar.Z.

Appendix C
Vocabulary Acquisition Test

This Appendix details the lexicon used to test the vocabulary acquisition mechanism (see §9.3).

C.1. Test Lexicon

Paradigms:

BED	$	<sg>
	s	<pl>
FOXES	$	<sg>
	es	<pl>
SHEEP	$	<sg pl>
NOPLUR	$	<sg>
PULL	s	<pres3 pl>
	ed	<past pastpart>
	ing	<prespart>
	$	<inf pres sg>
LOVE	e	<inf pres sg>
	ed	<past pastpart>
	ing	<prespart>
	es	<pres3 pl>
BUILD	d	<inf pres pres3 prespart>
	t	<past pastpart>
CAST	s	<pres3 pl>
	ed	<past>
	ing	<prespart>
	$	<inf pres sg pastpart>

Do	o	<pres inf>
	id	<past pastpart>
	oing	<prespart>
	oes	<pres3>
DRESS	$	<inf pres sg>
	es	<pres3 pl>
	ed	<past pastpart>
	ing	<prespart>
GiVE	i	<sg pl inf pres pres3 prespart pastpart>
	a	<past>
GIVen	e	<inf pres past>
	es	<pres3>
	en	<pastpart>
	ing	<prespart>
FaLL	a	<sg pl inf pres pres3 prespart>
	e	<past pastpart>
FALLen	s	<pres3>
	en	<pastpart>
	ing	<prespart>
	$	<inf pres past>
FiND	i	<sg pl inf pres pres3 prespart>
	ou	<past pastpart>
FINDing	s	<pres3 pl>
	ing	<prespart>
	$	<inf pres past pastpart sg>
FLy	y	<inf pres prespart sg>
	ie	<pres3 pl>
	ew	<past>
	own	<pastpart>
HAve	ve	<pres inf>
	s	<pres3>
	d	<past pastpart>
	ving	<prespart>
KNeW	o	<inf pres pres3 pastpart prespart>
	e	<past>

KNOWn	s	<pres3>
	n	<pastpart>
	ing	<prespart>
	$	<inf pres past>
MAdE	ke	<inf pres>
	kes	<pres3>
	de	<past pastpart>
	king	<prespart>
SiNG	i	<inf pres pres3 prespart>
	a	<past>
	o	<sg pl>
	u	<pastpart>
SiT	i	<inf pres pres3 prespart>
	a	<past>
SLeeP	ee	<pres3 inf pres sg pl>
	e	<past>
SLEPt	t	<past>
	s	<pres3 pl>
	ing	<prespart>
	$	<inf pres sg>
STooD	an	<sg pl inf pres pres3 prespart>
	oo	<past pastpart>
TaKE	a	<sg pl inf pres pres3 prespart pastpart>
	oo	<past>
TAKe	e	<sg inf pres>
	$	<past>
	ing	<prespart>
	en	<pastpart>
	es	<pres3>
TRy	y	<inf pres sg>
	ies	<pres3 pl>
	ied	<past pastpart>
	ying	<prespart>
WRoTE	i	<inf pres pres3 prespart pastpart>
	o	<past>
NN	n	<pastpart prespart>
	$	<pres3 inf pres past pastpart prespart sg pl>

```
TT  t  <pastpart prespart>
    $  <pres3 inf pres past pastpart prespart sg pl>

IforY  i  <pres3 past pastpart pl>
       y  <inf pres prespart sg>

be  be     <inf>
    are    <pres>
    is     <pres3>
    was    <past>
    were   <past>
    been   <pastpart>
    being  <prespart>

LOC  $  <loc>

Modal  $  <past>

Gen  s  <sg>
     $  <pl>
```

Lexicon:

Common NOUNs

activity	cat NOUN	morph activit_IforY__FOXES_
area	cat NOUN	morph area_BED_
arrow	cat NOUN	morph arrow_BED_
assembler	cat NOUN	morph assembler_BED_
assembly	cat NOUN	morph assembl_IforY__FOXES_
barn	cat NOUN	morph barn_BED_
bibliography	cat NOUN	morph bibliograph_IforY__FOXES_
bird	cat NOUN	morph bird_BED_
book	cat NOUN	morph book_BED_
body	cat NOUN	morph bod_IforY__FOXES_
boy	cat NOUN	morph boy_BED_
box	cat NOUN	morph box_FOXES_
block	cat NOUN	morph block_BED_
bucket	cat NOUN	morph bucket_BED_
candy	cat NOUN	morph cand_IforY__FOXES_
chapter	cat NOUN	morph chapter_BED_
chess	cat NOUN	morph chess_FOXES_
command	cat NOUN	morph command_BED_
concept	cat NOUN	morph concept_BED_
communication	cat NOUN	morph communication_BED_
completeness	cat NOUN	morph completeness_NOPLUR_
computer	cat NOUN	morph computer_BED_
context	cat NOUN	morph context_BED_
cpu	cat NOUN	morph cpu_BED_
data	cat NOUN	morph data_SHEEP_

destination	cat NOUN	morph destination_BED_
detail	cat NOUN	morph detail_BED_
direction	cat NOUN	morph direction_BED_
directive	cat NOUN	morph directive_BED_
entry	cat NOUN	morph entr_IforY__FOXES_
ethernet	cat NOUN	morph ethernet_BED_
exercise	cat NOUN	morph exercise_BED_
field	cat NOUN	morph field_BED_
forest	cat NOUN	morph forest_BED_
form	cat NOUN	morph form_BED_
fox	cat NOUN	morph fox_FOXES_
ghost	cat NOUN	morph ghost_BED_
grammar	cat NOUN	morph grammar_BED_
guide	cat NOUN	morph ghost_BED_
history	cat NOUN	morph histor_IforY__FOXES_
horse	cat NOUN	morph horse_BED_
howto	cat NOUN	morph howto_BED_
insight	cat NOUN	morph insight_BED_
instruction	cat NOUN	morph instruction_BED_
intelligence	cat NOUN	morph intelligence_BED_
knowledge	cat NOUN	morph knowledge_NOPLUR_
language	cat NOUN	morph language_BED_
linguistics	cat NOUN	morph linguistics_NOPLUR_
literature	cat NOUN	morph literature_BED_
load	cat NOUN	morph load_BED_
location	cat NOUN	morph location_BED_
manipulation	cat NOUN	morph manipulation_BED_
material	cat NOUN	morph material_BED_
mechanism	cat NOUN	morph mechanism_BED_
machine	cat NOUN	morph machine_BED_
memory	cat NOUN	morph memor_IforY__FOXES_
mind	cat NOUN	morph mind_BED_
mouse	cat NOUN	morph mouse_BED_
name	cat NOUN	morph name_BED_
network	cat NOUN	morph network_BED_
newcomer	cat NOUN	morph newcomer_BED_
operand	cat NOUN	morph operand_BED_
packet	cat NOUN	morph packet_BED_
performance	cat NOUN	morph performance_BED_
pen	cat NOUN	morph pen_BED_
people	cat NOUN	morph people_BED_
person	cat NOUN	morph person_BED_
perspective	cat NOUN	morph perspective_BED_
plane	cat NOUN	morph plane_BED_
pointer	cat NOUN	morph pointer_BED_
purpose	cat NOUN	morph purpose_BED_
pseudoop	cat NOUN	morph pseudoop_BED_
question	cat NOUN	morph question_BED_
reading	cat NOUN	morph reading_BED_

recognition	cat NOUN	morph recognition_BED_
reference	cat NOUN	morph reference_BED_
robot	cat NOUN	morph robot_BED_
room	cat NOUN	morph room_BED_
science	cat NOUN	morph science_BED_
source	cat NOUN	morph source_BED_
sequence	cat NOUN	morph sequence_BED_
statement	cat NOUN	morph statement_BED_
station	cat NOUN	morph station_BED_
step	cat NOUN	morph step_BED_
storage	cat NOUN	morph storage_BED_
student	cat NOUN	morph student_BED_
suggestion	cat NOUN	morph suggestion_BED_
symbol	cat NOUN	morph symbol_BED_
system	cat NOUN	morph system_BED_
table	cat NOUN	morph table_BED_
task	cat NOUN	morph task_BED_
technique	cat NOUN	morph technique_BED_
text	cat NOUN	morph text_BED_
thought	cat NOUN	morph thought_BED_
transport	cat NOUN	morph transport_BED_
tree	cat NOUN	morph tree_BED_
variety	cat NOUN	morph variet_IforY__FOXES_
year	cat NOUN	morph year_BED_

Nouns/Verbs

access	cat NOUN #trans	morph access_DRESS_
address	cat NOUN #trans	morph address_DRESS_
allow	cat #trans	morph allow_PULL_
apply	cat #trans	morph appl_TRy_
approach	cat NOUN #trans	morph approach_DRESS_
assign	cat #bitrans #trans	morph assign_PULL_
attempt	cat INF_ V	morph attempt_PULL_
base	cat #trans	morph bas_LOVE_
begin	cat #trans	morph beg_SiNG_n_NN__FINDing_
believe	cat TRANS THAT_ V	morph believ_LOVE_
broadcast	cat #trans NOUN	morph broadcast_CAST_
build	cat #trans	morph buil_BUILD__FINDing_
building	cat NOUN	morph building_BED_
call	cat #bitrans V	morph call_PULL_
carry	cat #trans	morph carr_TRy_
chase	cat #trans NOUN	morph chas_LOVE_
code	cat NOUN #trans	morph cod_LOVE_
communicate	cat INTRANS V	morph communicat_LOVE_
concern	cat #trans	morph concern_PULL_
deal	cat INTRANS V	morph deal_PULL_
decide	cat INTRANS INF_ THAT_ V NOUN	morph decid_LOVE_
describe	cat #trans	morph describ_LOVE_
develop	cat #trans	morph develop_PULL_

design	cat #trans	morph design_PULL_
desire	cat #trans NOUN	morph desir_LOVE_
draw	cat INTRANS V	morph draw_PULL_
distribute	cat #trans	morph distribut_LOVE_
estimate	cat #trans	morph estimat_LOVE_
execute	cat TRANS INTRANS V	morph execut_LOVE_
fall	cat INTRANS V	morph f_FaLL_ll_FALLen_
find	cat #trans	morph f_FiND_nd_FINDing_
fly	cat INTRANS #trans NOUN	morph fl_FLy__FINDing_
focus	cat NOUN #trans	morph focus_PULL_
follow	cat #trans	morph follow_PULL_
gain	cat #trans	morph gain_PULL_
go	cat INTRANS V	morph go_PULL_
graduate	cat NOUN #trans	morph graduat_LOVE_
guide	cat NOUN #trans	morph guid_LOVE_
hate	cat #trans	morph hat_LOVE_
help	cat TRANS INF_ V NOUN	morph help_PULL_
identify	cat #trans	morph identif_TRy_
include	cat #trans	morph includ_LOVE_
introduce	cat #trans	morph introduc_LOVE_
involve	cat #trans	morph involv_LOVE_
issue	cat NOUN #trans	morph issu_LOVE_
kick	cat TRANS V	morph kick_PULL_
know	cat TRANS INTRANS THAT_ INF_ V	morph kn_KNeW_w_KNOWn_
label	cat NOUN #trans	morph label_PULL_
limit	cat NOUN #trans	morph limit_PULL_
like	cat #prep	morph like_LOC_
like	cat #trans	morph lik_LOVE_
line	cat NOUN #trans	morph lin_LOVE_
love	cat #trans	morph lov_LOVE_
master	cat NOUN #trans	morph master_PULL_
mean	cat NOUN #trans	morph mean_PULL_
model	cat NOUN #trans	morph model_PULL_
motivate	cat #trans	morph motivat_LOVE_
move	cat NOUN TRANS INTRANS V	morph mov_LOVE_
obtain	cat #trans	morph obtain_PULL_
organize	cat #trans	morph organiz_LOVE_
outline	cat NOUN #trans	morph outlin_LOVE_
perform	cat #trans	morph perform_PULL_
point	cat NOUN #trans	morph point_PULL_
process	cat NOUN TRANS INTRANS V	morph process_DRESS_
program	cat NOUN #trans	morph program_PULL_
provide	cat #trans	morph provid_LOVE_
produce	cat #trans	morph produc_LOVE_
put	cat #trans	morph put_TT__FINDing_
race	cat INTRANS #trans	morph rac_LOVE_
read	cat #trans	morph read_PULL_
represent	cat #trans	morph represent_PULL_

require	cat #trans	morph requir_LOVE_
say	cat INTRANS TRANS THAT_ V	morph say_PULL_
set	cat #trans	morph set_TT__FINDing_
speak	cat #trans	morph speak_PULL_
squeak	cat INTRANS V	morph squeak_PULL_
sigh	cat INTRANS V	morph sigh_PULL_
sing	cat INTRANS TRANS #bitrans V	morph s_SiNG_ng_FINDing_
sit	cat INTRANS V	morph s_SiT_t_TT__FINDing_
sleep	cat INTRANS V	morph sl_SLeeP_p_SLEPt_
structure	cat #trans	morph structur_LOVE_
study	cat NOUN #trans	morph stud_TRy_
supplement	cat #trans NOUN	morph supplement_PULL_
switch	cat #bitrans V	morph switch_PULL_
think	cat #trans	morph think_PULL_
time	cat NOUN #trans	morph tim_LOVE_
try	cat INTRANS #trans	morph tr_TRy_
translate	cat #trans	morph translat_LOVE_
understand	cat #trans	morph underst_STooD_d_FINDing_
understanding	cat NOUN	morph understanding_BED_
use	cat NOUN TRANS INF_ V	morph us_LOVE_
want	cat TRANS INF_ V	morph want_PULL_
work	cat NOUN #trans	morph work_PULL_
write	cat TRANS INTRANS V	morph wr_WRoTE_t_TT__GIVen_

Some verbal idioms follow: "give" and some of its idioms...

give	cat #bitrans XO_ XIO_ V	morph g_GiVE_v_GIVen_
pull	cat #trans	morph pull_PULL_
take	cat #trans	morph t_TaKE_k_TAKe_
make	cat #trans	morph ma_MAdE_

auxiliary verbs

must	cat MODAL	morph must_Modal_
should	cat MODAL	morph should_Modal_
can	cat MODAL	morph can_Modal_
may	cat MODAL	morph may_Modal_
do	cat DO #trans	morph d_Do_
be	cat BE	morph _be_
will	cat MODAL	morph will_Modal_
would	cat MODAL	morph would_Modal_
haveAux	cat HAVE	morph ha_HAve_
have	cat TRANS INF_ V	morph ha_HAve_

prepositions

about	cat #prep	morph about_LOC_
among	cat #prep	morph among_LOC_
as	cat #prep	morph as_LOC_
for	cat #prep	morph for_LOC_
from	cat #prep	morph from_LOC_
in	cat #prep	morph in_LOC_
into	cat #prep	morph into_LOC_

on	cat #prep	morph on_LOC_
past	cat #prep	morph past_LOC_
through	cat #prep	morph through_LOC_
under	cat #prep	morph under_LOC_
with	cat #prep	morph with_LOC_
without	cat #prep	morph without_LOC_
to	cat IOBJ	
to	cat INF LINF RINF	
to	cat #prep	morph to_LOC_
by	cat PASSIVEBY #prep	morph by_LOC_
of	cat OFNP OFOBJ	

wh-type words

what	cat WH WHDET
which	cat WH #rel
who	cat WH #rel

negative particles

never	cat NEG
not	cat NEG
always	cat NEG

punctuation marks

.	cat CLOSE	
?	cat CLOSE	
,	cat COMMA0 COMMA1 COMMA2	
'	cat GEN	morph '_Gen_

adjectives

actual	cat ADJ
angry	cat ADJ CADJ
artificial	cat ADJ CADJ
available	cat ADJ CADJ
brief	cat ADJ CADJ
broad	cat ADJ CADJ
computing	cat ADJ
computational	cat ADJ CADJ
cognitive	cat ADJ CADJ
different	cat ADJ CADJ
digital	cat ADJ CADJ
extensive	cat ADJ CADJ
further	cat ADJ
fat	cat ADJ CADJ
heavy	cat ADJ CADJ
human	cat ADJ CADJ
large	cat ADJ CADJ
later	cat ADJ
linguistic	cat ADJ CADJ
local	cat ADJ CADJ
mental	cat ADJ CADJ
natural	cat ADJ CADJ
next	cat ADJ CADJ
other	cat ADJ

particular	cat ADJ CADJ
practical	cat ADJ CADJ
pragmatic	cat ADJ CADJ
purple	cat ADJ CADJ
proverbial	cat ADJ CADJ
receiving	cat ADJ
relevant	cat ADJ CADJ
red	cat ADJ CADJ
safe	cat ADJ CADJ
same	cat ADJ CADJ
small	cat ADJ CADJ
smart	cat ADJ CADJ
symbolic	cat ADJ CADJ
subsequent	cat ADJ
rulegoverned	cat ADJ CADJ
useful	cat ADJ CADJ
wide	cat ADJ CADJ

Other categories of adjective

aware	cat CADJOF
English	cat NOUN ADJ
major	cat NOUN ADJ

names

George	cat NAME	morph george_BED_
Martha	cat NAME	morph martha_BED_
Mary	cat NAME	morph mary_BED_

articles

a	cat INDEF
an	cat INDEF
any	cat DEF
each	cat DEF
the	cat DEF
this	cat PRON DEF
these	cat PRON DEF
that	cat CTHAT #rel PRON DEF
there	cat THERE
how	cat THERE

pronouns

I	cat PRON
you	cat PRON
he	cat PRON
she	cat PRON
it	cat PRON
we	cat PRON
they	cat PRON
me	cat PRON
him	cat PRON
us	cat PRON
them	cat PRON
my	cat GENPRON
your	cat GENPRON

his	cat GENPRON
her	cat GENPRON
its	cat GENPRON
our	cat GENPRON
their	cat GENPRON

numbers

single	cat NUMBER
many	cat NUMPRON NUMBER
some	cat NUMPRON NUMBER
much	cat NUMPRON NUMBER
several	cat NUMBER
two	cat NUMBER
four	cat NUMBER
forty	cat NUMBER

adverbs

also	cat ADV
directly	cat ADV
easily	cat ADV
heavily	cat ADV
locally	cat ADV
however	cat ADV

conjunctions

and	cat CONJ0
but	cat CONJ0
since	cat CONJ0

C.2. Sentence Testbed

What follows is a list of the test sentences from Appendix G of *Efficient Parsing for Natural Language* (Tomita, 1987).

(51) The assembly language provides a means for writing a program without having to be concerned with actual memory addresses.

(52) It allows the use of symbolic codes to represent the instructions.

(53) Labels can be assigned to a local instruction step in a source program to identify that step as an entry point for use in subsequent instructions.

(54) Operands which follow each instruction represent storage locations.

(55) The assembly language also includes assembler directives that supplement the machine instruction.

(56) A packet is a statement which is not translated into a machine instruction.

(57) A program written in assembly language is called a source program.

(58) It consists of symbolic commands called statements.

(59) Each statement is written on a single line, and it may consist of four entries.

(60) The source program is processed by the assembler to obtain a machine language program that can be executed directly by the CPU.

(61) Ethernet is a broadcast communication system for carrying digital data packets among computing stations which are locally distributed.

(62) The packet transport mechanism provided by Ethernet has been used to build systems which can be local computer networks.

(63) Switching of packets to their destinations on the Ethernet is distributed among the receiving stations using packet address recognition.

(64) A model for estimating performance under heavy tables is included for completeness.

(65) In writing this book, I had several purposes in mind.

(66) It is a practical book for students who are following graduate work in computer networks.

(67) It includes instructions identified to allow the student to use a network of computers.

(68) It is a practical book for people who are building computer systems that model with natural language.

(69) It is not assigned as a source book, but it provides the practical steps in data, and it includes an actual outline of English language.

(70) It is a practical source with many directives into the communication of language.

(71) I have tried to include a large table of locations to provide students with digital processes on the network.

(72) Each step includes statements for symbolic processing, and there is a smart machine.

(73) However, I have tried to limit the references to easily available material.

(74) This is a book about human language.

(75) It is motivated by two questions.

(76) What knowledge must a person have to speak language?

(77) How is the mind organized to make use of this knowledge in communicating?

(78) In looking at language as a cognitive process, we deal with issues that have been the focus of linguistic study of many years, and this book includes insights gained from these studies.

(79) We look at language from a different perspective.

(80) In forty years, since digital computers were developed, people have programed them to perform many activities that we think of as requiring some form of intelligence.

(81) Our study of the mental processes involved in language draws heavily on concepts that have been developed in the area called artificial intelligence.

(82) It is safe to say that much of the work in computer science has been pragmatic, based on a desire to produce computer programs that can perform useful tasks.

(83) The same concept of program can be applied to the understanding of any system which is executing processes that can be understood as the rulegoverned manipulation of symbols.

(84) The next chapter sets the computational approach into the context of other approaches by giving a brief history of the major directions in linguistics.

(85) In performing a mental task like deciding on a chess move, we are aware of going through a sequence of thought process.

(86) Draw.

(87) Do it.

(88) I have a pen.

(89) I must not do that.

(90) Time flies like an arrow.

Appendix D
UNIX File Management Parser

Boundaries: Clause NP NPmod Obj Pred Prep Subj Tense Topic

Ordering Features: SENT S TENS BE AF V O PASS THAT VSUB NP DET HEAD NEND REL ROLE PREP GAP RELEND

Default cond: -NP..DET

Properties: <first second third>
<pl sg>
<past pres>
imperative inf nom pastpart pp prespart

Semantic Roles: inh subj obj dat — rel1 rel2

Productions:

clausal

```
BE     L  cond +BE -AF ?NP -HEAD
          change -BE +AF +NP
          Tense new
          Tense = lex
CLOSE  I  cond -S..O ?NP -HEAD -ROLE -GAP..RELEND
          change +SENT..O -PASS..VSUB +NP -DET..RELEND
IMP    N  cond +TENS ?NP -HEAD -REL -GAP
          change -S..AF
          lexprop <inf>
          save Tense
          Pred agree lex
          Pred addprop <imperative>
PASS   N  cond -S -BE +AF..V ?NP -HEAD
          change -AF -O +PASS -NP
          lexprop <pastpart>
PROG   N  cond -S -BE +AF..V -PASS ?NP -HEAD
          change -AF +NP
          lexprop <prespart>
QUES   N  cond +TENS ?NP -HEAD -REL..ROLE
          change -TENS -AF -NP
          lexprop <past pres>
          save Tense
```

```
S       N  cond +SENT ?NP..DET
           change -SENT
           Pred new
           Main := Pred
           Clause := Main
SUBJ    N  cond +S -TENS ?NP -HEAD +ROLE
           change -S -DET..ROLE
           save Subj
           Subj := NP
           Subj agree Pred
           NPmod := Subj
TENS    N  cond +TENS ?NP -HEAD..NEND +ROLE
           change -TENS..AF
           lexprop <past pres>
           save Tense
           Pred agree lex
```
verb subcategories
```
INTRAN  S V  cond -S -PASS ?NP -HEAD
             change -O +VSUB
THAT_   S V  cond -S ?NP -HEAD
             change +THAT..VSUB
TRANS   S V  cond -S ?NP -HEAD
             change +VSUB..NP
```
verbs and predicates
```
CADJ  L       cond -S -BE +AF..V ?NP -HEAD
              change -AF..O -NP
PP    S PREP  cond -S..BE +AF..V ?NP -HEAD
              change -AF..O -NP +PREP
              save Prep
              Prep = lex
              NP new
V     L       cond -S -AF +V +VSUB ?NP -HEAD
              change -BE -V -VSUB
              save Pred
              Pred = lex
              Subj = Pred.subj
```
verb compliments
```
CTHAT  L  cond -V +THAT ?NP -HEAD
          change +S..O -PASS..VSUB +NP -DET..PREP -RELEND
OBJ    N  cond -V +O ?NP -HEAD +ROLE
          change -O -DET..ROLE
          save Obj
          Obj := NP
          Obj = Pred.obj
          NPmod := Obj
```

```
PASSBY  L       cond -V..O +PASS ?NP -HEAD
                change +NP..HEAD +PREP
PPEND   N       cond ?NP -HEAD +ROLE..PREP
                change -DET..PREP
                save NPmod
                NP = Prep.obj
                Prep addprop <pp sg:third>
                Pred → Prep
PREP    L       cond ?NP -HEAD +PREP
                change +DET..HEAD
VREL1   S PREP  cond -V..O ?NP -HEAD -PREP
                change -THAT +PREP
                save Prep
                Prep = lex
                NP new
```

noun phrase productions

```
ADJ     L   cond ?DET +HEAD
            change
            NP → lex
DET     L   cond ?NP +DET..HEAD
            change -DET
EXT     L   cond ?NP..DET +HEAD
            change -DET..HEAD +NEND..ROLE
            save NPmod
            NP = lex
NAME    L   cond ?NP +DET..HEAD
            change -DET..HEAD +NEND..ROLE
NOISE   L   cond ?NP..DET
            change
NOUN    L   cond ?NP..DET +HEAD
            change -DET..HEAD +NEND..ROLE
            lexprop <nom>
            save NPmod
            NP agree lex
            NP = lex
NP      N   cond -VSUB +NP -HEAD -PREP
            change -NP +DET..HEAD
            save NP
            NP new
NPEND   N   cond ?NP -HEAD +NEND -PREP
            change -DET..NEND +ROLE
            save NPmod
```

```
REDREL  N  cond +S..O ?NP -HEAD +REL +GAP
           change -S..BE -NP -REL +ROLE -GAP
           save NP
           Subj := Topic
RELC0   N  cond +O ?NP -HEAD +NEND -RELEND
           change +S..O -PASS..VSUB +NP -DET..NEND +REL -ROLE..PREP +GAP..RELEND
           shiftdown
           Topic := N̂P
           NP := Topic
           Pred new
RELEND  N  cond -S..O ?NP -HEAD -ROLE -GAP +RELEND
           change -DET..REL
           returnup
RELR0   N  cond -S..O ?NP -HEAD +NEND -GAP
           change +S..O -PASS..VSUB +NP -DET..NEND +REL -ROLE..PREP +GAP
           save Clause
           Topic := NP
           Pred new
questions
NGAP  N  cond +NP ?DET -REL +GAP
         change -NP..REL +ROLE -GAP
WH    L  cond +S ?NP -HEAD -GAP
         change -NP +GAP
         save Topic
```

Non-lexical Ordering: S SUBJ QUES TENS IMP PROG PASS PPEND OBJ NP NPEND RELR0 RELC0 REDREL RELEND NGAP

Paradigms

```
AM $ <sg:pres:first>

ARE $ <pres:pl pres:second>

BE ing <prespart>
   en <pastpart>
   $ <inf>

BED s <nom third:pl>
    $ <nom sg:third>

FiND ou <past pastpart>
     i <inf pres prespart>

FINDing ing <prespart>
        s <sg:third:pres>
        $ <first inf past pastpart pl second>

IS $ <sg:third:pres>
```

```
LOVE ing <prespart>
     ed <past pastpart>
     es <sg:third:pres>
     e <inf pres:first pres:pl pres:second>

PULL ing <prespart>
     ed <past pastpart>
     s <sg:third:pres>
     $ <inf pres:first pres:pl pres:second>

WAS ere <past:pl second:past>
    s <sg:past:first sg:third:past>
```

Lexicon

```
 preposition
 by    cat VREL1 PP PREP PASSBY  morph 'by'
 from  cat VREL1 PP PREP         morph 'from'
 in    cat VREL1 PP PREP         morph 'in'
 on    cat VREL1 PP PREP         morph 'on'
 common noun
 extension  cat NOUN  morph 'extension_BED_'
 file       cat NOUN  morph 'file_BED_'
 noun/verb
 like  cat VREL1 PP PREP NOUN TRANS V  morph 'lik_LOVE_'
                                       m 'like_BED_'
 verb
 be    cat INTRAN V BE        morph 'am_AM_'
                              m 'be_BE_'
                              m 'is_IS_'
                              m 'are_ARE_'
                              m 'w_WAS_'
 end   cat INTRAN V           morph 'end_PULL_'
 find  cat TRANS V            morph 'f_FiND_nd_FINDing_'
 list  cat TRANS V            morph 'list_PULL_'
 open  cat TRANS INTRAN V     morph 'open_PULL_'
 show  cat TRANS V            morph 'show_PULL_'
 wh word
 what  cat WH  morph 'what'
 punctuation
 . cat  CLOSE  morph '.'
 ? cat  CLOSE  morph '?'
 adjective
 all  cat ADJ  morph 'all'
 my   cat ADJ  morph 'my'
 determiner
 a     cat DET         morph 'a'
 an    cat DET         morph 'an'
 that  cat DET CTHAT   morph 'that'
 the   cat DET         morph 'the'
```

file type adjective

binary	cat ADJ	morph 'binary'
C	cat ADJ	morph 'c'
compressed	cat ADJ	morph 'compressed'
Cplusplus	cat ADJ	morph 'c++'
executable	cat ADJ	morph 'executable'
image	cat ADJ	morph 'image'
latex	cat ADJ	morph 'latex'
object	cat ADJ	morph 'object'
smalltalk	cat ADJ	morph 'smalltalk'
Tar	cat ADJ	morph 'tar'
text	cat ADJ	morph 'text'

file extension

c	cat EXT	morph 'c'
cplusplus	cat EXT	morph 'cc'
im	cat EXT	morph 'im'
o	cat EXT	morph 'o'
st	cat EXT	morph 'st'
tar	cat EXT	morph 'tar'
tex	cat EXT	morph 'tex'
txt	cat EXT	morph 'txt'
Z	cat EXT	morph 'Z'

Appendix E
File Management Application

This appendix contains the C++ source code for the UNIX file management application that uses the RV parser in Appendix D to parse commands.

E.1. ls-client.cc

```
// NL front end to the ls command
//
// Dave Astels  9-sep-92

#include <stream.h>
#include <GetOpt.h>
#include "Socket.H"
#include "DirectoryLister.H"
#include "SemEntry.H"

int debug = 0;

void
usage ()
{
  fprintf (stderr, "usage: ls-client [-d] <hostname> <port>\n");
  exit (2);
}


void
main (int argc, char ** argv)
{
  DirectoryLister lister;
  String sentence;
  int numToRead;

  if ( argc < 3 )
```

```
    usage();

  GetOpt options (argc, argv, "d");
  char opt_char;
  while ( (opt_char = options()) != EOF )
    switch ( opt_char ) {
    case 'd': debug = 1; break;
    case '?': usage(); break;
    }

  Socket skt (argv[options.optind], atoi (argv[options.optind + 1]));

  if ( debug )
    cerr << "Made connection to RV server.\n";

  while (1) {
    cout << "\nls>";
    String::delimiters ("\n");
    cin >> sentence;
    if ( sentence[0] == '.' )
      break;
    skt.writeString (sentence);
    numToRead = skt.readInteger();
    if ( numToRead == 0 )
      cout << "I don't understand that!\n";
    else {
      if ( debug )
        cerr << "Reading " << numToRead << " characters from the server.\n";
      String command = skt.readString (numToRead);
      istream cs (numToRead, (char *)command);
      char buffer [1023];
      cs.getline (buffer, 1023);            // strip off the production trace
      SemEntry commandStructure (cs);
      if ( debug )
        commandStructure.printOn (cout);
      lister.process (commandStructure);
    }
  }
}
```

E.2. DirectoryLister.cc

```
// Directory lister class
// Encapsulates NL access to the 'ls' command
//
// Accepts semantic structures output by the RV-Tools parser.
//
// Dave Astels  10-sep-92
```

```
#include "Set_String.h"
#include "Dictionary.t"
#include "SemEntry.H"

DECLARE_ONCE Dictionary<Set<String>>;

class DirectoryLister {
public:
  DirectoryLister ();
  void process (SemEntry & command);

private:
  void addTypeAndExtension (String & type, String & extension);
  Set<String> & whatIs (String & fname);
  void listFilesOfType (String & type);

  Dictionary<Set<String>> typeToExtension;
  Dictionary<Set<String>> extensionToType;
};

//########################################

//%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%


extern int debug;

DirectoryLister::DirectoryLister ()
{
//  typeToExtension["text"].add ("txt");
//  typeToExtension["text"].add ("tex");
}



void
DirectoryLister::process (SemEntry & command)
{
  String & verb = command.getName();
  if ( verb == "list" | verb == "show" ) {
    Set<String> extensions;
    Dictionary<SemEntry> & fileMods = command["obj"].getRoles();
    for ( DictionaryIterator i (fileMods); i(); i++ ) {
      String type = ((SemEntry *)i()->get_value())->getName();
      if ( typeToExtension.contains (type) )
        listFilesOfType (type);
    }
  }
  else if ( verb == "end" ) {
    String type = (command["subj"])["mod1"].getName();
    String ext = (command["mod1"])["obj"].getName();
    addTypeAndExtension (type, ext);
  }
  //...

}


void
DirectoryLister::addTypeAndExtension (String & type, String & extension)
{
  if ( debug )
    cerr << "Adding type (" << type << ") and extension (" << extension << ")\n";

  if ( !typeToExtension[type].contains (extension) )
    typeToExtension[type].add (extension);

  if ( !extensionToType[extension].contains (type) )
    extensionToType[extension].add (type);
}


Set<String> &
DirectoryLister::whatIs (String & fname)
{
}


void
DirectoryLister::listFilesOfType (String & type)
{
    String lsCommand = "ls ";

    for ( VectorIterator<String> vi (typeToExtension[type]); vi(); vi++ ) {
      lsCommand += "*.";
      lsCommand += *vi();
      lsCommand += " ";
    }

    if ( debug )
      cerr << lsCommand << '\n';
    else
      system ((char *)lsCommand);
}
```

E.3. SemEntry.cc

```
// Class excapsulating semantic structures.
//
// Dave Astels  12-sep-92

#include <MyString.h>
#include "Set.t"
#include "Dictionary.t"
#include "VectorIterator.t"

class SemEntry;

DECLARE_ONCE Set<String>
DECLARE_ONCE VectorIterator<String>
DECLARE_ONCE Dictionary<SemEntry>

class SemEntry {
public:

// Constructors

  SemEntry () : name(), properties(), semRoles() {}
  SemEntry (SemEntry & s) : name (s.name), properties (s.properties), semRoles (s.semRoles) {}
  SemEntry (istream &);

// Access

  String & getName ();
  Dictionary<SemEntry> & getRoles ();
  SemEntry & operator [] (String &);

// Printing

  void printOn (ostream &, int = 0);

private:
  String name;
  Set<String> properties;
  Dictionary<SemEntry> semRoles;
};

//########################################

INLINE String &
SemEntry::getName ()
{
  return name;
}
```

```
INLINE Dictionary<SemEntry> &
SemEntry::getRoles ()
{
  return semRoles;
}


INLINE SemEntry &
SemEntry::operator [] (String & role)
{
  return semRoles[role];
}


//%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%


SemEntry::SemEntry (istream & s)
{
  char lookahead, dummy;

  String::delimiters ("\n\t\n\r ");
  s >> lookahead;
  if ( lookahead != '\(' )
    s.unget (lookahead);

  s >> name >> dummy;
  s.unget (dummy);

  if ( lookahead == '\(' ) {
    s.skip (0);
    s >> lookahead;
    s.skip (1);
    while ( lookahead != '\n' ) { // get properties
      s.unget (lookahead);
      String prop (s);
      properties.add (prop);
      s.skip (0);
      s >> lookahead;
      s.skip (1);
    };

    s.unget (lookahead);
    s >> lookahead;
    while ( lookahead != '\)' ) {
      s.unget (lookahead);
      String role (s);             // get a role name
      SemEntry filler (s);
      semRoles[role] = filler;
      s >> lookahead;
    }
  }
}


void
SemEntry::printOn (ostream & s, int level)
{
  if ( !properties.empty() )
    s << '\(';

  name.pretty_print (s);

  if ( !properties.empty() ) {
    String indent ('\t', level);
    String indent1 ('\t', level + 1);
    s << '\n';
    indent1.pretty_print (s);
    for ( VectorIterator<String> vi (properties); vi(); vi++ ) {
      ((String *)vi())->pretty_print (s);
      s << ' ';
    }
    s << '\n';
    for ( DictionaryIterator di (semRoles); di(); di++ ) {
      indent1.pretty_print (s);
      di()->get_key().pretty_print (s);
      s << '\t';
      ((SemEntry *)(di()->get_value()))->printOn (s, level + 1);
    }
    indent.pretty_print (s);
    s << '\)';
  }
  s << '\n';
}
```

E.4. Socket.cc

```
// Socket encapsulating class
//
// Dave Astels  9-sep-92

#include <stdio.h>
#include <MyString.h>

class Socket {
```

```
public:
  Socket (String & hostName, int port);
  ~Socket ();

  String readString (int maxLength);
  int readInteger ();
  void writeString (String & str);
  void eoln ();

private:
  int skt;                          // file descriptor
};


//##########################################

INLINE
Socket::~Socket ()
{
  eoln();
  close (skt);
}


INLINE void
Socket::eoln ()
{
  writeString ("\n");
}

//%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

extern "C" {
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
}

extern "C" struct hostent *gethostbyname();

#include <stream.h>

Socket::Socket (String & hostName, int port)
{
  struct sockaddr_in server;
  struct hostent *hp;

  skt = socket (AF_INET, SOCK_STREAM, 0);
  if ( skt < 0 ) {
    perror ("opening stream socket");
    exit (1);
  }

  server.sin_family = AF_INET;
  hp = gethostbyname ((char *)hostName);
  if ( hp == 0 ) {
    cerr << hostName << " unknown host\n";
    exit (2);
  }

  bcopy ((char *)hp->h_addr, (char *)&server.sin_addr, hp->h_length);
  server.sin_port = htons (port);

  if ( connect (skt, (struct sockaddr *)&server, sizeof(server)) < 0 ) {
    perror ("connecting stream socket");
    exit (1);
  }
}


String
Socket::readString (int maxLength)
{
  char buffer [maxLength + 1];

  bzero (buffer, maxLength + 1);
  if ( read (skt, buffer, maxLength) < 0 ) {
    perror ("reading stream socket");
    exit (1);
  }

  return String (buffer);
}



int
Socket::readInteger ()
{
  unsigned char buffer [2];

  if ( read (skt, buffer, 2) < 0 ) {
    perror ("reading stream socket");
    exit (1);
  }

  return buffer[0] * 256 + buffer[1];
}
```

```
void
Socket::writeString (String & str)
{
  if ( write (skt, str, str.length()) < 0 || write (skt, "\n", 1) < 0 ) {
    perror ("writing on stream socket");
    exit (1);
  }
}
```

APPENDIX F

User Manual

F.1. Introduction

This manual describes the operation and use of the set of RV development tools that are part of the `RV-Tools` system. They are:

- Launcher
- Grammar Browser
- Lexicon Browser
- Lexical Trie Browser
- Debugger

In addition to these, `RV-Tools` includes a fairly full RV parser engine currently implementing all features of Blank's system except discontinuous idioms.

`RV-Tools` was implemented using Smalltalk-80 release 4 on Sun workstations.

Familiarity with both RV (Astels, 1991; Blank, 1989; Blank, 1991; Blank, 1991; Blank & Kasson, 1989; Blank & Owens, 1990) and the *Objectworks\Smalltalk r4* environment (LaLonde & Pugh, 1990; LaLonde & Pugh, 1990; Systems, 1990) is assumed.

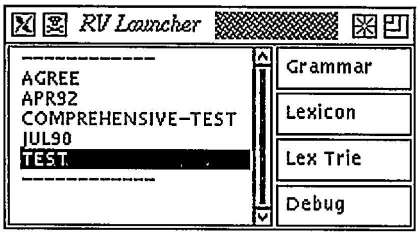


FIGURE F.1. Launcher Window

F.2. Launcher

The Launcher is the centerpiece of the tool set. It is used to organize and access parsers, and to invoke the browsers and debugger. To start the Launcher execute `RVLauncher open`. The Launcher window is shown in Figure F.1. It consists of two areas: the parser list and the tool buttons.

F.2.1. Parser List. Parsers are stored in a global dictionary called "`RVSystems`", keyed by their names.

The parser list contains a list of the names of all parsers that are currently a part of the system. These parsers are stored in the Smalltalk image, and as such are saved along with the image. A parser must be selected from the list before any operation can be performed using it. The parser list has an *operate* menu providing file input/output, parser manipulation, and utility functions:

print out: generates a readable representation of the selected parser in a file whose name is requested.

file out: saves the selected parser in a file whose name is requested (the file must end with "`.rv`". The parser is saved as chunks of Smalltalk code.

load: provides the user with a list of files ending in "`.rv`". There is a facility for changing the directory. Selecting a file and clicking the `Load` button causes the parser saved in the selected file to be loaded into the image. Its name will be added to the parser list in the launcher. Files ending with "`.rv`" are assumed to have been written using the `file out` option of the launcher. Such files consist of a parser described by chunks of Smalltalk code, which can be executed a chunk at a time to reconstruct the parser.

add parser: prompts for the name of the new parser. It then creates an empty parser, and adds its name to the list.

rename: prompts for a new name for the selected parser.

remove: removes a parser from the system, first verifying that it should.

copy: allows the user to make a copy of a parser. Copying a parser is useful when experimental changes are to be tried, and the original parser should be retained.

update: brings the parser list up-to-date with the current state of the system. This is useful if there are two launchers active, which is highly unusual and undesirable, or if the system changes due to code being executed in a text view such as a workspace.

F.2.2. Tool Buttons. To the right of the parser list are the three tool buttons. These are used to invoke the browsers and debugger. Clicking on a tool button invokes the corresponding tool with the parser whose name is selected in the parser list. Nothing is done if no parser is selected.

F.3. Grammar Browser

The grammar browser provides access to the aspects of a parser that are directly related to syntax. This includes:

- boundary registers
- ordering features

- ordering feature macros
- production categories
- productions
- non-lexical production ordering

The window for a grammar browser is shown in Figure F.2. It is divided into six major areas:

Boundary view: at the top left corner of the window

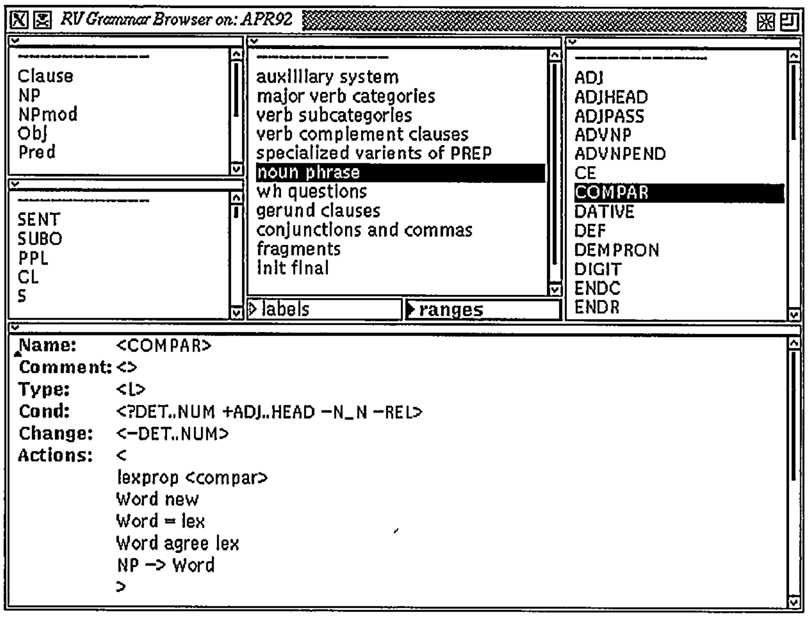
Ordering feature view: below the boundary register view

Production category view: top center

Vector format buttons: below the production category view

Production view: top right

Editing view: across the bottom


RV Grammar Browser on: APR92

Clause
NP
NPmod
Obj
Pred

SENT
SUBO
PPL
CL
S

auxiliary system
major verb categories
verb subcategories
verb complement clauses
specialized varients of PREP
noun phrase
wh questions
gerund clauses
conjunctions and commas
fragments
init final

ADJ
ADJHEAD
ADJPASS
ADVNP
ADVNPEND
CE
COMPAR
DATIVE
DEF
DEMPRON
DIGIT
ENDC
ENDR

```
Name:     <COMPAR>
Comment:  <>
Type:     <L>
Cond:     <?DET..NUM +ADJ..HEAD -N_N -REL>
Change:   <-DET..NUM>
Actions:  <
          lexprop <compar>
          Word new
          Word = lex
          Word agree lex
          NP -> Word
          >
```

FIGURE F.2. Grammar Browser Window

F.3.1. Boundary View. The boundary view contains a list of alphabetically sorted boundary names. The *operate* menu is very basic:

add boundary: prompts for a boundary name and adds it to the list.

rename: prompts for a new name, and replaces the selected boundary with the one the user enters. All references to the selected boundary are updated.

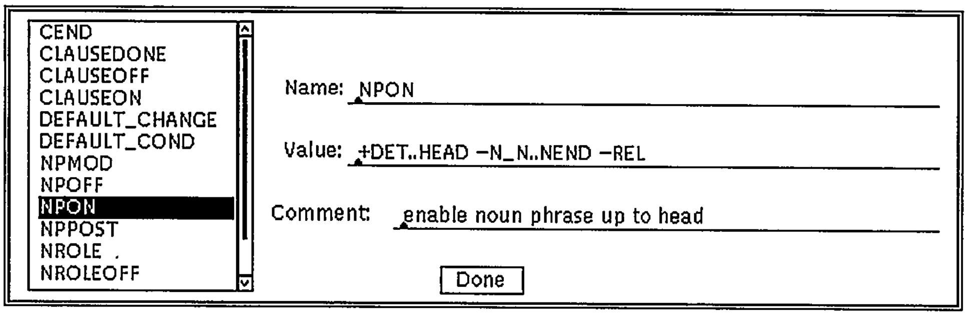
remove: removes the selected boundary from the list, after first verifying that it should be removed, and that it is not referred to by any action.

spawn production: opens a production browser on those productions that manipulate the selected boundary.

When a boundary name is selected, an associated comment can be edited in the editing view.

F.3.2. Ordering Feature View. This view operates much like the boundary view. It contains a list, and has a similar *operate* menu. The `spawn production` option opens a production browser on those productions that assigned a + or - value to the selected feature in either the condition or change vector. The ordering of this list is significant, as it defines labels for each position in ordering feature vectors. Adding a feature, places it before the selected feature, or at the end if there is no selection. Also, a comment for the selected feature is available for editing in the editing view.

F.3.2.1. *Feature macro editor.* The grammar browser supports the facility of feature macros. This provides a shorthand for specifying condition and change vectors of productions. The macro editor is a separate pop-up dialog that is invoked from the category view's *operate* menu. This dialog is shown in Figure F.3. The list at the far left contains the names of all defined ordering feature macros. The text fields


CEND
CLAUSEDONE
CLAUSEOFF
CLAUSEON
DEFAULT_CHANGE
DEFAULT_COND
NPMOD
NPOFF
NPON
NPPOST
NROLE .
NROLEOFF

Name: NPON

Value: +DET..HEAD -N_N..NEND -REL

Comment: enable noun phrase up to head

FIGURE F.3. Ordering Feature Macro Editor

to the right of this allow the user to enter a macro name, value, and optionally a comment. The name is automatically converted to uppercase. The value is a ternary vector using labels and ranges, and can include any macros that have been previously defined. The `<tab>` key can be used to move from field to field, or the pointer can be clicked on a specific field. Pressing `<return>` causes the displayed macro to be added to the list, possibly replacing an existing macro of the same name. When macros are used in vectors, they are replaced with their definitions when the vector is accepted. Changing a macro does not change vectors that were previously defined.

When a macro name is selected from the list, its definition is displayed in the text fields. It can then be edited.

There are two special macros that must exist. They are predefined to be an empty vector: `DEFAULT_COND` and `DEFAULT_CHANGE`. These are implicitly at the far left of every production's condition and change vectors, respectively.

When the user has finished using the macro editor, they click the `Done` button. Currently, the macro editor is a blocking dialog.

F.3.3. Production Category View. For the grammar designer's convenience productions can be grouped into categories which can be given descriptive names. A list of these category names are displayed in the category view. When a category is selected, the names of any productions in that category are displayed in the production view (see § F.3.5). Also, a production template is displayed in the editing view.

The *operate* menu for this view includes category related functions, editing view control functions, and utility functions. A description of each menu option follows:

add category: prompts for a new category name and adds it to the list. The new category is inserted before the selected one, or at the end if none are selected.

rename: allows the name of the selected category to be changed.

remove: removes a category. If it is empty, it is quietly removed. If, however, there are productions in the category, the user is asked if those productions should be removed. If the user responds positively then they are removed from the grammar, and the category is removed.

update: brings the browser up-to-date. This is useful if there is more than one grammar browser open on the same parser: changes made in one browser do not appear automatically in others.

definition: displays the definition of the selected production in the editing view, or a production template if no production is selected.

category structure: displays in the editing view a list of all categories and the productions that are in each. This list is made up of parenthesized entries. The first element in each entry is the category name, enclosed in braces. Following this are the productions in that category. There is an entry for each category.

non-lexicals: displays a list of all non-lexical productions, in the order in which they are searched. This list is then edited to modify the search order.

edit feature macros: invokes the macro editor described in §F.3.2.1.

find production: prompts for the name of a production, and attempts to find that production in the grammar. If one is found, the category and production lists are positioned to display it and its definition is displayed in the editing view. If no production can be found, an alert box informs the user.

F.3.4. Vector Format Buttons. The grammar browser can accept/display ternary vectors with or without ranges. This is controlled by the vector format buttons: `labels` uses plain labelled vectors where elements are displayed separately; `ranges` provides a more efficient display, grouping contiguous elements with the same value. This is done by using an ellipsis to join the first and last elements of the range. All elements and/or ranges are prefixed by the corresponding ternary value (`+`, `-`, or `?`).

F.3.5. Production View. This view provides a list of the names of productions in the selected category. It is empty when no category is selected. The definition of the selected production is shown in the editing view, or a production template if no production is selected.

The *operate* menu of this view has four options: `move`, `rename`, and `remove`. Of these only `move` has not been described as of yet. It allows the user to change the category of a production. It prompts for a new category, and moves the selected production to it. This is a simpler way to change the category of a single production than editing the category structure. Also present are options to spawn other browsers:

`spawn entry`: opens a browser on all lexical entries that include the selected production as a category;

`spawn`: opens a browser on the selected production.

Production definitions consist of six labelled fields:

Name: the name/label of the production. Production names are unique within a parser. When a production is accepted a check is made first to see if there is an existing production with the same name but in a different category. If this is the case, the user is asked if the existing production should be removed. If they answer negatively, the existing production is left and the new production is not added to the grammar. If there is an existing production with the same name in the same category, it is simply replaced.

Comment: a comment attached to the production.

Type: the lexical type of the production. It must be one of:

- **N:** non-lexical. These productions are not lexically constrained, they are considered whenever there is no (semi)lexical production that can be used. Non-lexical productions are searched in a specified order, determined as described in § F.3.3. The search stops when a usable production is found. When a non-lexical production is used, it does not cause the current word to be consumed.
- **S:** semi-lexical. These are lexically constrained, meaning that they are specified in the definition of lexical entries, but do not consume input. The type field of semi-lexical productions can also specify the lexical productions that they subcategorize. This is done by following the type specifier, `S`, by the names of lexical productions, separated by whitespace.
- **L:** lexical. Lexical productions are also lexically constrained, but they do cause input to be consumed.
- **I:** init-final. This is a special case of lexical. Init-final productions define the set of final states of the parser which signifies the completion of a successful parse. The change vector of the last init-final production to be defined is

used to set the initial state of the parser. In practice there are very few init-final productions in a grammar, and they should all have the same change vector. Because of this, any of them can be used to initialize the parser state.

Cond: the condition vector. When a production is under consideration for use, its condition vector is matched against the current parser state. If this match fails, the production can not be used, if it passes other tests are done to determine if the production can be used.

Change: the change vector. This vector is used to update the parser state whenever the production is successfully used.

Actions: actions to be performed when a production is used. If all actions are executed successfully, then the parse state is changed according to the change vector of the production. If any action fails, the effects of any previous actions for the current use of the production are undone, and the production is not used. Backtracking will then cause another production to be selected.

Any type of whitespace can be used to separate alphabetic tokens, and separation is not required between alphabetic and non-alphabetic tokens, although it is used when actions are displayed.

The values of these fields are enclosed in angle brackets. This has two purposes:

(1) It allows the entire field contents to be selected by clicking just inside either angle bracket, and

(2) it allows the accepting mechanism to easily extract the field values.

F.3.6. Editing View. The editing view is a standard ST80r4 `TextView`, supporting all editing functions this implies (Systems, 1990).

Comments for boundaries and features are simply straight text: anything in the editing view is taken as the comment when `accept` is selected from the *operate* menu. The formats of the editing view text for the category structure, non-lexical modes, and production definitions have been described previously in § F.3.3 and § F.3.5.

F.3.6.1. *Editing Ternary Vectors.* Ternary vectors can be edited with or without ranges, as described in § F.3.4. Also, feature macros can be used in either mode by using the name of the macro prefixed by '#'. When the production is accepted, any macros in the vectors are expanded. Vector element are processed from left to right, so feature values later in the vector override those appearing earlier. As an example, say macro `M` is defined as `+A -B`. The vector `#M -A` results in `-A -B`. This is most commonly used for overriding feature values in the default macros (see § F.3.2.1). When condition and change vectors are displayed, only the differences from the corresponding default macro is used. This is done since the default macro will be used if the production is accepted. Due to this there will sometimes be elements with '?' as their value.

Using ranges is the most useful mode, and is recommended.

F.4. Lexicon Browser

The lexicon browser is slightly more complex that the grammar browser. It provides access to the parts of parsers that are directly related to the lexicon:

- semantic features
- semantic feature macros
- semantic relations
- morphosyntactic properties
- morphosyntactic paradigms

- entry categories
- lexical entries
- lexical category macros

A lexicon browser window is shown in Figure F.4. It is divided into seven views:

Semantic feature view: top left

Semantic relation view: below the semantic feature view

Property view: right of the semantic feature view

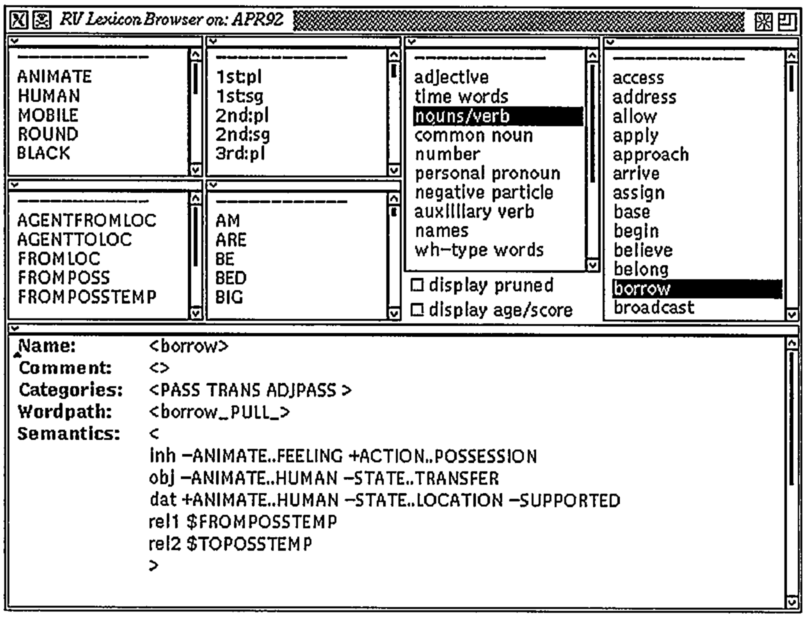
Paradigm view: below the property view

Entry category view: right of the property and paradigm views

Entry format buttons: below the category view

Entry view: top right

Editing view: across the bottom



```
Name:        <borrow>
Comment:     <>
Categories:  <PASS TRANS ADJPASS >
Wordpath:    <borrow_PULL_>
Semantics:   <
             inh -ANIMATE..FEELING +ACTION..POSSESSION
             obj -ANIMATE..HUMAN -STATE..TRANSFER
             dat +ANIMATE..HUMAN -STATE..LOCATION -SUPPORTED
             rel1 $FROMPOSSTEMP
             rel2 $TOPOSSTEMP
             >
```

FIGURE F.4. Lexicon Browser Window

```
inh  -ANIMATE..FEELING +ACTION..TRANSFER +LOCATION..DEST
obj  ^subj
dat  -STATE..TRANSFER +LOCATION
```

FIGURE F.5. Relation: AGENTTOLOC

F.4.1. Semantic Feature View. This view operates the same as the ordering feature view of the grammar browser (see § F.3.2), except that it deals with semantic features rather than ordering features. There is also a corresponding feature macro editor which is the same as that for ordering features (see § F.3.2.1), but does not require the `DEFAULT_COND` and `DEFAULT_CHANGE` macros.

F.4.2. Semantic Relation View. This view provides access to the list of semantic relations, sorted alphabetically. It has an *operate* menu that is the basic add/rename/remove menu which has been described before, with the addition of a `spawn entry` option which opens a browser on all lexical entries that reference the selected relation. When a new relation is added, a very basic template is provided in the editing view. Selecting a relation places its definition in the editing view. The format of relations is simple: a series of role-filler pairs. The roles must be from the list of defined semantic roles (see § F.4.5), while the fillers can be either a binding to a predicate role, or a semantic feature vector. Figure F.5 shows an example relation definition, in particular `AGENTTOLOC` from the parser `APR92`.

F.4.3. Property View. The property view holds a list of all morphosyntactic properties used in paradigms and `lexprop`/`addprop` actions. There are two types of properties:

agreement properties: are used by the `agree` actions for testing agreement. They are the cross product of several agreement sets. Example of agreement sets include `<1st 2nd 3rd>` which contains values for the *person* attribute. Another would be values for *number*: `<sng pl>`. Agreement properties are made up of an element from each of the agreement sets, separated by a colon. For example: `1st:pl`.

non-agreement properties: includes all other properties such as `infinitive`, `past`, `prespart`, etc. These properties are not used in agreement checks.

Non-agreement properties are manipulated using the now familiar add/rename/remove options in the view's *operate* menu.

Agreement properties can not be edited individually. They can only be manipulated by editing the agreement sets. This is done by selecting `edit agreement sets` from the *operate* menu. The sets are then displayed in the editing view.

The *operate* menu has two browser spawning options: `spawn paradigm` and `spawn production`. The former opens a browser on all paradigms that explicitly reference the selected property, and the later opens a browser on all productions who have `addprop` or `lexprop` actions that reference the selected property.

Properties are among the first things defined for a parser, thus once defined they should not change. There is currently no mechanism in place to ripple changes to the properties throughout the rest of the parser.

F.4.4. Paradigm View. This view contains a list of paradigms in the parser, sorted alphabetically. The basic add/rename/remove *operate* menu is present in this view, along with a `spawn entry` option which opens a browser on all lexical entries that have a wordpath containing the selected paradigm.. Selecting a paradigm from the list displays its definition in the editing view, while adding a new paradigm displays a template for one pattern–property set pair. The definition of a paradigm

```
<ying> : <prespart >
<ies>  : <3rd:sg present >
<own>  : <pastpart >
<ew>   : <past >
<y>    : <1st 2nd inf pl present >
```

FIGURE F.6. Example Paradigm Definition

consists of one or more pattern-property set pairs. The pattern is the literal substring to be matched against the input stream, and the property set is the set of morphosyntactic properties that the pattern implies.

Non-agreement properties must be listed literally, while the simplest description of the agreement properties suffices. For example if all agreement properties containing `1st` are desired, then only `1st` need be specified. If only `3rd:pl` is desired, then it must be explicitly specified. Whenever properties are displayed, this simplification is automatically performed, sometimes with surprising results due to the fact that there is not always a single simplest description. Not specifying any agreement properties implies that the set of agreement properties is unconstrained.

Each part of the pair is enclosed in angle brackets for reasons discussed in § F.3.5. When a paradigm definition is generated for display, colons are placed between patterns and their property sets. The purpose of this is merely to visually connect the pairs. As an example, the definition of the paradigm `FLy` is shown in Figure F.6. Note that the pairs in a paradigm are automatically sorted in descending order on the length of the pattern.

F.4.5. Entry Category View. The purpose of lexical entry categories is similar to that of production categories: to organize the lexicon. Like the production category view, this view has a lengthy *operate* menu. The first four options serve the same purpose, see § F.3.3 for details on these. The options are:

add category: adds a new category

rename: renames the selected category

remove: removes the selected category

update: brings the browser up-to-date

definition: displays the definition of the select entry in the editing view. If no entry is selected but an entry category is, then an entry definition template is displayed.

category structure: allows the user to edit the entry category organization. See § F.3.3 for details.

semantic roles: displays the list of semantic roles for the parser. There are two types of roles: inner and outer. Inner roles generally relate to a syntactic entity (subject, object, etc.), while outer roles are intended for relations. In the list a dash is placed between the two types. Inner roles appear first, followed by the dash, and then the outer roles.

spawn trie: opens a browser on the lexical trie. The trie browser is described in §F.5.

edit feature macros: works the same way as the ordering feature macro editor, with the exception of the `DEFAULT` macros. See § F.3.2.1 for details.

edit category macros: works like the feature macro editors, with one major difference: the macros are not defined in terms of ternary vectors, but rather ordered sequences of production names. The macros defined here are used when specifying the categories of lexical entries.

find entry: allows a specific lexical entry to be found quickly. It operates similarly to the `find production` option in the grammar browser (see § F.3.3 for details).

F.4.6. Entry Format Buttons. There are two buttons available for controlling the format of displayed entries:

display pruned: Includes a list of any pruned categories in the entry specification. This information is not editable.

display age/score: Includes age and score information for each active categorization.

F.4.7. Entry View. Similar to the production view in the grammar browser (see § F.3.5), this view presents a list of entry names in the selected category, or is empty if no category is selected. If an entry is selected, its definition is displayed in the editing view. Otherwise a definition template is displayed. The *operate* menu is similar to that of the production view, except that instead of the `spawn entry` option there is a `spawn production` option. This opens a browser on all productions that are named in the selected entry's category list.

Entry definitions consist of five fields:

Name: the name of the entry. The name is used when the entry is referred to in parser traces, debug output, or semantic structures.

Comment: a comment that is associated with the entry.

Categories: the list of lexical and semi-lexical productions that will be examined, to see if they can be used, whenever the entry is encountered in the input stream. The order of the list is significant, the first production that can be used will be.

Wordpath: specifies the path(s) through the trie that end in a leaf referring to the entry. Literal characters in the path appear as literal characters, while a

```
inh   -ANIMATE..FEELING +ACTION..POSSESSION
obj   -ANIMATE..HUMAN -STATE..TRANSFER
dat   +ANIMATE..HUMAN -STATE..LOCATION -SUPPORTED
rel1  $FROMPOSSTEMP
rel2  $TOPOSSTEMP
```

FIGURE F.7. Semantic Information for `borrow`

paradigm is referenced by using its name surrounded by underscore characters. For example, wordpath for the entry "find" is `f_FiND_nd_FINDing_`

Note that the trie is slightly misnamed, as it is not a tree but rather a DAG: several paths can lead to a single entry, and one path can lead to several entries. It is still tree-like in that paths do not merge if they end in nodes having the same pattern/paradigm.

Whitespace other that the space character is used to separate paths, spaces are taken literally as part of a path so that multi-word idioms can be supported.

Semantics: This field defines the semantic properties of the entry. Entries will often have at least some inherent properties that are used to implement selectional restrictions. Values for any or all of the other defined semantic roles can be specified. Values for inner roles can only be semantic feature vectors, while those of outer roles must be references to semantic relations. Relations are referred to by using the relation name prefixed by '$'. As an example, the semantic information for the entry `borrow` is shown in Figure F.7.

F.4.8. Editing View. This view has the same capabilities as the editing view of the grammar browser (§ F.3.6).

F.5. Lexical Trie Browser

The lexical trie browser consists of a window containing a graphical display of the lexical trie, and an editing pane. An example trie browser is shown in Figure F.8. Paradigms in the trie, whether in a paradigm mode or a paradigm set node, are displayed in boldface; lexical entry names are in italics, and literal strings are in the default typeface.

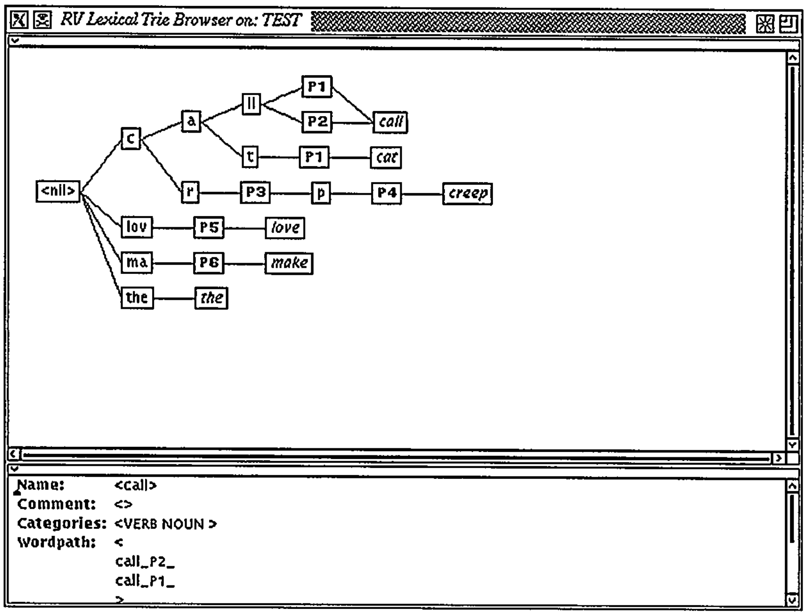
When a node in the trie is clicked on with the select button, the definition of that node is displayed in the editing pane. If the node is a leaf (i.e. a lexical entry) the definition of the lexical entry is displayed, and can be edited. If the node is internal, the node pattern is displayed. Each node has an operate button menu: currently the only options are to open a standard inspector on the node and to remove the node and its descendants from the trie. **NOTE** that this will **NOT** remove anything from the lexicon... only from the trie. The operate menu for the area between nodes contains a single option: update. This causes the trie to be redisplayed, and is useful if a word was learned or the trie otherwise modified.

A word can be entered in the editing pane, selected, and looked up in the trie using the operate menu of the editing pane. The path(es) from the trie root to the appropriate leaves will be hilighted by using wider lines for the node borders and connecting edges.

F.6. Debugger

The RV Debugger provides facilities for testing parsers through the following capabilities:

- specifying input to the parser
- controlling parser execution



```
Name:       <call>
Comment:    <>
Categories: <VERB NOUN >
Wordpath:   <
            call_P2_
            call_P1_
            >
```

FIGURE F.8. Lexical Trie Browser

- controlling the information displayed during parser execution
- inspecting the contents of boundary registers
- inspecting semantic structures

The debugger window is shown in Figure F.9. It consists of five main areas:

Input view: across the top

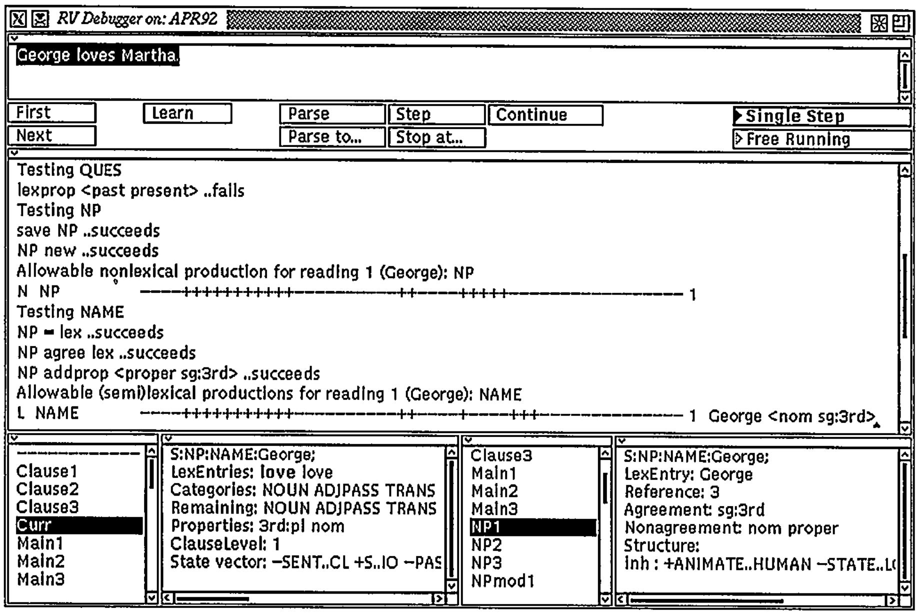
Control buttons: below the input view

Trace view: below the control buttons

Boundary register inspector: the two views in the bottom left

Grammatical Role inspector: the two views in the bottom right

F.6.1. Input View. Input to the parser is entered in this text editor view. The text entered here should consist of one or more sentences, each terminated by a `<return>`. Sentences beginning with '#' are comments and ignored.



```
George loves Martha.
```

```
Testing QUES
lexprop <past present> ..fails
Testing NP
save NP ..succeeds
NP new ..succeeds
Allowable nonlexical production for reading 1 (George): NP
N  NP        ----+++++++++++-------------++-----+++++-------------------- 1
Testing NAME
NP = lex ..succeeds
NP agree lex ..succeeds
NP addprop <proper sg:3rd> ..succeeds
Allowable (semi)lexical productions for reading 1 (George): NAME
L  NAME      ----+++++++++++-------------++-----+-----+++--------------- 1  George <nom sg:3rd>
```

```
------------   S:NP:NAME:George;                    Clause3   S:NP:NAME:George;
Clause1        LexEntries: love love                Main1     LexEntry: George
Clause2        Categories: NOUN ADJPASS TRANS       Main2     Reference: 3
Clause3        Remaining: NOUN ADJPASS TRANS        Main3     Agreement: sg:3rd
Curr           Properties: 3rd:pl nom               NP1       Nonagreement: nom proper
Main1          ClauseLevel: 1                       NP2       Structure:
Main2          State vector: -SENT..CL +S..IO -PAS  NP3       Inh : +ANIMATE..HUMAN -STATE..L
Main3                                               NPmod1
```

FIGURE F.9. Debugger Window

The *operate* menu contains some of the standard text editing functions as well as other special purpose functions:

load file...: presents the user with a file chooser listing filenames matching the pattern: "`sentin.*`". These files are expected to conform to the rules given for the input view. Selecting a file and clicking the `Load` button causes the selected file to be loaded into the input view.

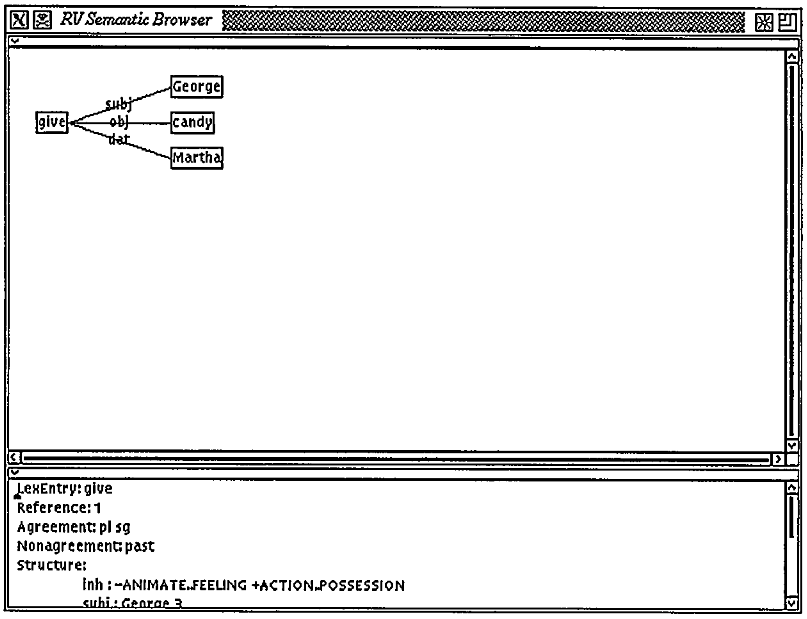
start server: puts the debugger into server mode. A socket is set up for clients to connect to. The port number of the server is displayed in the trace view. The server then waits for a client to connect. If no client has connected in two minutes, the socket is closed and server mode terminated. A message to this effect is displayed.

If a client connects within the time limit, this is also indicated. This causes the debugger to wait for a sentence from the client, which is displayed in the input view, selected, and given to the parser to parse. Parser control and the inspectors are still available to the user as usual. The server is terminated when the user selects the `stop server` option from the input view *operate* menu or an empty string is received from the client.

When the server is operating, results of sentence parses are sent to the client as well as being displayed in the trace view.

stop server: causes the server mode to be terminated, and the socket closed.

browse semantics: When a parse has completed successfully and there are semantic interpretations resulting a submenu is displayed listing the different interpretations. The user can choose one of them to have a graphical browser opened to examine it. An example of the semantic browser is shown in Figure F.10.



```
LexEntry: give
Reference: 1
Agreement: pl sg
Nonagreement: past
Structure:
          inh : -ANIMATE.FEELING +ACTION.POSSESSION
          subj : George 3
```

FIGURE F.10. Semantic Browser

reread input: causes the parser to read from the input stream. This is to be used when the parser fails to recognise a lexical entry from the input stream. The idea is that the user would then fix the problem using the lexicon browser and ask the parser to reread it.

clear: empties the input view.

F.6.2. Control Buttons. These buttons are all located between the input and trace views. The buttons are currently grouped into four sets, with functions as follows:

Sentence Scanning: These two buttons provide a convenient method of moving through the input view, sentence by sentence. This is especially useful after loading a sentence file. Sentences beginning with '`#`' are skipped. The two buttons are:

`First`: select the first sentence in the input view.

`Next`: select the next sentence. If the last sentence was selected, then no selection will be made and the input view will flash. If `Next` is clicked again the first sentence will be selected.

Learning: There is only one button in this section:

`Learn`: This button invokes the vocabulary learner, passing it the selected "word".

Parser Control: There are currently five buttons that control the operation of the parser:

`Parse`: Start the parser, passing it the selection in the input view. Confirms the user's intention to restart the parser if it already running.

`Parse to`: Present a dialog listing all productions from which the user can then choose one. The parser is started and will run until the chosen production is used successfully or it terminates normally. Clicking the cancel button aborts the `Parse to` operation.

`Step`: Have the parser execute one cycle. This means that the parser will backtrack (usually using `Curr`) and try to find a usable production. If one is found, it will be used, otherwise the parser will have to backtrack further in the next cycle. In either case, control will be returned to the user.

`Stop At`: Similar to the `Parse to` button, but it continues the parse from the current state rather than from the initial state. The user regains control when the chosen production is used or the parse terminates normally.

`Continue`: Place the parser in free-running mode and complete the parse without stopping.

To indicate that the parser is running, the cursor changes into the execute form. When the cursor is in its normal form the user has control.

Currently there is no way to stop the parser while it is running, short of using `<Ctrl-C>`. A future version of the debugger will have a `Stop` button.

Parser Mode: The parser can operate in one of two modes, selected by the corresponding buttons:

`Single Step`: the parser stops after each cycle and returns control to the user.

`Free Running`: the parser will continue to execute until the parse has been completed. A parse completes when the *resume* stack is empty (i.e. all alternative interpretations have been explored).

F.6.3. Trace View. Most output from the debugger and parser is displayed in the trace view. The level of detail of this information can be controlled by a series of dialog boxes that are accessible through the view's *operate* menu. This menu contains the following options:

`parser...`: Opens a dialog of parser related settings:

Production application: : Displays information when a production is successfully used. This information includes:

- the type of the production
- the name of the production
- the parser state vector after the production has been used
- the embedding level (1–3) after the production has been used
- if the production was lexical, the name of the entry, and the resultant set of morphosyntactic properties

If this switch is turned "off", the three following switches are locked "off" as well.

Allowable productions: Displays the production that are under consideration during this cycle, separated into (semi)lexical and non-lexical. If there are any (semi)lexical productions, then non-lexical productions are completely ignored, and thus not reported.

Reasons productions disallowed: Displays the reason that each possible production is not usable at the current time. This causes volumous output to be generated, so it is not recommended except during serious debugging.

Actions: Each action of the production being used is displayed as it is executed, along with it's return status (succeeds/fails).

Backtracking: Displays when the parser backtracks using boundary registers other than `Curr`. If this switch is "off", the following switch is locked "off".

Backtracking with Curr: Displays when the parser backtracks using the `Curr` boundary register.

Parse trace at each step: Displays the linear trace after each parser cycle.

Boundaries in trace: Includes boundary placement in the parse trace. This involves having the boundary register name and level enclosed in square brackets and inserted into the trace at the point at which it is saved.

Semantic structure at each step: Displays the semantic structure rooted in Main1 after each parser cycle.

Label vectors: Causes state vectors to be displayed using ordering feature labels and ranges rather than the default, which is to use the raw `+/-` notation.

`scanner...`: Opens a dialog of scanner related settings:

Lexical lookup: Display the search through the lexical trie. This indicates what is being looked up in which node, as well as what entry was found, and with what properties. For paradigms, the pattern that matched is also indicated.

Indicate unrecognisable input: Indicate when the parser can not find an entry through the trie, and up to ten characters of input that caused the problem.

`vocab learner...`: Opens a dialog of vocabulary learner related settings:

Enable integrated learning: Have the vocabulary learner invoked whenever there is unrecognized input.

Trace output: Output trace information regarding what the vocabulary learner is doing.

Initial score: (1–100) The score that new categorizations are given.

Immortal score: (1–10000) The score at which a categorization becomes persistent.

Score increment: (1–10) The amount by which a categorization's score is increased when it is used in a successful parse.

Death age: (1–100) The age at which a categorization's score begins to decrease.

Score decay: (1–10) The amount by which a categorization's score decreases, after its age exceeds the death age.

`clear`: Clear the trace view.

In each of the above dialogs, the settings can be adjusted as required. When they are satisfactory, the `Done` button is clicked to dismiss the dialog. Other pieces of information that are always displayed include the initial parser state, notification of a successful parse, and the results of the parse once it has completed. This last item consists of the linear trace and resulting semantic structure for each possible interpretation. If no interpretations were found, then the input was ungrammatical, and this is also indicated.

F.6.4. Boundary Register Inspector. The boundary register inspector consists of two views. The left-most displays a list of all boundary registers: `Curr`, `Word`, and the cross product of `Main` along with the user defined boundaries and the embedding levels. The list is in alphabetical order.

When a boundary register is selected from the list, its contents are displayed in the righthand view. The information displayed includes:

- the linear parse trace
- the list of lexical entries, with the current one in boldface
- the list of (semi)lexical production names associated with the current entry
- the set of morphosyntactic properties associated with the current entry
- the embedding level
- the parser state, displayed using ordering feature labels and ranges.

F.6.4.1. *Comparing productions.* The *operate* menu of the boundary register contents view currently has a single option: `compare productions`. Selecting it opens a dialog which presents a list of production names and a text field labelled: *Difference*. When a production is selected in the list, the difference between its condition vector and the parser state saved in the selected boundary register is displayed in the text field[1]. This difference signifies the changes that would have to be made to the state in order for the production's condition vector to match it.

F.6.5. Grammatical Role Inspector. The grammatical role inspector operates in a similar fashion to the boundary register inspector. A grammatical role is selected, and its contents and the contents of the associated semantic entry are displayed. The following information is displayed:

- the current linear parser trace
- the lexical entry associated with the semantic entry
- the grammatical role's index into the reference queue
- the agreement and non-agreement properties associated with the grammatical role

[1]Note that this is display only!

- the structure of the semantic entry, its semantic roles and their fillers. Role fillers that are themselves semantic entries, only display as the associated lexical entry and the reference index.

F.6.5.1. *Reference Inspector.* The *operate* menu of the grammatical role contents view also has only one option currently: `inspect references`. This invokes an inspector that is fashioned after the standard `SequencableCollectionInspector`: on the left is a list of reference indexes, and on the right is a text view that displays the contents of the selected reference. The information displayed is much like that displayed in the grammatical role contents view, with the exclusion of the parse trace and reference index, and the addition of the timestamp associated with the reference.

Using the reference inspector, one can easily inspect the semantic structure being built. Note that while the contents of boundary registers and grammatical roles are automatically updated whenever the parser stops, the reference inspector is not updated, but does provide access to the up-to-date reference contents, the display must be manually refreshed.

Also note that while most popups in the system are blocking, the reference inspector is not.