COLORADO TECHNICAL UNIVERSITY

ENABLING A LEGACY MORPHOLOGICAL PARSER TO USE DATR-BASED LEXICONS

VOLUME 1 of 2

A DISSERTATION SUBMITTED TO
THE GRADUATE COUNCIL
IN PARTIAL FULFILLMENT OF
THE REQUIREMENTS FOR THE DEGREE OF
DOCTOR OF COMPUTER SCIENCE

DEPARTMENT OF COMPUTER SCIENCE

BY

MICHAEL A. COLBURN

M.C.I.S., University of Denver, Denver, CO, 1991
M.A., University of Texas, Arlington, TX, 1981
B.A., Rockmont College, Denver, CO, 1975

DENVER, COLORADO
DECEMBER 1999

ENABLING A LEGACY MORPHOLOGICAL PARSER

TO USE DATR-BASED LEXICONS

BY

MICHAEL A. COLBURN

THE DISSERTATION IS APPROVED

_____

Dr. Carol Keene, Committee Chair

_____

Dr. H. Andrew Black

_____

Dr. Jay Rothman

_____

Date Approved

ABSTRACT

This research is an investigation of the feasibility and desirability of using DATR with AMPLE, a legacy morphology exploration tool developed by the Summer Institute of Linguistics. The research demonstrates the feasibility of using DATR with AMPLE by showing how DATR can be used to encode the lexical information required by AMPLE and by presenting an interface between AMPLE and DATR-based lexicons that does not require modification of AMPLE itself. The desirability of using DATR with AMPLE is shown by demonstrating that there are generalizations that cannot be captured using AMPLE's lexical knowledge representation language (LKRL) that can be captured in DATR, thereby reducing redundancy of lexical information.

The research demonstrated the truth of the four hypotheses. In order to prove these hypotheses, the author analyzed the morphophonology and morphotactics of a Papuan language, Ogea, and built a comprehensive AMPLE unified database lexicon that supports the parsing of every example found in the author's write-up of Ogea. Next, the author developed a DATR-based version of the lexicon, as well as an interface to generate the AMPLE lexicon from DATR. It was found that linguistic generalizations not possible in AMPLE's LKRL could be made in DATR. Through these generalizations, redundancy in the Ogea AMPLE lexicon was reduced by 64% in the DATR version of the same lexicon. The validity of the interface from the DATR lexicon to the AMPLE lexicon was verified by programmatically. A portion of an AMPLE root database file for a second language, Yalálag Zapotec, was also analyzed. It was demonstrated that for that language also, generalizations not possible to make with the AMPLE LKRL could be made in DATR. Redundancy in the Yalálag Zapotec subset AMPLE lexicon was reduced by nearly 58% in the DATR version.

ACKNOWLEDGEMENTS

We all stand on the shoulders of those who went before us. In that sense, the list of acknowledgements would be quite long. However, this research was made possible by the following people: my employer, who allowed my work schedule to be flexible; Kristin and Gabriel, who put up with a pre-occupied father for several years; my wife, Lisa, who helped with proof-reading and offered moral support; Kelebai Iriwai, who over the years has tirelessly helped me to understand his language, Ogea; members of the Papua New Guinea branch of the Summer Insititute of Linguistics who shared their insights into Papuan linguistics and phonology to help me in my analysis of Ogea, Cindy Farr, Dotty James, Sjakk and Jacqueline van Kleef, Denise Potts, Ger Reesink, and others; Dr. Donald Burquest, who first encouraged me to pursue doctoral work; the members of my committee for their insightful comments and encouragement, especially Dr. H. Andrew Black, due to his expertise in linguistics and AMPLE.

Of course, this author takes full responsibility for the content presented here, including decisions made regarding the analysis of Ogea morphophonemics and morphotactics.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF ABBREVIATIONS

1	First person
2	Second person
3	Third person
B	Benefactive
cat	Category
cert	Certainty
C	Consonants
Contra	Contrafactive
d	Dual
Hab	Habitual
imp	Imperative
int	Intensive
inf	Infinitive
Nom	Nominative
O	Object
p	Plural (note: a number after the p indicates the 1st or 2nd type of plural)
P	Possessive
perf	Perfective
pred	Predictive
s	Singular
S	Subject
Sj	Subjunctive
SR	Switch Referent
syn	Syntactic
Tf	Future Tense
Tip	Immediate Past Tense
Tp	Present Tense
Trp	Remote Past Tense
TO	Temporal overlap
TS	Temporal Succession
vAux	Verbal Auxilliary
vAuxR	Verbal Auxilliary Root
V	Vowels

LIST OF ACRONYMS

AVM	Attribute-Value Matrix
BNF	Backus-Naur Form
CARLA	Computer-Aided Related Language Adaptation
DAG	Directed Acyclic Graph
FSM	Finite-State Machine
FT	Final Test
GPSG	Generalized Phrase Structure Grammar
HPSG	Head-Driver Phrase Structure Grammar
IDE	Integrated Development Environment
LFG	Lexical Functional Grammar
LTAG	Lexicalized Tree Adjoining Grammar
LKRL	Lexical Knowledge Representation Language
MCC	Morpheme Co-Occurrence Constraint
MEC	Morpheme Environment Constraint
MT	Machine Translation
NLP	Natural Language Processing
NP	Noun Phrase
PSG	Phrase Structure Grammar
SEC	String Environment Constraint
SIL	Summer Institute of Linguistics
ST	Successor Test
TIC	Traffic Information Collator
VP	Verb Phrase

CHAPTER 1

INTRODUCTION

Consider a computerized text-to-speech system that encounters the word *boathouse*[1]. Armed only with the rule that the two-letter sequence *th* is a voiceless inter-dental fricative sound, the system would incorrectly pronounce *boathouse* as 'bow-thous', with 'thous' rhyming with the first part of the word *thousand*. But further armed with the knowledge that *boathouse* is actually composed of two components, *boat* + *house*, and that in such cases the *t* and the *h* are pronounced separately, rather than together, the system could correctly pronounce *boathouse*.

Consider searching the Internet for web pages containing information about turtles. What if in response to the user's request, the search engine only returned pages that literally contained the plural form *turtles*, while ignoring pages that contain the singular form, *turtle*? Fortunately this is not the case. Query languages and search engines include components that are programmed to identify and strip off plural endings in order to normalize indexing and search terms, so pages with both the singular and plural forms of words will be returned independently of the actual form the user enters.

Consider a natural language processing (NLP) system designed to "understand" newspaper articles. Imagine it attempting to determine who hit whom in the sentence *The boy the boys were hitting disappeared*. Since both *boy* and *boys* are animate objects semantically capable of performing the action of hitting, semantics provides no clue in determining who hit whom in this sentence. However, by recognizing that *boys* in *the boys* is composed of two components, *boy* + *s*, indicating a plural noun, and by knowing that *were* in

were hitting requires a plural noun as subject, the NLP system could correctly identify *the boys* as the subject noun phrase of *were hitting* and avoid incorrectly identifying *the boy* as the subject noun phrase of *were hitting*.

What do these examples have in common? Though simplistic, they illustrate the usefulness of morphological parsers—software that can identify the individual components (morphemes) that make up words.

In order to be useful, morphological parsers require access to information about the vocabulary of a natural language, typically contained in lexicons. Machine-readable lexicons play a crucial role in computational approaches to morphological parsing in particular and NLP in general. Many modern linguistic theories make use of the unification of grammatical rules with lexical entries that contain complex features. This has led to grammatical rules that are simplified at the expense of increased complexity in the lexicon itself. To better manage this complexity, researchers have investigated the use of inheritance-based lexical knowledge representation languages (LKRLs). Inheritance is viewed as a means to improve the capturing of linguistic generalizations in the lexicon and to increase the maintainability of lexicons by eliminating or minimizing redundancy of information across lexical entries.

One outcome of such research is an LKRL called DATR [29, 30, 32]. Although DATR is a proven language for representing lexical information for unification and feature-based approaches to grammar, there are existing morphological parsers that were developed based on previous linguistic traditions or that use ad hoc approaches. Such legacy parsers could potentially benefit from the use of an LKRL like DATR, if DATR is indeed capable of expressing the lexical information required by such parsers.

---

[1] This example comes from [68:7]

This author's research investigates the feasibility and desirability of using DATR with AMPLE, a legacy morphology exploration tool developed by the Summer Institute of Linguistics (SIL) [6, 9, 70], the world's largest linguistic organization. This author's research demonstrates the feasibility of using DATR with AMPLE by showing how DATR can be used to encode the lexical information required by AMPLE and by presenting an interface between AMPLE and DATR-based lexicons that does not require modification of AMPLE itself. The desirability of using DATR with AMPLE is shown by demonstrating that there are generalizations that cannot be captured using AMPLE's LKRL that can be captured in DATR, thereby reducing redundancy of lexical information.

In the remainder of the introduction, a background for the research problem is presented, followed by a statement of the research problem itself and the hypotheses upon which the research is based.

1.1 Problem Background

This section provides a background to the research problem by focusing on some basic questions: What is morphology and why is morphology important? What are morphological parsers and why are morphological parsers important? What is the purpose of lexicons, and what are some problems with lexicons? What is AMPLE and how does it work? What is DATR and how might it benefit AMPLE? Some knowledge of these topics is required to understand the research problem and the research hypotheses. Relevant literature concerning these topics is presented in Chapter Two, and extended examples of AMPLE and DATR are provided in the appendices, using the Ogea language of Papua New Guinea and Yalálag Zapotec, a language of Mexico.

1.1.1 What is Morphology?

Morphology is a sub-field of linguistics that studies word structure and word formation. Although it is a sub-field, it is none-the-less a crucial one. Spencer and Zwicky [67:1] assert that conceptually morphology is at the center of linguistics because "…morphology is the study of word structure, and words are at the interface between phonology, syntax, and semantics".

One way to view words is that they are concatenated strings of morphemes. A morpheme is the minimal meaning bearing unit of human language. For example, *laughing* is composed of two morphemes, *laugh* + *-ing*. Morphemes are classified as roots or affixes. Roots bear the main meaning of a word, e.g., *laugh* in *laughing*. Affixes attach to roots and fall into three major types: prefixes (e.g., *un-* in *undo),* suffixes (e.g., *-ing* in *laughing),* and infixes[2]. Affixes can also be categorized as inflectional or derivational. Inflectional affixes encode grammatical meaning, e.g., person, number, tense. Derivational affixes combine with roots to form stems. They often change the grammatical class of the word. For example, adding *–al* to the noun *nation* results in the adjective *national*, and adding *–ize* to *national* changes it to *nationalize*, a verb.

Two general areas of morphology are morphophonemics[3] and morphotactics. Information about a language's morphophonemics and morphotactics provides important clues in the computational identification of morphemes.

Morphophonemics is the study of phonologically conditioned variants of morphemes (allomorphs). For example, three allomorphs for the English plural suffix can be seen in

---

[2] Infixes are discussed in Chapter Two.

examples (1), (2), and (3). The morpheme –*s* is realized as {-s} following voiceless consonants (e.g., *t* in *bet*), as {-z} following voiced consonants (e.g., *d* in *bed*), and as {- z} following another *s* (e.g., following *place*[4]).

(1) bets {-s}
(2) beds {-z}
(3) places {- z}

Morphotactics is the study of morpheme co-occurrence restrictions. These restrictions include the order of occurrence, e.g., **nation-tion-al-ize*[5] vs. *nation-al-iz-ation*, and what morphemes can combine with other morphemes, e.g. –*al* combines with a morpheme that is a noun, and not with a verb.

Morphological analysis of words is important as an aid in identifying syntactic structures and roles (syntactic parsing) as was seen above. The meaning and function of the individual morphemes that make up a word contribute to the meaning of the word as a whole, and to the sentence in which the word occurs. Thus, morphological analysis plays an important role in determining the meaning of words and sentences.

1.1.2 What are Morphological Parsers?

A morphological parser is software that accepts as input a natural language string and returns one or more of the following types of information about a word: the word structure (morpheme identification), part of speech (grammatical class), the meaning of individual morphemes in the word, the meaning of the word itself, and possibly other information.

---

[3] According to Crystal [24], the preferred term in Europe is *morphophonology*. This dissertation follows the American tradition of using the term *morphophonemics*, even when discussing linguistic literature deriving from Europe.

[4] Note that phonologically the word *place* ends with an *s* though it is orthographically represented as *c*. The *e* following the *c* is silent. When the plural suffix –s attaches to a root ending in -s, a vowel is inserted ( ), and the s becomes voiced, yielding {- z }.

[5] Traditionally in linguistics, unnatural constructions are marked with an asterisk.

Morphological parsers are important for a number of reasons. They are useful to linguists for testing linguistic theories and exploring the morphology of a specific language. In NLP systems, morphological parsers are often a component used for information retrieval, speech recognition, text-to-speech processing, and generating input to syntactic parsers.

1.1.3 What is the Purpose of Lexicons?

Lexicons contain information about the vocabulary of a language. This information can range from a simple word list to a list of roots, affixes, and lexical rules, along with phonological, morphological, syntactic, and semantic information for each entry. It is important to realize that it is difficult, if not impossible, to list every possible surface form of each word of a natural language. There must be a means to identify and analyze variations of a word in addition to the form(s) contained in a lexicon. This problem further motivates the need for morphological parsers.

Many modern linguistic theories make use of the notion of unification. The unification operation combines two descriptions into a single description. By using unification and lexical entries that contain complex features, it is possible to develop relatively simple grammatical rules. However, this relative simplicity is at the expense of increased complexity in the lexicon itself. The trend in linguistic theories of moving increasing information into the lexicon is called lexicalization. Lexicalization has highlighted the importance of which LKRL is selected to encode lexical entries.

The choice of LKRL impacts the ability to capture linguistic generalizations and to deal with redundancy in the lexicon. It is generally desirable that entries in lexicons be encoded in such a way as to capture linguistic generalizations and to eliminate, or at least minimize, redundancy. The ability to capture linguistic generalizations and to minimize

redundancy is desirable from both a theoretical and practical perspective. There has been a traditional belief in the field of linguistics that analyses that capture abstractions or generalizations are preferred over analyses that do not. Generalizations are useful for establishing universals of natural languages [24:165]. On the practical side, a lexicon that contains redundant information is one that requires more labor to develop and maintain than one that either minimizes or eliminates redundancy. A change to a redundant piece of information requires the change to be made in many places rather than in just one place. Therefore, for practical reasons it is desirable to have a lexicon that captures linguistic generalizations and minimizes redundancy of data. The degree to which a lexicon can do this depends greatly on the LKRL chosen.

One mechanism for capturing generalizations and minimizing redundancy that has received much attention by the research community is inheritance. In this approach, lexical entries form a hierarchy. Entries may be the children of other entries and inherit the properties of their parent. Consider the case of transitive verbs. Transitive verbs take two arguments (a subject noun phrase and an object noun phrase). Lexical entries for individual transitive verbs should not carry such information. If they did, the information would be redundantly carried in the lexicon. Instead, an abstract lexical entry for transitive verbs could carry the information that transitive verbs take two arguments. Individual entries (children) would then indicate that they belong to the transitive verb class (their parent). Through inheritance, all lexical entries that are children of the abstract entry for transitive verbs inherit the information contained at the parent level, the abstract lexical entry for transitive verbs.

Thus far, this introduction has provided background information concerning the nature and role of morphology, morphological parsers, and lexicons. The introduction now turns to a specific morphological parser, AMPLE, and a specific LKRL, DATR, and then discusses how AMPLE could potentially benefit from the use of DATR.

1.1.4 What is AMPLE? How Does it Work?

As stated above, AMPLE is a morphological exploration tool that was developed by the SIL. It has been used for research with literally hundreds of languages throughout the Americas and the rest of the world. Though primarily used within the SIL, AMPLE is known outside the SIL community. Historically, AMPLE has its roots in language specific parsers, starting in 1979 [70:2]. AMPLE version 1.0 was released in 1988 as a language independent parser. Although over 10 years old, AMPLE has been maintained and enhanced over the years. Recently, version 3.1 (July, 1998) added support for digraphic characters[6], and version 3.2 (October, 1998) added the ability to state reduplication patterns [58].

AMPLE uses files that contain information about a language's roots and affixes. This information is used for tests to identify the morphemes that make up a word. Without tests, AMPLE would return potentially incorrect combinations of morphemes. AMPLE uses two major categories of constraints to test the validity of a parse—morphophonemic constraints and morphotactic constraints.

There are three major categories of morphophonemic constraints available in AMPLE. String environments are used to constrain the occurrence of allomorphs based on strings and string classes. Morpheme environments are used to constrain the occurrence of

---

[6] That is, phonemic segments represented orthographically by two or more characters.

allomorphs based on morphemes and morpheme classes. Lastly, AMPLE allows users of the parser to define their own tests (user-written tests).

There are four major types of morphotactic constraints available for tests in AMPLE. Orderclass constraints are rules regarding the order in which morpheme classes may appear in a word. Morpheme co-occurrence constraints are rules about whether a specific morpheme or morpheme class can or cannot appear if another morpheme or morpheme class is present or absent in a word. Category mapping constraints are rules based on the notion of from-categories and to-categories (à la categorial grammar). Each affix is assigned a from-category and to-category. A parse is rejected if the from-category of an affix in question does not match the to-category of some other relevant affix. When there is no other way to provide a constraint for a test, the fourth way to provide a constraint is to define ad-hoc pairs, pairs of morphemes that simply may not occur together.

AMPLE's LKRL uses record and field markers to encode lexical entries. These markers define a database schema for encoding lexical information. The AMPLE LKRL allows entries to be assigned to classes, but lacks an inheritance mechanism, which makes capturing generalizations difficult and increases redundancy in the lexicon.

1.1.5 What is DATR? How Might it Benefit AMPLE?

DATR is currently the most widely used LKRL in the computational linguistics and NLP community. DATR is a declarative language that was designed to support the kind of lexical entries used by unification approaches to linguistics. However, Evans and Gazdar [32] claim that DATR is linguistic theory neutral and in fact can encode polytheoretic lexicons, with specific theory selection based on parameters used at run-time. DATR can capture both generalizations and sub-generalizations for phonological, orthographical,

morphological, syntactic, and semantic information. The capture of generalizations and reduction of redundancy is achieved in DATR through the use of default multiple inheritance.

Default, or non-monotonic, inheritance allows a child to override the properties or features inherited from a parent. This supports the capture of regularities, irregularities, and sub-regularities. Inheritance schemes that allow a child to have only one parent use simple inheritance. DATR, on the other hand, supports multiple inheritance—a child may have more than one parent. This increases the ability to capture generalizations and reduce redundancy. However, DATR does not allow a child to inherit the same properties from more than one parent. This orthogonality requirement avoids some of the pitfalls of multiple inheritance.

The lexical encoding scheme used by AMPLE allows lexical entries to be assigned to classes, but it lacks a mechanism whereby the features of a parent class may be inherited by a child class. Through default multiple inheritance, DATR provides a means not available to AMPLE to capture generalizations and reduce or eliminate redundancy. Theoretically, then, it appears that AMPLE could benefit from the use of DATR as the LKRL to encode lexical information for AMPLE.

1.2 Problem Statement

AMPLE is based on traditional approaches to linguistics that preceded the unification-based approaches for which DATR was developed. Before this author's research was completed, the problem, therefore, was that it was not clear that the kinds of information that AMPLE uses could indeed be encoded in DATR. Also, it was not clear that a means could be provided by which AMPLE can make use of DATR-based lexicons without

modification to AMPLE itself. These issues had not been previously investigated. The basic problem addressed by this research, then, was that it appeared that AMPLE could benefit from use of DATR as an LKRL, but it was not known if it was truly feasible or desirable to do so because it had apparently never been attempted.

1.3 Overview of the Research

1.3.1 Purpose of the Research

The purpose of this research was to answer the following basic research questions: Is it feasible to use DATR as an LKRL for AMPLE? How can DATR be used as an LKRL for AMPLE without modification to AMPLE itself? Is it desirable to use DATR as the LKRL for AMPLE? What are the advantages and disadvantages of using DATR as an LKRL for AMPLE?

1.3.2 Significance

Based on a search of the literature and discussion with members of the SIL computational linguistics community, it is this author's belief that apart from this author's research, no past or present research addresses the issue of interfacing AMPLE with DATR-based lexicons. This research contributes to the body of knowledge of computational morphology and lexicology in a number of ways. First, the research demonstrates that lexical data required for AMPLE-style morphological parsing can be encoded using DATR. DATR was originally developed for the requirements of modern Phrase Structure approaches to grammar that rely on complex features and unification. AMPLE, on the other hand, is rooted in the American Structuralist school of linguistics and other approaches that preceded modern Phrase Structure Grammars. Demonstrating that the data required for AMPLE can be encoded in DATR adds support to the claim made by the developers of DATR that DATR

is theory neutral and powerful enough to encode a wide variety of types of lexical information. Second, the research adds some support to the notion that inheritance-based lexical organization is applicable to all natural languages, not just European ones. In a cautionary note, Daelemans et al. [25:214] state that "…existing work on inheritance lexicons has been almost wholly based on familiar European languages". The natural languages used for the research are non-Indo-European, specifically Ogea, a Papuan language of Papua New Guinea, and Yalálag Zapotec, a language of Mexico. Third, there are other legacy parsers besides AMPLE that could potentially benefit from the use of DATR-based lexicons. Documentation of the issues involved in development of an interface between AMPLE and DATR-based lexicons could provide data for the development of an object-oriented domain framework, where the domain is that of interfacing legacy parsers with DATR-based lexicons. The purpose of such a framework would be to provide a toolkit to build interfaces between DATR-based lexicons and a variety of parsers. Object-oriented frameworks are a relatively new field of study and a current topic of research in computer science and software engineering [40, 56]. Per the framework literature, frameworks should be developed based on a minimum of three actual examples. The interface developed for this research could serve as one of the three examples for future development of a framework.

This research contributes to the practice of computational morphology and lexicology by providing the AMPLE user community with a means to use DATR-based lexicons and an example of how to do so. This is important for a number of reasons. First, AMPLE is widely used within SIL, the world's largest linguistic organization. Of the world's languages for which linguistic descriptions exist, the majority were analyzed by linguists working with SIL. This means that SIL is the major supplier of language data to the worldwide linguistic

community[7], and in this way contributes to both theoretical and descriptive linguistics. The development of lexicons for a language is not a trivial undertaking. By developing and demonstrating an interface between AMPLE and DATR-based lexicons, the effort to develop and maintain AMPLE lexicons will potentially be reduced. Also, the use of DATR-based lexicons could potentially result in lexical analyses that are more linguistically elegant because of DATR's mechanisms for capturing generalizations. Second, because the SIL engages in community development through applied linguistics, by aiding the work of SIL linguists, the research could indirectly benefit applied linguistics projects in many third-world communities.

1.3.3 Research Hypotheses

It is the thesis of this research that it is both feasible and desirable to use DATR as an LKRL for AMPLE. The goal of the research was to verify this thesis and to develop and demonstrate a generalized means by which AMPLE can take advantage of DATR-based lexicons without having to modify the parser itself. With this in mind, four research hypotheses were formulated:

Feasibility

H_1: All types of lexical information expressible in the AMPLE legacy LKRL are also expressible in DATR.

H_2: It is possible to translate AMPLE-oriented DATR lexicons back into AMPLE legacy format for use by AMPLE.

Desirability

H_3: There are generalizations not possible to capture in the AMPLE legacy LKRL that can be captured by use of DATR's inheritance mechanism.

---

[7] This is acknowledged by non-SIL linguists, for example Sproat [68:265].

H_4: Such generalizations can be exploited by 5% or more of lexical entries in the lexicon for a specific language[8].

H_1 and H_2 are concerned with the feasibility of using DATR with AMPLE. Feasibility addresses the status quo in using AMPLE. Feasibility in this context means that any lexical information that can be stated in the AMPLE legacy LKRL is also stateable in DATR, and that it is possible to provide a mechanism for AMPLE to use a DATR-based lexicon. H_3 and H_4 are concerned with the desirability of using DATR with AMPLE. Desirability addresses potential benefits from using DATR as an LKRL for AMPLE lexicons. Feasibility and desirability should both be present to some degree to justify the use of DATR as an LKRL for AMPLE. Neither is sufficient in and of itself. If it is not feasible to use DATR with AMPLE, then it would not be possible to realize the potential benefits of DATR. Although it may be feasible to use DATR, it may be the case that the type of lexical information required by AMPLE does not have characteristics that may be exploited by the inheritance mechanisms of DATR. If this were the case, though it were feasible to use DATR, it would not be desirable to use DATR with AMPLE.

Through the claims of feasibility and desirability, the hypotheses predict three things: that it is possible to state all AMPLE lexical information[9] in DATR, that a mechanism can be provided to allow AMPLE to interface with a DATR-based lexicon, and that natural language lexical information used by AMPLE has characteristics that can be exploited by

---

[8] Dr. H. Andrew Black was consulted as an expert on AMPLE. Dr. Black is a linguist with the Summer Institute of Linguistics, and one of the developers of AMPLE. His opinion was that if even as few as 5% of lexical entries could take advantage of the generalization capability of DATR, this would be a significant improvement in the eyes of those who build and maintain lexicons for AMPLE.

[9] As will been seen in the description of AMPLE, below, in addition to morphemes, morpheme glosses, allomorphs, etymology, etc., lexical information in AMPLE includes ad hoc morpheme pairs, compound root pairs, properties, category pairs, morpheme classes, order classes, string classes, and environmental constraints. AMPLE control file data, input text control data, orthography change data, and tests are excluded since they are not, strictly speaking, lexical information.

DATR inheritance mechanisms. It is through its simple and multiple default inheritance mechanisms that DATR facilitates a reduction of redundancy, an increase in generalizability, and an increase in maintainability. The question is not whether this is true of DATR as an LKRL. These characteristics of DATR are supported by an extensive literature, and therefore are not explicit to the hypotheses above. The real question about the desirability of using DATR with AMPLE is whether the types of lexical information used for morphophonological and morphotactical constraints in AMPLE exhibit features that can be exploited by inheritance mechanisms. If this is true, the rest follows—a reduction in redundancy, an increase in generalizability, and an increase in maintainability.

1.3.4 Proof Criteria

Proof criterion 1. H_1 will be considered true if for each and every field type found in AMPLE dictionary records, equivalent DATR code can be presented.

Proof criterion 2. H_2 will be considered true if it can be shown that a usable AMPLE version of a dictionary database can be generated from a DATR version. By *usable* it is meant that the generated AMPLE file(s) contain the lexical input required to correctly parse the words of a language.

Proof criterion 3. H_3 will be considered true if examples can be shown from AMPLE lexicons for at least two languages where lexical entries contain information that may be generalized and DATR code is presented showing how the generalizations may be captured.

Proof criterion 4. H_4 will be considered true if for some DATR lexicon, $I / A * 100 \geq 5$, where *I* is the count of all inheriting nodes (lexical entries that inherit

information from another node), and *A* is the count of all nodes (all lexical entries).

1.3.5 Methodology

Based on the proof criteria defined above, the methodology to prove the hypothesis will be as follows.

1. Demonstrate that each type of lexical information that can be encoded in AMPLE's legacy LKRL can also be encoded in DATR (proof criterion 1). For each type of lexical information that can be encoded in AMPLE's legacy LKRL:
   - 1.1 Determine the range of variation for that type of information. (This will be based on the Backus-Naur Form (BNF) syntax provided as an appendix to the AMPLE user's manual).
   - 1.2 Find or develop enough examples to cover the range of variation for that type of information.
   - 1.3 Develop a DATR version of each of the examples.
   - 1.4 Present the DATR code, along with output demonstrating that the code works.
2. Demonstrate a mechanism by which it is possible to translate AMPLE-oriented DATR lexicons back into AMPLE legacy format for use by AMPLE (proof criterion 2).
   - 2.1 Develop or select an AMPLE legacy LKRL-based lexicon to use for the demonstration.
   - 2.2 If not already done, develop a DATR-based version of the lexicon.
   - 2.3 Develop the interface mechanism.
   - 2.4 Convert the DATR-based lexicon from step 2.2 into an AMPLE legacy LKRL-based lexicon using the mechanism developed in step 2.3.

2.5 Compare the file formats and contents to verify that the output from step 2.4 (the generated AMPLE lexicon) is identical to the original AMPLE lexicon developed or selected in step 2.1.

2.6 Process the original AMPLE lexicon from step 2.1 using AMPLE. Process the AMPLE lexicon generated from DATR in step 2.4 using AMPLE. Compare the output from both runs. Ensure that they are identical.

3. Demonstrate that DATR's inheritance mechanism can be exploited for AMPLE lexicons (proof criterion 3).

3.1 Analyze existing AMPLE lexicons for examples where inheritance can be exploited.

3.2 Develop a DATR version of the examples.

3.3 Present the DATR code, an analysis, and output demonstrating that the code works.

Using a DATR-based AMPLE lexicon, demonstrate that 5% or more of lexical entries make use of generalizations through DATR's inheritance mechanism (proof criterion 2.2). Do this using the following formula: I / A * 100, where *I* is the count of all inheriting nodes (lexical entries that inherit information from another node), and *A* is the count of all nodes (all lexical entries).

1.4 Summary

This chapter has introduced the author's research, which investigates the feasibility and desirability of using DATR with AMPLE, a legacy morphology exploration tool. This author's research demonstrates the feasibility of using DATR with AMPLE by showing how DATR can be used to encode the lexical information required by AMPLE and by presenting

an interface between AMPLE and DATR-based lexicons that does not require modification of AMPLE itself. The desirability of using DATR with AMPLE is shown by demonstrating that there are generalizations that cannot be captured using AMPLE's LKRL that can be captured in DATR, thereby reducing redundancy of lexical information.

In Chapter 2, a review and analysis of the relevant literature is presented. In Chapter 3, the author's foundational research is presented, followed by a discussion of the research in terms of the four research hypotheses stated above. Chapter 4 is the conclusion, which also presents some topics for further work and research. Five appendices are also presented. Appendix 1 is this author's analysis of the morphophonemics and morphotactics of the Ogea language. Appendix 2 presents a listing of an AMPLE unified database file that supports the parsing of all examples of Ogea presented in Appendix 1. Appendix 3 presents the DATR version of the Ogea lexicon. Appendix 4 presents Yalálag Zapotec AMPLE and DATR lexicons, in both cases using only a sub-set of entries from an AMPLE lexicon supplied by Dr. H. Andrew Black to this author. Appendix 5 provides listings of various Perl scripts developed for the research.

CHAPTER 2

ANALYSIS OF THE LITERATURE

2.1 Introduction

This chapter presents a review and analysis of the literature relevant to this author's research. The review begins with an exposition of the nature of morphology as a sub-field of grammar, and the role of the morphological parser in NLP. Next, a specific morphological parser, AMPLE, will be examined. Following a general discussion of the role of the lexicon in linguistic theory, the use of inheritance will be explored as a mechanism to manage complexity in the lexicon and to better capture linguistic generalizations. Next, attention will be paid to the role of the lexicon in NLP. Following that, a specific LKRL will be examined, namely, DATR, and its ability to support default multiple inheritance will be demonstrated. Finally, research related to this author's will be examined.

2.2 The Morphological Parser

2.2.1 Morphology

In linguistics, morphology[10] is a sub-field of grammar that focuses on the study of the formation or structure of words. Morphology contrasts with syntax, which focuses on the formation or structure of units formed by combining words, e.g., phrases, clauses, and sentences. A central concept in morphology is the notion of the morpheme, which is the minimal meaning bearing unit of language. From this perspective, words are viewed as composed of one or more morphemes. For example, the English word *laughs* consists of two morphemes, the verb root *laugh* and the plural suffix *-s*.

Morphemes can be classified as roots or affixes. Roots are the base form of a word and carry its central meaning. Roots that do not combine with other morphemes are free morphemes. Affixes are bound (non-free) morphemes that combine with a root and possibly other morphemes to form a word. Affix classes have a small number of members, and the bulk of a language's vocabulary is composed of roots.

Two types of affixes familiar to speakers of English are prefixes (e.g., *un-* in *undo*) and suffixes (e.g., *-ing* in *laughing*). The notion of attaching affixes before a root or stem, or after a root or stem is consistent with a purely concatenative view of morphology. However, the concatenative model does not work well with other types of affixes found in many languages--the so-called non-concatenative morphemes. Examples include infixes, circumfixes, root-and-pattern morphemes, and reduplication. A morphological parser that is designed to work with any of the world's languages should be capable of handling both concatenative and non-concatenative morphemes. Because the non-concatenative morphemes are not as familiar to English speakers as are the concatenative ones, the following will provide examples of each of the non-concatenative morphemes listed above.

Infixes are affixes that are embedded in another morpheme, typically at a morpheme boundary. Infixes do not occur in Indo-European languages (except in loan words), but are common in a number of language families, especially American Indian, Asian, and Semitic languages [24:195]. An example is Tagalog (from [66:12-13], cited by [6:9]), in which the root *sulat* is interrupted by a focus morpheme (*-um-* or *–in-*):

---

[10] See standard introductions to linguistics for an overview of morphology. For definitions see Crystal [24], and for an introduction to morphology from a generative grammar perspective see Katamba [51].

(4) a. sulat 'to write or writing (infinitive form)'
b. sumulat 'to write (with actor focus)'
c. sinulat 'to write (with object focus)'

Rather than interrupting a root as do infixes, circumflexes are discontinuous morphemes that "wrap" the root or stem of a word. Example (5) is from Dolan [27:78], cited by Sproat [68:50].

(5) a. besar k + besar + an
big bigness

b. bangun k + bangun + an
arise awakening

The circumflex *k -an* changes verbs or adjectives into abstract nouns.

The classic example of root-and-pattern morphemes are the Semitic languages. The following discussion is based on Sproat [68:51]. In the Semitic languages, three elements are required to form a verb stem: a root, a vowel pattern, and a template. The root typically contains three consonants. The vowel pattern is used to indicate the voice and aspect of the verb, and the template indicates the class of the derived verb. Sproat provides the following example (6) from McCarthy [57:134]:

(6)

Binyan[11]	**ACT(A)**	**PASS(UI)**	**Template**	**Gloss**
I	*k*A*t*A*b*	*k*U*t*I*b*	CVCVC	'to write'
II	*k*A*tt*A*b*	*k*U*tt*I*b*	CVCCVC	'cause to write'
III	*k*AA*t*A*b*	*k*UU*t*I*b*	CVVCVC	'correspond'
VI	tA*k*AA*t*A*b*	tU*k*UU*t*I*b*	tVCVVCVC	'to write to each other'
VII	n*k*AA*t*A*b*	n*k*UU*t*I*b*	nCVVCVC	'subscribe'
VIII	*kt*A*t*A*b*	*k*TU*t*I*b*	CtVCVC	'write'
X	stA*kt*A*b*	stU*kt*I*b*	stVCCVC	'dictate'

In (6), each example is derived from the verb *ktb*, which means 'to write', with the root shown in italics in each of the forms derived from the basic root. The active forms are shown in column two, and the passive forms in column three. The template describes the word shape based on combinations of consonants (C) and vowels (V). Where a specific consonant

is called for, it is specified in the template (e.g., the 'st' required for the Binyan X template). Note that in the case of some Binyanim (e.g. II), the template requires more consonants than are supplied by the verb root, resulting in the surface form of the root having four instead of three consonants, i.e., *k*A*tt*A*b*. Other Binyanim have templates that require more vowels than are supplied by the vowel pattern, resulting in a surface form that has a duplicated vowel, i.e., *k*AA*t*A*b* in Binyan III.

In reduplication, either an entire morpheme or part of a morpheme is duplicated. The following examples of morpheme reduplication come from this author's work on the Ogea language of Papua New Guinea.

(7) fai hilou 'good man'
man good

(8) fai hilou-hilou 'good men'
man good-good

(9) le -Ø[12]-na 'he is speaking'
speak-Tp-S3s

(10) le -le -Ø -na 'he is repeatedly speaking'
speak-speak-Tp-S3s

(11) le -tu -Ø -na 'he is speaking to him[13]'
speak-3sO-Tp-S3s

(12) le -tu -tu -Ø -na 'he is repeatedly speaking to him'
speak-O3s-O3s-Tp-S3s

(13) tau -Ø -ni 'I plant (a single object)'
plant-Tp-S1s

(14) tau -tau -Ø -ni 'I repeatedly plant (a single object)'
plant-plant-Tp-S1s

---

[11] Traditionally in the analysis of Semitic verbs, derivational classes have been called *binyanim*, with the singular being *binyan*. Each Roman numeral refers to a specific derivational class.

[12] This is a zero morpheme. That is, there is meaning to the absence of a tense suffix in Ogea.

[13] Although glossed as 'he', Ogea is actually neutral in pronouns, object suffixes, and subject suffixes, and does not mark gender.

(15) tata -ru-∅ -ni 'I plant (many objects)'
plant-pl-Tp-S1s

(16) tata -ru-tata -ru-∅ -ni 'I repeatedly plant (many objects)'
plant-pl-plant-pl-Tp-S1s

In Ogea, there are no affixes to mark a noun as plural. Instead, if a noun is modified by an adjective, plurality can be indicated by reduplication of the entire adjective as shown by comparison of (7) and (8)[14]. Reduplication is also used to indicate repetitive action. In (10), the root is reduplicated to indicate repeated action. Though this example does not show it, in such reduplication the entire root is reduplicated. If an object suffix is present, the object suffix is reduplicated instead of the verb root. In (12), the third person object suffix *-tu-* is reduplicated to indicate that someone is speaking to someone else repeatedly. Examples (15) and (16) illustrate partial reduplication. In Ogea, with verbs of a certain class, the first syllable of the root may be reduplicated and a plural marker -ru- added to form a stem as in (15). In (16), the entire stem is reduplicated to indicate repetition of action. Sproat stresses that replication is unusual in that unlike other types of morphemes, computational approaches to replication require memory of a previous string [68: 60].

In his survey of morphology, Sproat [68] also discusses subsegmental morphology (e.g., a morpheme indicated by a subsegmental feature), zero morphology (a morpheme indicated by the absence of an explicit string), and subtractive morphology. These and other phenomena described by Sproat underscore the defectiveness of a simplistic definition of affixes in particular, and morphemes in general. Sproat suggests that an affix is "…any kind of phonological expression of a morphological category μ that involves the addition of phonological material." [68:51]. He suggests that a morpheme should be thought of as

[14] If there are no adjectives modifying a noun, plurality can be indicated by use of a plural pronoun.

"…more properly constituting an ordered pair in which the first member of the pair represents the morphological categories which are expressed by the morpheme—i.e., its syntactic and semantic features—and the second member represents its phonological form, along with information on how the form attaches to its stem." [68:65].

Another way to view affixes is to classify them as either derivational or inflectional. Derivational affixes combine with roots to form a stem, and often result in a new word that has a grammatical class different from the original word[15]. For example, the suffix *-ize* is added to *national* (an adjective) to form *nationaliz*e (a verb). Inflectional affixes attach to stems, do not change the class of a word, and signify grammatical meanings such as person, number, and tense. Stems consist of one or more roots (e.g., *blackbird* is a stem with two roots) and optionally one or more derivational affixes.

Sometimes a morpheme will have phonologically conditioned variants. These variants, termed allomorphs, are conditioned by phonological phenomena when morphemes co-occur in a word. That is, the phonological properties of morphemes interact with and influence each other. For example, as noted in chapter one, in English, the plural morpheme {-s} has three allomorphs, {-s}[16], {-z}, and {-ɪz}, as seen in (17) - (19).

(17) bets
(18) beds
(19) places

The study of phonologically conditioned variance of morphemes is the subject of morphophonemics. Morphophonemic rules may be developed to predict or explain the allomorphs of a morpheme. For example, the difference between the {-s} and {-z}

---

[15] Derivational morphology is also known as lexical morphology.

[16] Phonological representations of morphemes will be in a non-italic font, delimited by braces, e.g., {-s}.

allomorphs of the plural morpheme in English is voicing. The choice between the voiced and voiceless variants is determined by the voicing of the final non-sibilant consonant in the root (e.g., in (17), the 't' in *bets* is voiceless, and in (18), 'd' in *beds* is voiced). And, the occurrence of the {-ɪz} variant is dependent on the root ending with a sibilant (e.g., 's'). It should be noted that whereas morphophonemics generally deals with phonological processes operating on underlying phonological forms of words, NLP usually deals with orthographic variants via spelling rules [3]. Phonological and orthographical variation are directly related, yet separate, notions. Unless noted otherwise, discussion of morphophonemics in this paper refers to orthographic representations of morphophonemic phenomena.

The study of morpheme co-occurrence is called morphotactics. Morphotactic rules govern whether a particular morpheme can occur in a word if another morpheme is present or absent. The notion of morphotactics is generally associated with the item-and-arrangement viewpoint of word analysis. This perspective views words as linear sequences or arrangements of morphemes [24]. The item-and-arrangement model is associated with the American Structuralist school of linguistics. An alternative viewpoint to item-and-arrangement is item-and-process, which views word analysis from the perspective of processes that act on a word to form a new word. The item-and-process model of word analysis has been dominant in linguistic theory since the advent of transformational grammar, and can more easily account for non-concatenative morphemes than can the item-and-arrangement model. In the item-and-process approach, lexicons contain the underlying form of morphemes, and rules are applied to generate the surface form. However, there are times when the item-and-arrangement model is chosen for NLP systems for pragmatic

reasons, as will be seen below with AMPLE. In fact, Sproat [68:205] notes the predominance of the item-and-arrangement model in NLP.

2.2.2 The Role of the Morphological Parser in NLP

Morphological parsing is an analysis of a word to identify its constituent morphemes. A morphological parser is a computer program that performs a morphological analysis of words. For purposes of this paper, unless noted otherwise, the phrase 'morphological parsing' implies parsing that is performed computationally.

For many years, computational approaches to morphology were neglected. Antworth [3:5] provides a discussion of this fact. He points out that economically dominant languages have morphologies that are relatively simple compared to other of the world's languages. Because research funds were available for these economically dominant languages, and because morphology was not a pressing issue for languages with simple morphology, research tended to focus on syntactical rather than morphological parsing. The parsing of morphology in languages such as English was usually handled by ad hoc methods. Despite the cliché, it is often true that necessity is the mother of invention. A computational approach to highly inflected non-Indo-European languages requires techniques to handle morphology. While most highly inflected languages are not economically dominant, fortunately Finnish is an exception[17]. Funding for research on computational approaches to Finnish morphology has been available. Finnish work in particular played a central role in the underlying model used in two-level parsers such as SIL's PC-KIMMO [2, 3]. (More will be said about two-level parsers below.) Work on morphologically complex languages has

[17] Finnish belongs to the Finno-Urgric language family, and is unrelated to Indo-European.

benefited approaches to computational morphology for English and other economically dominant languages that have relatively simple morphology.

Morphological parsing plays a role in a variety of NLP tasks. For example, AMPLE [70], can be used as:

- a means to test morphological hypotheses about a specific language,
- an exploratory device to uncover morphological phenomena in a language,
- a spell checker for highly inflected languages,
- input to machine-translation from a dialect or language into closely related dialects or languages,
- a means to produce glossed text.

Glossed text typically consists of natural language text with the words broken into morphemes, along with an interlinear gloss of each morpheme on a second line.

Both Weber et. al [70] and Hindle [46] mention the role of morphological parsing as a component to syntactic parsing. Except for isolating languages[18], it is difficult to imagine a syntactic parser that would not rely on the results of morphological parsing as input to determine the syntactic structures of sentences.

Sproat [68] discusses the applications of not just morphological parsers, but computational morphology in general. Applications discussed include those mentioned above, as well as the following:

- an aid to identifying word boundaries in text,
- dictionary construction,

---

[18] Crystal [24:205] states that isolating languages are ones with single morpheme words that do not have variant forms. In such languages, syntactic roles are indicated by word order rather than by inflection.

- lemmatization (identification of the dictionary form of a word occurring in text),
- text-to-speech processing,
- speech recognition,
- automatic orthographic conversion (e.g., morpheme-based Kana-Kanji conversion for Japanese).

Natural language understanding is a task of NLP that is not yet fully realized. However, natural language understanding will require the input of morphological parsers. Natural language cannot be understood without the identification and knowledge of the meaning of individual morphemes of a word.

Morphological parsing can also play a role in computational lexicography. Biber [5] describes the use of factor analysis to computationally identify word senses. This is important for lexicography (the art and science of dictionary making) and lexicology (the study of a language's vocabulary), and could be used for information retrieval. In Biber's approach, word co-occurrence information from large corpora is grouped via factor analysis. The results obtained appear to correlate with word senses. Biber suggests that his approach could produce better correlations if co-occurrence data included not just the words themselves, but grammatical categories. Unless text was hand tagged with category information, implementation of Biber's suggestion would presumably require a morphological parser, and, perhaps, a syntactic parser. Information from a syntactic parser can be useful to a morphological parser to resolve ambiguity in the case of morphemes that can potentially be assigned to more than one category.

In information retrieval, morphological parsing is used by some retrieval schemes to normalize morphological variants of retrieval terms through a technique called stemming. A

stemmed word is a single canonical form that is used to match query terms to the index terms in text [55]. The canonical form can be the root or stem of the word, with all affixation stripped away, or a standard citation form, which is the dictionary entry (lexical or lemmatized) form. Sproat [68:13] observes that stemming algorithms are often trivial in languages such as English, but more difficult in highly inflected languages. These languages often require morphological components for stemming algorithms. Various morphology based approaches to stemming are described by Hull [47] in his case study of techniques used to evaluate stemming algorithms, and by Hull and Grefenstette [48].

Before describing AMPLE, the morphological parser of concern to this research, some discussion of mainstream approaches to morphological parsing is in order. The majority of approaches are based on two-level models and use finite-state mechanisms. As noted above, research on computational approaches to Finnish morphology has made a large impact on the field. This is due to the work of the Finnish computational linguist Kimmo Koskenniemi [54]. The following discussion draws from Antworth [3] and Sproat [68].

Koskenniemi developed KIMMO, an approach to Finnish morphology that uses a two-level model for phonological rules. When Koskenniemi attempted to model ordered phonological rules as a single combined automaton of finite state transducers, he found that the automaton became prohibitively large. Koskenniemi's solution was to model phonological rules as a bank of automata running in parallel [3:6]. This approach dictated that only two levels be used—an underlying level and a surface level—rather than multiple levels as is the case with traditional generative phonology. In contrast to generative phonology, the phonological rules of the two-level model are not ordered. All rules are

applied simultaneously. A further implication of this approach is that the two-level model may be used both for recognition and generation of words [3:9].

In addition to the dominance of two-level approaches, as with Koskenniemi's approach, most morphological parsers rely on finite-state machines. This is true for not only computational morphology specifically, but for computational linguistics in general. Sproat [68:125-143] discusses both finite-state morphotactics and finite-state phonology. He notes that concatenative morphology in which "…the allowability of a morpheme in position *n* depends only upon the morpheme that occurred in position *n – 1*…." can easily be modeled using finite-state machines [68:127]. Finite-state phonology in KIMMO and other two-level models rely on finite-state transducers (FSTs). FSTs accept pairs of symbols rather than single symbols. That is, next-state transitions are based on the current-state plus a pair of symbols, rather than current-state plus a single symbol as is the case with ordinary finite-state automata.

2.2.3 The AMPLE Morphological Parser

As noted above, AMPLE is a morphological parser that was developed by the Summer Institute of Linguistics (SIL). AMPLE consists of control files, dictionary files, text files, output files, and, of course, software programs. A text processing component of AMPLE uses the information from control files to identify and extract words from the text files, and passes them in a normalized form to an analysis component. Dictionary files provide information about each morpheme in the language, including constraints that are used by the analysis subsystem to test hypothesized analyses. Both built-in and user-defined tests are contained in a control file. Finally, each word is written to an output file with the analysis or analyses for that word. It should be noted that the AMPLE software is language

independent, and embodies an abstraction of those characteristics common to language specific parsers. The information about the language whose text is being processed is contained in the external files. Unlike most morphological parsers, AMPLE does not rely on finite-state machines, though some mechanisms used by AMPLE could be modeled using finite-state machines as will be discussed below.

AMPLE has evolved over the years as different individuals have improved it, taking into account experience using AMPLE with various languages around the world. Indeed, AMPLE is one of the few morphological parsers intended for use with any natural language. It is commonly used for a number of purposes, listed above, including Computer-Aided Related Language Adaptation (CARLA), which is the machine-translation of text from a dialect into one or more dialects, or translating from a language to one or more closely related languages. AMPLE is described in Simons [65], Weber et. al [70], Buseman et. al [9], and Black and Black [6]. Possibly the only non-SIL literature that discusses AMPLE is a brief overview by Sproat [68]. The description of AMPLE presented below draws from these five references. An extended example of the use of AMPLE with the Ogea language is provided in Appendix Two.

AMPLE parses a word by systematically attempting to match sub-strings of the word with candidate prefixes, infixes, roots, and suffixes listed in dictionary files. If unconstrained, this process would be exhaustive, and would return all possible combinations of morphemes that match, including incorrect combinations. In order to avoid incorrect analyses, the combinations of morphemes to be considered by the parser are constrained by various types of tests, described below. Even with constraints, ambiguity can be encountered since human languages are naturally ambiguous, and therefore more than one legitimate

analysis may be possible for a word. Unlike some parsers, AMPLE returns all possible parses, not merely the first legitimate analysis found.

In common with most morphological parsers, AMPLE follows the item-and-arrangement model of word analysis. An implication of this choice is that all surface forms (allomorphs) are listed in the lexicon or dictionary. This model was chosen for pragmatic reasons, not theoretical ones [70]. Although the item-and-arrangement model does not account for all morphological phenomena, as was discussed above, and is inadequate as a theory of morphology, it is none-the-less a productive notion for the analysis of words, and the model most commonly used for morphological parsers per Sproat [68:205].

The AMPLE software consists of two modules—TEXTIN and ANALYSIS. The purpose of TEXTIN is to identify individual words in a text that is being processed. For example, the individual words in a sentence must be extracted from the sentence. AMPLE expects that texts that are used as input to the parser have been marked up using SIL's Standard Format Markers as tags. These markers begin with an escape sequence, which is the backslash symbol '\', followed by one or more characters, without a space between the backslash and the other characters. For example, *id* is used to mark the identifier for a file, e.g., the filename. Format markers can be partitioned into tags of text that should be parsed, and tags of text that should not be parsed. AMPLE uses an input text control file which specifies tags that identify text to be excluded from the parse, or alternatively, specifies tags that identify text to be included in the parse. In addition to the issue of which parts of an input file contain text to be parsed, there are the issues of orthography, capitalization, and identifying word formation characters. AMPLE provides a mechanism for the user to provide control information for each of these issues via a text input control file. TEXTIN

normalizes the individual words in a text by removing format markers, punctuation, and white space, and by converting words to their lower-case form. TEXTIN can also apply orthographical changes as directed by the user. Whereas the input text may have been written in a practical orthography, for purposes of morphological analysis by AMPLE it may be desirable to convert the text to an orthography that is more linguistic in nature. One example given by Weber et al. [70:17] is the Latin *x* used in English, which is actually a sequence of a stop plus a fricative, that is, *ks*. In converting from a practical to linguistic-oriented orthography, the *x* could be converted to *ks*. Such conversions may be required in order to state phonological rules for allomorphs. TEXTIN preserves the original orthography, punctuation, and formatting data from the input file, and passes individual words to the ANALYSIS module for parsing. Subsequent to parsing by ANALYSIS, the information saved by TEXTIN can be used to recreate the text, incorporating the parsing analysis and morpheme glosses.

AMPLE makes use of up to eight types of files, four of which are control files, and four of which are dictionary files[19]. The analysis data file and the dictionary code table file are mandatory control files. The analysis data file serves two main purposes. First, it defines the categories and properties that are used for dictionary entries (a form of metadata). Second, it defines tests to constrain the co-occurrence of morphemes[20]. More will be said, below, about tests. The purpose of the dictionary code table file is to allow users to define their own field codes to use for dictionary entries. The file contains the mapping between these user-defined codes and the ones that AMPLE uses internally. One application of this

---

[19] AMPLE allows the prefix, infix, suffix, and root files to be combined into a single file.

[20] Whereas an individual morpheme may have its own co-occurrence constraints, those specified in the analysis data file should apply to more than one morpheme.

feature is to support use of AMPLE by non-English speakers. This allows non-English speakers to define field codes that are meaningful in their own language, which are automatically converted to AMPLE's internal codes. The two optional control files are the dictionary orthography change table file and the text input control file. The dictionary orthography change table file controls the conversion of dictionary entries from one orthography to another. The text input control file controls the conversion of text from one orthography to another, and also defines special formatting information and special characters to be added to the assumed word formation characters[21]. The dictionary file types include a root dictionary, a prefix dictionary, an infix dictionary, and a suffix dictionary. The types of affix dictionaries required depends, of course, on the language being processed, since not all languages have, say, prefixes and infixes. Following is a description of the record format used for roots and affixes in AMPLE. This description is useful for understanding the discussion of how AMPLE constrains analyses. (The AMPLE documentation provides a full BNF description.)

The metamodel or record format model for AMPLE root dictionary entries shown in (20) is adapted from Weber et al. [70:125], and the model for AMPLE affix dictionary entries shown in (21) is adapted from Weber et al. [70:112]. In this author's adaptation, characteristics of the relationship between a morpheme and its fields and between an allomorph and its fields are indicted using notation borrowed from entity relationship diagramming[22].

---

[21] AMPLE assumes that both upper and lowercase a-z are word forming characters.

[22] A leading zero (e.g., 0:1) means the relationship is optional. A leading one (e.g. 1:1) means the relationship is mandatory. On the right side, a one (e.g. 1:1) means there is only one occurrence of the field if it occurs. An 'M' (e.g. 1:M) means that there may be many occurrences of the field.

```
(20)  root[23]
        |---- 0:1 etymology or gloss
        |---- 0:1 underlying form
        |---- 0:M morpheme property
        |---- 1:M category
        |---- 0:M morpheme co-occurrence constraint
        |---- 1:M allomorph (string of characters)
                |---- 0:M allomorph property
                |---- 0:M morpheme environment constraint
                |---- 0:M string environment constraint
                |---- 0:M comment

(21)  Affix
        |---- 1:1   morph name (gloss)
        |---- 0:1   underlying form
        |---- 0:M morpheme property
        |---- 0:1 order class
        |---- 1:M category pair (from/to)
        |---- 0:M morpheme co-occurrence constraint
        |---- 0:M infix location constraint (for infixes
                                            only, and then mandatory)
        |---- 0:M string environment constraint
        |---- 1:M allomorph (string of characters)
                |---- 0:M allomorph property
                |---- 0:M morpheme environment constraint
                |---- 0:M string environment constraint
        |---- 0:M comment
```

Note in both (20) and (21) that morpheme co-occurrence constraints apply to the morpheme as a whole, but environment constraints apply to allomorphs of the morpheme. **Error! Reference source not found.** lists the various types of fields used for both root and affix dictionary entries and provides a description of their contents and usage.

AMPLE provides a number of methods to constrain possible morpheme combinations. Many of these methods are based on linguistic phenomena, but in the event a linguistically based constraint has not yet been identified or does not exist, provision is made for ad hoc constraints. The purpose of constraints is to reduce ambiguity and to block incorrect analyses. There are seven types of constraints that a user can define to AMPLE. Three of these constraints are morphophonemic in nature, and the other four are morphotactic

[23] Not shown are the Elsewhere Allomorph, Feature Descriptor, and Do Not Load fields.

in nature [6]. In contrast, KIMMO only allows one kind of morphophonemic constraint, and one kind of morphotactic constraint [6]. The morphophonemic constraints allowed by AMPLE are string environments, morpheme environments, and user-written tests. The morphotactic constraints are order classes, morpheme co-occurrences, category mappings, and ad-hoc pairs. Each of these is discussed below. In these discussions, two terms will be used that require some explanation. Conceptually, in AMPLE, a word is viewed as consisting of a linear sequence of morphemes (the concatenative model). It is possible to think of having a pointer to a specific morpheme in a word. The morpheme pointed to at any one point in time is the 'current' morpheme. Reference will be made to a left morpheme (a morpheme one position to the left of the current morpheme), and a right morpheme (a morpheme one position to the right of the current morpheme). In the case of a left-to-right parse, the left morpheme has been tentatively identified by the parser as occurring in the word being parsed.

Morphophonemic constraints specify environments that constrain the occurrence of allomorphs. Note that in AMPLE, all allomorphs are listed in the lexicon, and the role of phonological rules is to constrain which allomorphs are acceptable as a valid parse. This is in contrast to morphological parsers that rely on phonological rules to recognize morphological variants rather than listing each allomorph in the lexicon (the item-and-process model noted above).

Table 1. AMPLE Dictionary Field Types

Name	Description	Purpose
Affix Morph Name	Unique identifier for a morpheme that is an affix. Feature information, e.g., person, number, gender for nouns, is typically used as the name.	Identifies a morpheme used in ad hoc pairs, morpheme co-occurrence constraints, morpheme environment constraints, successor tests, and final tests. Used as the identifier when writing to the output analysis file.
Allomorph	A string of characters representing a variant of a root or affix as found in input texts.	Allows AMPLE to recognize the allomorph.
Category	For roots, the morphological category (e.g., Noun, Adjective). For affixes, the from/to categories.	Used for user-defined successor and final category tests.
Infix Location	Lists the type of morpheme in which the infix may occur (prefix, root, or suffix), and string environment constraints.	Constrains the searches AMPLE makes to locate an infix in a morpheme.
Morpheme Co-Occurrence Constraint	Indicates whether a morpheme may or may not occur depending on the presence or absence of another morpheme.	Used by the built-in final test that uses morpheme co-occurrence constraints.
Morpheme Environment Constraint	Specifies a constraining environment based on morph names and morpheme classes.	Used by the built-in MEC final test to constrain allomorphs, and as part of MCCs.
Morpheme Property	Specifies a characteristic that conditions the occurrence of allomorphs of adjacent morphemes.	Used in user-written tests and in the built-in MCC and MEC final tests
Morpheme Type	Identifies whether the dictionary entry is for a prefix, infix, suffix, or root.	Used to identify the record type when only a single (unified) dictionary is used.
Order Class	The relative order in which a morpheme may occur in a word. The values range from -32767 and 32767. The default value is zero. A root is implicitly assigned a zero.	Used for order class tests.
Root Etymology or Gloss	An etymological form.	When occurs, is used in place of the root.
Root Morph Name	Uniquely identifies a morpheme that is a root. Typically, either a gloss or etymology is used as the name.	Identifies a morpheme used in ad hoc pairs, morpheme co-occurrence constraints, morpheme environment constraints, and tests. It is the default name used when writing to the output analysis file.
String Environment Constraint	Specifies a constraining environment based on strings and string classes, namely the characters that precede or follow an allomorph.	Used by the built-in successor test that uses string environment constraints.
Underlying Form	Specifies the underlying form (instead of the surface form) of the morpheme.	Used for writing to the output analysis file.

The first morphophonemic constraint to be discussed is the string environment constraint. String environment constraints block incorrect analyses by constraining a morpheme's occurrence based on the character(s) immediately preceding or following the morpheme. These constraints are expressed as environmental rules. An example from Ogea[24] is shown in (22)-(29).

```
(22)  tu      -∅ -na                ({tuna})
      give.O3s-Tp-S3s
      'he gives it to him'

(23)  tu  -∅ -na                    ({t  na})
      poke-Tp-S3s
      'he pokes it'

(24)  tum -bo-na
      poke-TO-S3s
      'while he poked it'

(25)  tun -de  -wa -u
      poke-well-imp-S3s
      'poke it well!'

(26)  tung-g -a  -ne -nga
      poke-TO-Trp-S3s-SR
      'As they poked it…'

(27)  \scl B p b  | bilabials
      \scl A t d  | alveolars
      \scl V k g  | velars
      \scl  O     | all other consonants[25]

(28)  \r tu       | non-nasalized u

(29)  \r tuN[26]     | nasalized u
      \a tum / _ [B]
      \a tun / _ [A]
      \a tung[27] / _ [V]
      \a tu[28] / _ [O]
```

[24] Ogea is a Papuan language spoken in Papua New Guinea. It is described in [21] and [22].
[25] It is not valid in AMPLE to specify 'all others' in this way, but is used for simplicity in the example.
[26] N is used in linguistics to signify a nasal of an unspecified point of articulation.
[27] As in English, 'ng' is the orthographic representation for a voiced velar nasal, ŋ.
[28] *tu-* is the orthographic representation of the root {t -} in Ogea.

In example (23), above, the underlying vowel { } is nasalized. Although nasalization is phonemic[29] in Ogea, it is not represented in Ogea orthography since context allows it to be disambiguated from its non-nasalized counterpart, {u}. This is illustrated by comparing (22) and (23), which phonologically constitute a minimal pair[30], but are orthographically identical. When an Ogea morpheme that ends with a nasalized vowel is followed by a morpheme that begins with a stop, the underlying nasal consonant is realized on the surface as a nasal of the same point of articulation as the following stop[31]. String classes such as (27) and string environment constraints such as (28) and (29) could be used to prevent AMPLE from considering a parse path that considered {tu-} as the root of (24), (25), or (26), when {tuN-} is the correct root. (27) establishes classes of characters (e.g., B for bilabial stops), which are used as shorthand in other rules. (29) states that the morpheme {tuN-} is realized as {tum-} before bilabials, {tun-} before alveolars, {tuŋ-} before velars, and {t -} elsewhere.

Black and Black [6] discuss how string environment constraints in AMPLE can be used to handle other phenomena such as reduplication and epenthesis. Whereas string environments constrain the occurrence of allomorphs based on strings and string classes, morpheme environments constrain the occurrence of allomorphs based on morphemes and morpheme classes.

The first morphotactic constraint to be discussed is that of orderclass. It is commonly accepted in linguistics that morphemes do not concatenate in an unordered manner. Natural

---

[29] A phonemic contrast between nasalized and non-nasalized vowels means that in Ogea the meaning of a word changes based on whether a vowel has nasalization.

[30] Minimal pairs are two words that have different meanings and differ by only one phoneme.

languages impose rules regarding which morphemes may occur after other morphemes. In other words, word structure consists of ordered morpheme class positions that may only be filled by members of a particular class. For example, in the Ogea language of Papua New Guinea, medial verbs consist of a verb root followed by up to five classes of suffixes, e.g.:

(30) le -nigi-boro -wa-ne -nga
speak-3pO -completely-rp-3sS-SR
'having said everything to them…'

3pO=third plural Object, 3sS=third singular Subject, rp=remote past, SR=Switch Reference[32]

In Ogea, suffix classes occur in the following order: object, aspect, tense, subject, and switch reference. This means that a tense suffix cannot appear before, say, an object suffix. Ordering, or position, rules such as these may be defined to AMPLE and exploited to eliminate invalid parses. A parse can be discarded if it would result in a violation of ordering constraints. Order class constraints are indicated by assigning a number to each class. In AMPLE, order class values range from –32,767 to 32,767. If an order class is not explicitly stated, AMPLE uses zero as a default value. A root is implicitly assigned zero as its order class value. Negative numbers are used for prefixes, and positive numbers for suffixes. The orderclass value for each suffix must be larger than the preceding one to be considered valid. Prefixes are constrained in a similar manner, but using negative numbers.

---

[31] Ogea syllables are open, that is, they end in a vowel, and never end with a consonant, with the exception of syllables that end with a nasalized vowel, where the underlying nasal is realized on the surface string in certain environments.

[32] Some linguists use the phrase 'different subject' rather than 'switch reference'. However, in many languages, such as Ogea, 'switch reference' is preferred because if one of the individuals involved in an action in one clause goes on to do another action, that individual is technically a different subject from the first clause, but considered to be the same referent. The medial clause will be marked for 'same referent', though only one of the referents goes on to perform the action of the second clause. The grammatical subject is plural for the medial clause, but singular for the subsequent clause.

Drawn from categorial grammar, category mapping constraints are constraints based on the allocation, or partitioning, of roots and affixes to classes. For example, in some languages, a wrong analysis can be avoided by partitioning roots into a noun class versus a verb class, and partitioning affixes into those that bind with items of the noun class versus those that bind to those of the verb class. The class categories that are productive for preventing wrong morphological analyses will vary from language to language. In AMPLE's category mapping scheme, a from-category and a to-category are specified for each affix. An affix can only be considered as a valid path for a parse if its from-category matches the to-category of another affix. The other affix may be either adjacent or non-adjacent depending on the mapping strategy being used.

In addition to partitioning, the notion of category mapping in AMPLE includes two other concepts—obligation and propagation. Obligation states that a word must have affixes of certain morpheme classes present in order to be considered well formed. (For example, in Ogea, a final verb must have a tense suffix and a person suffix to be well formed.) In the event that the current morpheme does not fulfill the category obligation of the obligating morpheme, the obligation is passed to the next morpheme to the right or left of the current morpheme, depending on the category mapping scheme being used. This passing of obligation is called propagation in AMPLE.

Although AMPLE does not do so, AMPLE's full category mapping scheme could be implemented using finite-state machines (FSMs). The FSM nature of AMPLE's category mapping scheme is illustrated by Weber et. al's example from Quechua [70:31]. They posit categories based on a verb's valence. In linguistics, the valence of a verb specifies the number of arguments that the verb takes. Arguments can be on a syntactic level or, in the

case of highly inflected languages, valence can also be on a morphological level. For example, on a syntactic level, a verb with a valence of two may have both a subject noun phrase and an object noun phrase. On a morphological level, a verb with a valence of two may have both a subject affix and an object affix. For Quechua, Weber et. al posit a category of V2 (bivalent) for a verb that may have an object suffix, and V1 (univalent) for a verb that may have a subject suffix. In their discussion of how a parse would occur using valence-based categories in Quechua, Weber et al. state that

> It is significant that the object marking suffixes (-ma(:) 'first person object' and -shu 'second person object') reduce the valence from V2 to V1. This is like saying that these suffixes fulfill the obligation to have an object marker but not the obligation to have a subject marker. [70:31].

The categories V2, V1, and V0 are applied to a single verb during the parse process. As a parse proceeds, the obligation decrements from V2 to V1, and from V1 to V0. When V0 is reached, it indicates that the verb is now complete, and that no more affixes need follow. This indicates that the parse analysis is complete. It appears, therefore, that the notions of obligation and propagation, as used in AMPLE, could be viewed as parse states of an FSM. The notion that the attachment of a particular affix reduces the valence could be viewed as a change of parse state. From this perspective, obligation rules would be state-transition rules governing parse states. When a parse is in state V2, the parser would know to accept matching affixes that are object markers, or any other affix category associated with state V2. When the parse is in the state V1, the parser would accept matching affixes that are subject markers, or any other affix category associated with state V1. From a state-transition perspective, propagation in AMPLE would mean a transition to the same state, rather than to the next state. Category rules in AMPLE, then, could be implemented as rules about parse

states in an FSM. In fact, FSMs are well suited for modeling the linguistic notions of obligation and category mapping.

Morpheme co-occurrence constraints are the third type of morphotactic constraint that AMPLE recognizes. Whereas morpheme environments constrain the occurrence of allomorphs, morpheme co-occurrence constraints state rules about whether a specific morpheme or morpheme class can appear or cannot appear if another morpheme or morpheme class is present or absent in the word. In fact, the recognition of a morpheme may depend on the recognition of some other morpheme as seen in (31) - (34), from Ogea.

```
(31)  yafai              'he sat'
      yaf-a -i
      sit-Tp-S3s

(32)  yafainga           'after he sat, (someone else did something)'
      yaf-∅ -a  -i  -nga
      sit-TS-Trp-S3s-SR

(33)  yafagainga         'while he sat, (someone else did something)'
      yafa-g -a  -i  -nga
      sit -TO-Trp-S3s-SR

(34)  yafagai                  'he habitually sat'
      yafa-g  -a  -i
      sit -hab-Trp-S3s
```

In (32), the recognition of a zero morpheme indicating temporal succession cannot occur until the final suffix *-nga* (switch reference) is encountered. And in (33) and (34), the identification of *-g-* as a suffix indicating temporal overlap versus one indicating habitual action cannot be made until the end of the word is reached. Temporal overlap, temporal succession, and switch reference apply to medial verbs occurring in a chain of clauses. When the referent changes, a medial verb is marked by the switch reference suffix. Morpheme co-occurrence constraints could be used to allow AMPLE to successfully identify the morphemes in this example.

In the event that a linguist cannot identify a linguistic-based constraint or test, AMPLE provides an ad-hoc mechanism to eliminate an analysis by explicitly naming two morphemes that cannot occur together. Such ad-hoc pairs are a last resort constraint, and should be avoided where possible.

As has been seen, AMPLE provides a wide variety of methods to constrain analyses. These methods are in some ways overlapping. That is, a specific linguistic phenomena may be handled by more than one type of AMPLE constraint. However, Weber et al. [70:25] point out that allomorphs themselves are the strongest type of constraint. If there is no match what-so-ever between the substring of a word and the allomorphs of a morpheme, that morpheme does not occur in the word. The authors also discuss the relationship between the length of an allomorph and the likelihood of ambiguity. In general, the shorter the allomorph string is, the greater likelihood there is for ambiguity. Zero morphemes or single segment morphemes are the worst case scenarios. Zero morphemes lack even a single segment and therefore cannot constrain analyses. On the other hand, an allomorph string may be so long that it is unique, and no further constraint will be necessary beyond that of the string itself.

The information provided in the dictionary files provides much of the raw material, so to speak, for AMPLE to perform tests. Further information and many of the actual tests themselves are defined in the analysis data file. Tests include both built-in and user-defined successor or final tests. Successor tests are applied to each morpheme at the time it is encountered during the parse, and final tests are applied to each morpheme in the word after the entire word has been analyzed. Final tests check “long distance” constraints—that is, constraints that apply to morphemes that are not contiguous to each other. In addition to tests, the analysis data file contains declarations and information on prefixes, infixes, roots,

and suffixes. The declarations provide metadata specifications for information stated in the dictionary files. These declarations include allomorph properties, morpheme properties, categories, category classes, morpheme classes, and string classes. Properties, categories, and classes must first be declared in order to be used by dictionary entries. In earlier versions of AMPLE, there was a limit of 255 properties (divided between allomorph and morpheme properties) and a limit of 255 categories. Users may now increase this number through a maximum property field in the analysis data control file. Morpheme properties hold for all allomorphs of a morpheme, whereas allomorph properties only hold for a specific allomorph. For each type of morpheme (prefix, infix, root, and suffix), the maximum number of occurrences of that morpheme type in a word is declared, successor tests are defined, and any required ad hoc pairs are stated.

The built-in tests provided by AMPLE include three successor tests (SEC_ST[33], ADHOC_ST, ROOTS_ST) and two final tests (MEC_FT, and MCC_FT). The SEC_ST is a successor test based on string environment constraints. This test checks the string environment of the current morpheme to validate the analysis at that point. The ADHOC_ST is a successor test based on ad-hoc pairs, and the ROOTS_ST is a successor test based on compound roots. The MEC_FT is a final test based on morpheme environment constraints, and the MCC_FT is a final test based on morpheme co-occurrence constraints.

In addition to the built-in tests, AMPLE allows the user to define tests. Tests can be defined based on categories, order classes, properties, morpheme types, morpheme names, allomorph strings, surface strings, and adjacent words. It is possible to build complex tests

---

[33] SEC = String Environment Constraint, MEC = Morpheme Environment Constraint, MCC = Morpheme Co-Occurrence Constraint, ST = Successor Test, and FT = Final Test.

using various logical operators. Also, it is possible to write tests that are applied to all morphemes (either all morphemes preceding the one in question, or following it).

Users have a number of ways to control tests. Although built-in tests are automatically applied after the application of user-defined tests, the user can explicitly invoke the built-in tests first for efficiency reasons. Also, the user can control tests by allocating them to a specific morpheme type, i.e., prefix, infix, root, or suffix. Thus, AMPLE will only invoke a test if the test applies to the type of morpheme under consideration. Also, once a test is declared, it can be reused elsewhere.

The output of AMPLE is an analysis file. This file contains a record for each word in the input file. Each record contains one or more fields, the mandatory field being an analysis field. The analysis field provides the name of each affix (morphname), the root category, and the root gloss or etymology[34]. The analysis information is listed following the order in which the morphemes occur in the word. The optional fields include the word's decomposition, word and morpheme categories, properties, feature descriptors, underlying forms, the original word as found in the input text, formatting, capitalization, and any trailing nonalphabetic information (e.g., punctuation or whitespace). If the analysis is ambiguous (more than one analysis is possible), the record will contain the number of possible analyses and present each alternative analysis. If no analysis is possible, AMPLE will indicate this in the output record.

Before concluding this section, the observations of Sproat [68:202-205] regarding AMPLE will be mentioned. Sproat provides what is perhaps the only non-SIL review of AMPLE. Sproat notes that AMPLE is an exception to the dominance of finite-state-based approaches. At the time of writing, Sproat [68:204] states that "…AMPLE has the

---

[34] AMPLE uses a single field to contain either the gloss or etymology.

conceptually cleanest model of infixing of any morphological analysis system of which I am aware." He also notes that AMPLE is one of the few morphological parsers that handle other non-concatenative phenomena such as reduplication. However, Weber et al. [70:13] state that in AMPLE, "non-concatenative phenomena such as ablaut, vowel harmony, tone sandhi…" and submorphemic phenomena can only be handled indirectly via concatenative solutions. That is, by listing allomorphs and constraining their co-occurrence.

To summarize, AMPLE is a morphological parser with great power and utility, with morphophonemics based on traditional American Structuralist notions of morphology, and morphotactics based on a variety of linguistic approaches, including categorial grammar. As has been seen above, AMPLE provides many and various ways to constrain a parse analysis through both morphophonemic and morphotactic rules. This wealth of constraint mechanisms allows AMPLE to handle morphological phenomena from a great variety of the world's languages. AMPLE has proven to be an extremely useful tool within the SIL community for fieldwork in indigenous languages around the world.

2.3 The Lexicon

2.3.1 The Role of the Lexicon in Linguistic Theory

In both early transformational grammar, and linguistic theories preceding transformational grammar, the lexicon was viewed merely as the place to keep an alphabetic listing of a language's vocabulary and any irregularities. Predictable aspects of the language's lexemes[35], such as morphological rules, were handled separately from the lexicon. Syntactic rules were also stated outside the lexicon. During the 1980s and 1990s, there was a

[35] Crystal [24] describes a lexeme as an abstract unit that underlies grammatical variants, includes idioms, and is the unit traditionally listed as a separate dictionary entry.

movement toward lexicalization—the capture of increasing linguistic information in the lexicon itself, rather than in separate rules. This occurred both within transformational and non-transformational approaches to grammar.

Both Briscoe [8] and Ooi [61] suggest that lexicalization perhaps began with a 1970 article by Chomsky. Chomsky proposed using lexical redundancy rules rather than transformation rules to handle the nominalization of verbs [20]. Researchers expanded on Chomsky's proposal over the decades that followed. Per Ooi, lexicalization can be seen in the work of Jackendoff [49], in Lexical Functional Grammar (LFG), in Generalized Phrase-Structure Grammar (GPSG) [42], and in Head-Driven Phrase-Structure Grammar (HPSG) [62]. Competitors to transformational grammar, both GPSG and HPSG have been influential in computational linguistics and NLP [7:1]. GPSG and HPSG are classified as modern phrase structure grammars (PSGs) and rely on the notion of unification. Gazdar [41] traces lexicalization to two main factors: work on computational morphology in the early 1980s, especially the Finnish work, and the need for lexicons to encode information about subregularity. The ability to handle subregularity in the lexicon implied the ability to handle regularity as well, resulting in the movement of more and more rules into the lexicon.

Next, the nature of modern PSGs and the notion of unification will be elaborated for the following reasons. First, modern PSGs have played a key role in lexicalization. Second, as will be seen below, the lexical knowledge representation language, DATR, was developed in response to the lexical needs of modern PSGs, and specifically for unification-based approaches to syntax. It is useful to understand why the need has arisen for lexical knowledge representation languages like DATR.

The following description of both classical PSGs and modern PSGs[36] relies on Borsley [7]. Classical PSGs utilize grammars that consist of phrase structure rules with simple categories. For example:

(35) S → NP VP
(36) NP → article Noun
(37) VP → Verb NP
(38) Article → a, the
(39) Noun → child, flower
(40) Verb → saw

Using sets of phrase structure rules, such as (35)-(40), the sentence "The child saw a flower" may be represented as a simple tree, as in Figure 1. Such trees may be derived by

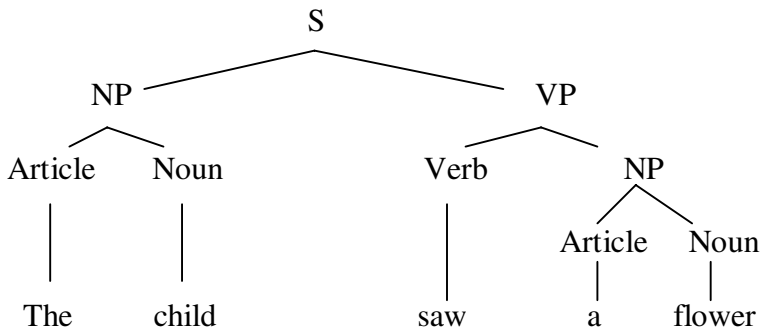

Figure 1. Tree Representation of Phrase Structure Rules

interpreting phrase structure rules to be re-write rules. As re-write rules, the left-hand element of the top-most rule (i.e., the *S* of rule (35)) is re-written as the right-hand constituents (i.e., the *NP* and *VP* of rule (35)), and to each constituent, a new re-write rule is applied, if it exists[37].

---

[36] Throughout the discussion of Borsley's assessment of PSGs, please keep in mind that by Modern PSGs, Borsley is referring specifically to GPSG and HPSG. In this author's discussion, Modern PSGs are used merely as an illustration, and no position is being taken as to the value of GPSG or HPSG over other modern grammar theories such as Government and Binding or Principles and Parameters.

[37] An alternative is to view phrase structure rules as admissibility conditions. As admissibility conditions, phrase structure rules are used to determine whether a given tree is well-formed.

Unlike transformational grammar, both classical and modern PSGs are monostratal. That is, there is a direct mapping from underlying to surface forms, without the intermediate levels allowed by transformational grammar. This means that the representation of the syntactic structure of a sentence is a single tree, rather than multiple, intermediate trees. The monostratal nature of modern PSGs makes them easier to implement computationally than transformational grammars. Also, for those versions of both classical and modern PSGs that are context-free, reference is only made to local trees. Local trees are trees (or sub-trees) with a depth of one, where every node except the root is a daughter of the root [42:45]. This means that when checking to see if a tree is well-formed, phrase structure rules are only applied to local trees. A tree is well-formed if all its local trees are well-formed. Borsley notes three differences between classical PSGs and modern PSGs. Modern PSGs use complex categories (sometimes called features), replace phrase structure rules with immediate dominance and linear precedence rules, and tie analyses with semantics. The difference most relevant to the topic at hand is the use of complex categories.

The problem with the simple, atomic categories of classic PSGs is that there are types of generalizations that cannot be captured as a single statement. Instead, the number of categories and rules are multiplied because many natural language expressions contain both similarities and dissimilarities. Simple, atomic categories only allow expressions to be assigned to the same category if they behave exactly the same way. Borsley provides several examples of the power of complex categories, including Welsh prepositional phrases. In Welsh, prepositions are marked for person, number, and gender, and there must be agreement with the noun phrase. Classical PSGs would require seven separate statements to capture the required rules. In contrast, modern PSGs can capture the generalization in a single rule:

(41)

$$\begin{array}{ccccc} \text{PP} & \rightarrow & \text{P} & & \text{NP} \\ & & \begin{bmatrix} \text{number} & \alpha \\ \text{person} & \beta \\ \text{gender} & \gamma \end{bmatrix} & & \begin{bmatrix} \text{number} & \alpha \\ \text{person} & \beta \\ \text{gender} & \gamma \end{bmatrix} \end{array}$$

The variables, α, β, and γ co-index the categories for P and NP so they agree in number, person, and gender. Categories in modern PSGs can be much more complex than the example above. In HPSG, lexical entries are very complex, as is seen in Figure 2, a lexical entry for 'she', taken from Pollard and Sag [62:20].

$$\left[\begin{array}{l} \text{PHON} < \textit{she} > \\ \text{SYNSEM} \left[\begin{array}{l} \text{LOCAL} \left[\begin{array}{l} \text{CATEGORY} \left[\begin{array}{l} \text{HEAD noun [CASE } \textit{nom}] \\ \text{SUBCAT} <> \end{array}\right]_{cat} \\ \text{CONTENT} \left[\begin{array}{l} \text{INDEX} \quad [1] \left[\begin{array}{l} \text{PER} \quad \textit{3rd} \\ \text{NUM} \quad \textit{sing} \\ \text{GEND} \quad \textit{fem} \end{array}\right]_{ref} \\ \text{RESTR} \quad \{\,\} \end{array}\right]_{pro} \\ \text{CONTEXT} \left[\text{BACKGR} \left\{ \left[\begin{array}{l} \text{RELN} \quad \textit{female} \\ \text{INST} \quad [1] \end{array}\right]_{psoa} \right\} \right]_{context} \end{array}\right]_{local} \end{array}\right]_{synsem} \end{array}\right]_{word}$$

Figure 2. Attribute Value Matrix for 'she'

The lexical entry for 'she' in Figure 2 is stated as an attribute value matrix (AVM). The words in capital letters are attributes (categories, features) and are followed by their value(s). Note that a value can be atomic or can be another AVM. All feature values are assigned a sort (or, type). The sort label specifies the attribute value type. The sort labels in the example above are in italics at the bottom left of the matrix. Sorts play an important role in HPSG. Sorts are themselves arranged into a hierarchy, and sort rules are defined to state which features are allowed with which sorts. From the perspective of sorts, an atomic value is a sort for which no features are specified [7:35]. Another aspect of HPSG that is seen in

Figure 2 is structure sharing. The person, number, and gender of the feature INDEX are shared by INST, hence the co-index '[1]'. Pollard and Sag [62:19] point out that in such structure sharing, pointing is made to the same structure. It is not simply the case of two separate structures that are identical in form. The authors state that structure sharing of the type used in HPSG is what is meant by the so-called 'unification' approaches to linguistic frameworks. Pollard and Sag also state that the central explanatory mechanism of HPSG is structure sharing (unification). Because DATR was designed to support lexical entries within the unification grammar tradition [32], the notion of unification will next be examined.

Many of the modern tools and theories of linguistics, especially the modern PSGs, either directly utilize the notion of unification, or have approaches that may be encoded by using unification. Two standard introductions to unification-based grammars are Shieber [64] and Kasper and Rounds [50]. The material presented here is drawn from Shieber. Shieber presents the formalism underlying many modern theories of syntax—namely unification—along with an abstraction from such theories. That abstraction is PATR II, a formalism that is neutral regarding any specific theory of syntax. Shieber describes PATR II as "…a powerful, simple, least common denominator of the various unification-based formalisms". PATR II is used by Shieber to illustrate the common elements of unification-based formalisms. However, PATR II has assumed a life of its own, with various implementations now available. PATR II is important to the discussion at hand for two reasons. First, SIL has recently modified PC-PATR, its implementation of PATR II, to incorporate AMPLE's morphotactics. Second, PATR II influenced the development of

DATR[38], since DATR was modeled after PATR II in its syntax and much of its semantics. And, as will be seen below, the complexity of lexicons in unification-based approaches was also a motivation for the development of DATR.

Shieber notes that unification-based, or, as they are sometimes called, complex-feature-based, grammars are based on research from both computational and formal linguistics, as well as research in knowledge representation and theorem proving[39]. Linguistic theories that are unification-based include Categorial grammar, Definite Clause Grammar, Functional Unification Grammar, GPSG, HPSG, and Lexical Functional Grammar. The degree to which unification is used in these approaches varies.

PATR II defines two basic operations—concatenation and unification. Concatenation is the sole string-combining operation, and unification is the sole information-combining operation. Shieber points out that because unification is the sole information-combining operation, statements are declarative in nature and order independent. Order independence is a point that will be returned to in the section on inheritance, below.

A grammar in PATR II contains two basic sections—a list of phrase structure rules, and a lexicon. PATR II phrase structure rules are context-free, and, unlike classical PSGs, have features associated with them to constrain the rules. Lexical entries take the form of feature structures, similar to the ones shown in (41) and in Figure 2. Shieber defines a feature structure as a partial function that maps features to values. Unification combines the information from two feature structures into a new feature structure. This is illustrated by (42).

---

[38] PATR is a formalism for syntax, whereas DATR is a formalism for lexicology.

[39] Shieber states that "unification was originally discussed as a component of the resolution procedure for automatic theorem-proving" [64:91].

(42)

(a) $\begin{bmatrix} cat: & NP \\ agreement: & [number: singular] \end{bmatrix}$

(b) $\begin{bmatrix} cat: & NP \\ agreement: & [person: third] \end{bmatrix}$

(c) $\begin{bmatrix} cat: & NP \\ agreement: & \begin{bmatrix} number: singular \\ person: third \end{bmatrix} \end{bmatrix}$

The feature structure (c) is the result of unifying (a) and (b). In unification, the information from two feature structures is combined into a new feature structure that contains all the information from the original two. Unification is said to *fail* if there is a conflict between information contained in two feature structures. For example, (43) (a) and (b), below, cannot unify because the values for number conflict:

(43)

(a) $\begin{bmatrix} cat: & NP \\ agreement & : & [number & : singular] \end{bmatrix}$

(b) $\begin{bmatrix} cat: & NP \\ agreement & : [number & : plural] \end{bmatrix}$

Unification is used as the mechanism to govern the applicability of phrase structure rules to lexical items. This is done by attempting to unify the feature constraints associated with a phrase structure rule with the features of lexical items.

Shieber illustrates the power of PATR II with several sample grammars. One linguistic phenomenon illustrated is that of agreement of person and number between a verb and the subject noun phrase:

(44)

(a) $S \rightarrow NP\ VP$
$< S\ head > = < VP\ head >$
$< S\ head\ subject > = < NP\ head >$

(b) $VP \rightarrow V$
$< VP\ head > = < V\ head >$

(c) $Uther \mapsto \begin{bmatrix} cat : & NP \\ head : & \begin{bmatrix} agreement : \begin{bmatrix} number : singular \\ person : third \end{bmatrix} \end{bmatrix} \end{bmatrix}$

(d) $sleeps \mapsto \begin{bmatrix} cat : & V \\ head : & \begin{bmatrix} form : & finite \\ subject : & \begin{bmatrix} agreement : \begin{bmatrix} number : singular \\ person : third \end{bmatrix} \end{bmatrix} \end{bmatrix} \end{bmatrix}$

(e) $sleep \mapsto \begin{bmatrix} cat : & V \\ head : & \begin{bmatrix} form : & finite \\ subject : & [agreement : [number : plural]] \end{bmatrix} \end{bmatrix}$

Note the feature structures associated with the lexical items (44) (a) and (b). These features state the agreement values for subjects. Also note the agreement values for (44) (c). When combined with the grammar rules (44) (a) and (b), the effect is agreement in person and number between the subject noun phrase and the verb of a sentence. The head features in lexical item (44) (c) and the head subject features in (44) (d) can unify with the rule constraints, which will properly yield the sentence *Uther sleeps*. However, the head features for (44) (e) and the head subject features for (44) cannot unify, correctly blocking **Uther sleep*[40].

An aspect of feature structures that should be noted is that they can be represented as directed acyclic graphs (DAGs). For example, (44) could be represented as shown in Figure 3. Note that feature names label the arcs and that the arcs point to either an atomic value or

[40] Traditionally in linguistics, an ill-formed expression is marked with an asterisk preceding it.

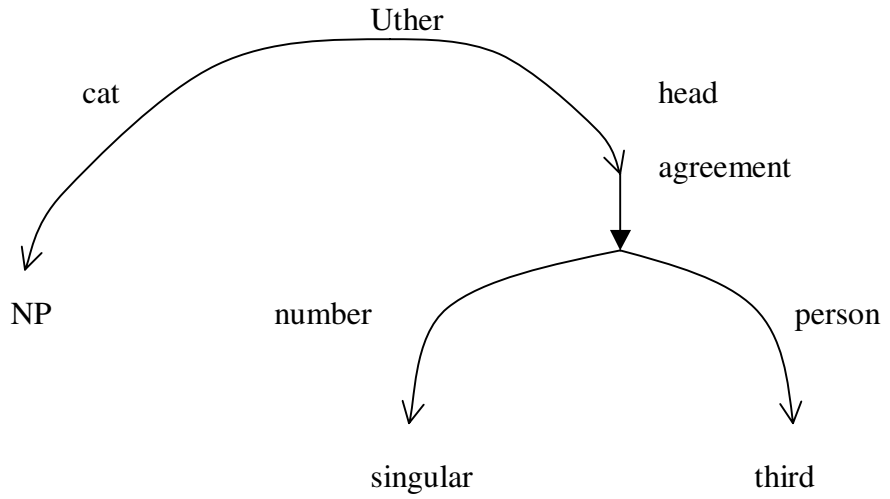

Figure 3. Directed Acyclic Graph for 'Uther'

to another DAG. Shieber notes that by viewing feature structures as graphs, the extensive research into graph theory can be applied to unification-based grammars, particularly in terms of graph theory vocabulary and operations.

The examples above provide a sense of how unification-based grammars simplify grammatical rules at the expense of increased complexity of lexical entries. In fact, the examples above are much simpler than what is truly required to fully handle the grammar of a natural language. Lexicalization resulted in a need to address both the management of the size and complexity[41] of lexicons and the ability to capture generalizations that hold across the features of lexical entries. A number of mechanisms were proposed and researched to address this need. The main two mechanisms are the use of (default and/or simple) inheritance hierarchies and lexical redundancy rules. Use of inheritance hierarchies to simplify management of lexicons and capture generalizations is noted by Shieber for PATR

[41] It should be noted that apart from increased complexity in the lexicon due to lexicalization, the inherent complexity of many natural languages also results in complexity in the lexicon.

II [64:54] and by Pollard and Sag for HPSG [62:36]. Inheritance hierarchies will be the subject of the next section.

This section has discussed the role of the lexicon in linguistic theory. The trend toward lexicalization was noted—that is, the simplification of grammatical rules at the expense of more complex lexical entries. The use of complex features was illustrated from HPSG, an example of a modern Phrase Structure Grammar that relies heavily on unification. The notion of unification was discussed because of the role it plays in the majority of modern approaches to syntax, and because DATR was explicitly developed to encode lexical entries for unification-based grammars. A brief overview of PATR II was provided to illustrate unification and how the use of complex features in lexical entries simplifies grammar rules at the expense of increased complexity in the lexicon. A further motivation for discussing PATR II is that DATR is patterned on the syntax of PATR and has a mostly common semantics with PATR.

2.3.2 Inheritance and the Lexicon

> In the interests of linguistic parsimony and sensible knowledge engineering, it is necessary for lexicalist approaches to factor away at the lexicon-encoding interface as many as possible of the commonalties between lexical items. [1].

As shown in the previous section, the trend toward lexicalization has increased the complexity of the lexicon. This has led to research into ways to manage that complexity. Many formalisms rely in varying degrees on inheritance as one of the mechanisms to manage the lexicon, e.g., HPSG [7:38], GPSG [62:36], and template inheritance in PATR II [64:58].

One of the earliest, and often referenced, expositions of the use of inheritance to organize the lexicon is Flickinger [39]. Flickinger presents a framework that includes a word class hierarchy, an inheritance mechanism that allows information from superclasses to flow

down to subclasses, and phrase structure rules that combine with lexical information to form phrase constituents. (The author also discusses lexical rules, another technique to manage complexity in the lexicon).

In 1992, the journal *Computational Linguistics* published two special issues on inheritance in NLP. In the first of the special issues, Daelemans et al. [25] provide an overview of the types of inheritance, its origins in three separate fields of study, its application to phonology, morphology, syntax, semantics, and pragmatics, and provide an extensive bibliography. The authors also note several trends occurring in the early 1990s. The following illustrations of the types of inheritance are from Daelemans et al.

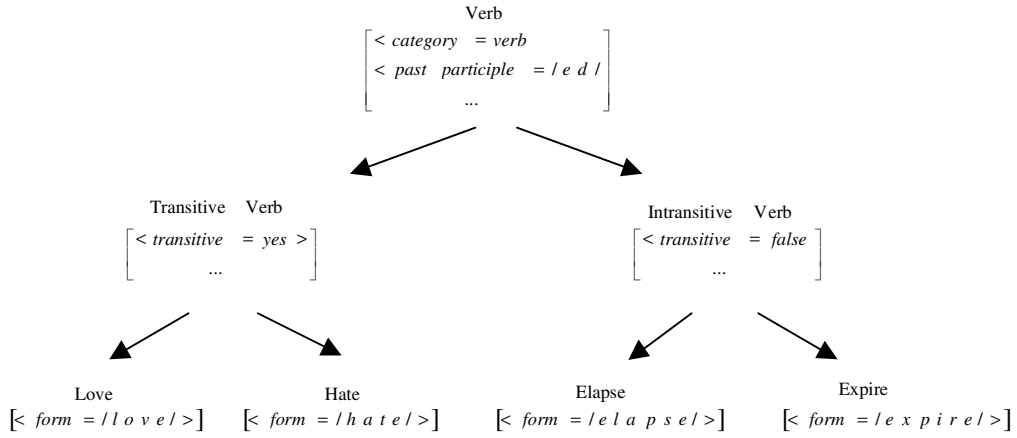

Figure 4. Monotonic Single Inheritance

Inheritance networks come in various forms. Variations typically depend on choices between single versus multiple inheritance, and monotonic versus non-monotonic inheritance. Figure 4 illustrates monotonic single inheritance. In single inheritance, a child node inherits properties from only one parent. In this case, for example, the verb node `Love` has a property for `<form>`, and inherits the `<transitive>` property and value from its parent,

the node `Transitive Verb`, which in turn inherits from the node `Verb`. Through inheritance, therefore, the node `Love` has properties and values for `<category = verb>`, `<past participle = /e d/>`, `<transitive = true>`, and `<form = /l o v e />`, as well as any other properties that might be indicated by the ellipsis (…). Because all properties of the parent are inherited, this type of inheritance is monotonic. With the given example, monotonic single inheritance works well. However, Daelemans et al. bring up the problem of verbs like *beat*. Whereas the past participle form of *love, hate, elapse*, and *expire* are all formed by adding *–ed*, verbs like *beat* form the past participle by adding *–en*. If the verb *beat* were added under the transitive node, it would inherit from it and, in turn, from the Verb node, to form *beated* instead of *beaten*. This means that the property `<past participle>` cannot be placed at the Verb node. Problems like this can lead to classifications where nodes high in the hierarchy only have a single property—an undesirable situation. Two alternative

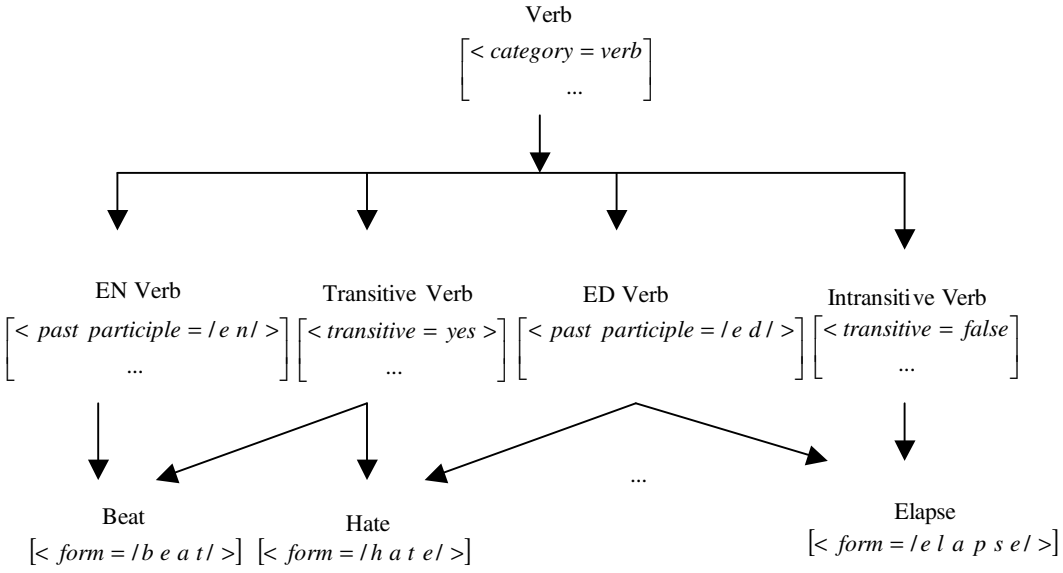

Figure 5. Monotonic Multiple Inheritance

solutions to this problem are to either use monotonic multiple inheritance, or to use non-monotonic single inheritance. A possible monotonic solution is shown in **Error! Reference source not found.**. In **Error! Reference source not found.**, multiple inheritance occurs. That is, properties may be inherited from more than one parent. In this scheme, verbs are simultaneously classified as to whether they are intransitive or transitive, and whether they take the *–en* or *–ed* affix to form the past participle. The node `Beat` inherits from both the node `EN Verb` and the node `Transitive Verb`. Thus, it will correctly take the *–en* suffix to form the past participle. An alternative to handle verbs such as *beat* is non-monotonic single inheritance as seen in Figure 6. In this solution, child nodes only inherit from a single parent. However, the generalization that most verbs in English add the *–ed* suffix to form the past participle can reside at the topmost node (for the Verb), but can be overridden for exceptions such as the node `Beat`. When the child properties take precedence over the properties of a parent, this is more commonly referred to as default inheritance.

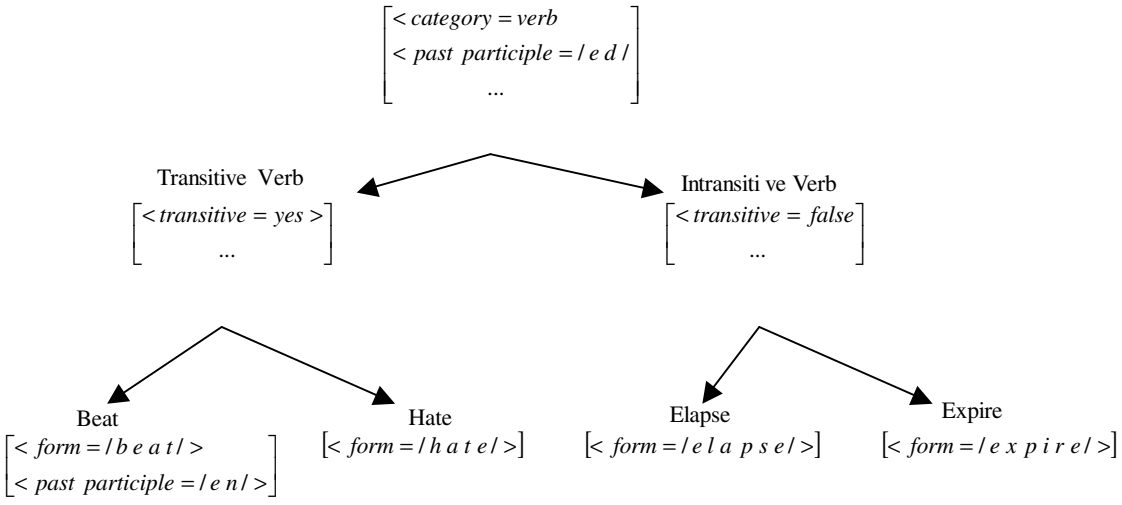

Figure 6. Non-Monotonic Single Inheritance

Daelemans et al. [25] also discuss a problem that arises in multiple inheritance when a child node inherits the same property from multiple parents, but with conflicting values for the property. The authors discuss two possible solutions to such a situation. One solution is orthogonal multiple inheritance. In this approach, parents of a child are not allowed to have the same properties. For example, in **Error! Reference source not found.** the `<past participle>` property came from one parent, the `<transitive>` property from another. This neatly side steps the problem. A second solution is prioritized multiple inheritance. In this approach, the first parent inherited from 'wins', and the child uses the property value from the winner[42].

Russel [63:147] points out that when unification is combined with default inheritance the pure, unordered nature of a unification formalism (such as PATR II) is compromised. Shieber [64:60] also makes this observation.

Having briefly explored the use of inheritance to manage a lexicon, what, then, are the advantages? Cahill [13] points out that in maintaining an NLP lexicon, it is advantageous to be able to make changes at a higher level in the hierarchy rather than at each leaf node, as would be the case with a lexical knowledge representation language that did not support inheritance. Daelemans et al. state that it is easier to make a change to only a few, high level nodes versus thousands in a non-inheritance-based lexicon. They also make the claim that "…inheritance lexicons can be made one to two orders of magnitude smaller than their full-entry counterparts" [25:214]. In addition to increased maintainability by reduction in redundancy, it has also been shown in the examples above how inheritance allows one to capture generalizations about classes of lexical items. While the use of classes, or categories,

---

[42] Though they do not say so, presumably a prioritized scheme could also specify that the last parent

has been a common tool even since the advent of American Structuralism, the advantage that inheritance adds is the ability to state in a single place the properties or features associated with a class, and have those properties be inherited by a member of the class. Furthermore, the use of default inheritance in particular is a powerful tool to allow linguists to capture generalizations (stated high in the hierarchy) as well as exceptions (overrides occurring low in the hierarchy). Default inheritance, therefore, avoids the needless proliferation of classes that occur in traditional approaches. Briscoe [8:9] notes that without a mechanism such as default inheritance, it is difficult to capture relationships that exist between irregular and regular classes.

Having discussed the general concepts of the use of inheritance in lexicons, attention will now be paid to the role of the lexicon in NLP, and then to a specific lexical knowledge representation language that uses inheritance.

2.3.3 The Role of the Lexicon in NLP

All languages, be they artificial or natural, have a vocabulary. Because natural language processing (NLP) systems deal with natural language, the need to have access to the vocabulary of the language is unavoidable. It is the lexicon that contains that vocabulary, be it a hard-coded list in a software program, a simple list of words in a file, or a linguistically sophisticated database of lexical information.

Gazdar and Mellish [43:217] state that lexicons have played an ever increasing role in NLP due to the increasing use of feature-based approaches to NLP (a reflection of developments in linguistic theory as discussed above). They state that a lexicon for an NLP

---

inherited from is the winner.

system must supply morphological, syntactic, and semantic information[43]. They also point out that in the case of highly inflected languages, the lexicon should list roots and affixes along with information needed to generate the surface forms of the words of a language. Gazdar and Mellish identify three basic types of syntactic information: part of speech, syntactic co-occurrence information, and properties relevant to syntax such as gender. The syntactic properties of a lexical item play an important role in parsing the syntax of a natural language string (e.g., a sentence). If the NLP system requires some form of natural language understanding, it is also necessary to provide semantic information in the lexicon.

For reasons of economy, the words of a language are usually ambiguous in that words have multiple meanings, and the required meaning of a word is selected by its context. Guthrie [45] discusses the role of the lexicon in disambiguating the meaning of natural language strings being processed by an NLP system.

Cater [19] states that the lexical information required for NLP varies with the task at hand. He explores the lexical information requirements from the perspective of the following tasks: single-sentence analysis, single-sentence generation, discourse-level processing, speech processing, text analysis, dialog participation, language transfer, and language acquisition. Like Gazdar and Mellish, Cater emphasizes the need for morphological information in the lexicon since it is usually not feasible to list every variant of a word.

However, although all NLP systems have some form of a lexicon, not all systems use lexicons that list morphemes, for example, stemming components of information retrieval systems. However, Hull [47] and Hull and Grefenstette [48] underscore the problems that arise with information retrieval stemming algorithms that do not utilize a lexicon with

---

[43] Presumably phonological information is grouped with morphological in their discussion.

morpheme meanings. Failure to take into account the meaning of morphemes can result in the conflation or merging of words with different meanings into the same normalized form, which can result in erroneous retrievals.

Except for the case of linguistically naïve, ad hoc approaches, NLP has its underpinnings in linguistic theory. The role of lexicons in NLP should, therefore, be understood from the perspective of the role of the lexicon in linguistic theories, as discussed above.

2.3.4 Lexical Knowledge Representation with DATR

Per Keller [53], the most commonly used lexical knowledge representation language in the NLP community is DATR. DATR was developed by Evans and Gazdar and has an extensive literature that includes [29, 30, 33-38, 52, 53]. The main introduction to DATR is [32]. Whereas some other languages (e.g., ACQUILEX [23]) are meant to be general-purpose knowledge representation languages, DATR is specifically a language for the representation of lexical knowledge. DATR is a language that utilizes semantic networks with non-monotonic (default) multiple inheritance. Nodes in DATR consist of sets of path/value equations. DATR uses orthogonal multiple inheritance to solve the problem of conflicts between properties inherited from multiple parents. It is, however, possible to encode prioritized multiple inheritance using DATR [36]. The stated objective for DATR is a language that:

> (i) has an explicit theory of inference, (ii) has an explicit declarative semantics, (iii) can be readily and efficiently implemented, (iv) has the necessary expressive power to encode the lexical entries presupposed by work in the unification grammar tradition, and (v) can express all the evident generalizations and subgeneralizations about such entries. [32].

Lexical generalizations that can be captured include those required for phonological, orthographic, morphological, morphophonological, syntactic, and semantic phenomena. The name and the syntax of DATR were modeled on PATR, though the semantics has subtle differences[44].

Evans and Gazdar [32] provide samples of code (called theorems in DATR) that illustrate the use of DATR. The following is a brief overview, using some examples from Evans and Gazdar. An extended example of the use of DATR with the Ogea language is provided in Appendix Three.

A non-inheritance type of representation for *love* in DATR might be:

```
(45)  Word1:
      <syn cat> = verb
      <syn type> = main
      <syn form> = present participle
      <mor form> = love ing.

      Word2:
      <syn cat> = verb
      <syn type> = main
      <syn form> = passive participle
      <mor form> = love ed.
```

In (45) and all other DATR examples, 'syn' stands for 'syntactic', 'cat' is 'category', 'mor' is 'morphological'. A theorem consists of a set of nodes in a lexical network. Each

[44] Although DATR was modeled on PATR, and although DATR is intended to support lexicons for unification-based grammars, DATR itself does not use unification (per Gerald Gazdar, personal communication).

node consists of a list of one or more path/value pairs[45]. The left-hand side of an equation contains the path, and the right-hand side contains the value. The theorem (45) states that the node `Word1` is a main verb, its syntactic form is a present participle, and its morphological form is 'love ing'. Similarly, the theorem states that `Word2` is a main verb, its syntactic form is a passive participle, and its morphological form is 'love ed'.

The problem with (45) is that it fails to capture generalizations. DATR can capture generalizations through its inheritance mechanism, as in the following theorem:

```
(46)  VERB:
      <syn cat> == verb
      <syn type> == main.

      Love:
      <> == VERB
      <mor root> == love.

      Word1:
      <> == Love
      <syn form> == present participle
      <mor form> == <mor root> ing.

      Word2:
      <> == Love
      <syn form> == passive participle
      <mor form> == <mor root> ed.
```

In DATR theorem (46), the node `Word1` inherits all paths found in the node `Love`. The empty path symbol `<>` indicates this. (Alternatively, the path symbol could have indicated a specific path in the node `Love`—more will be said about this below.) In turn, the node Love inherits from the node `VERB`, which provides values for the paths `<syn cat>` and

---

[45] Note that a DATR node can be expressed as an attribute value matrix:

$$Word1\begin{bmatrix} syn: \begin{bmatrix} cat: verb \\ type: main \\ form: present\ \ participle \end{bmatrix} \\ mor: [form: love\ ing] \end{bmatrix}$$

<syn type>. In addition to the paths inherited from Love and VERB, Word1 adds its own paths, <syn form> and <mor form>, and values for each path. The <mor root> value for the <mor form> path indicates that the <mor root> for Love is to be substituted, yielding 'love ed'.

Note that in (46), Word1:<mor form> == <mor root> ing. The right-hand of an equation can be a direct value, or, as in this case, a path name. When the right-hand side of an equation contains other than an explicit value, it is termed a *descriptor*. A descriptor uses either a node/path, node, or path instead of a direct value. The right-hand side of an equation does not have to be a single direct value or a single descriptor. It can be composed of multiple direct values and/or multiple descriptors.

Also note that in (45) a path is equated to a value using '=', whereas in (46) a path is equated to a value using '=='. The distinction in DATR between a single equality operator (=) and a double equality operator (==) is as follows. Statements that are implied via inheritance and therefore are not explicitly stated are termed *extensional*, and are designated by use of a single equality operator. Explicit statements, those that capture generalizations, are designated by use of a double equality operator, and are termed *definitional*. Note that because a definitional statement implies a corresponding extensional statement, a definitional statement may always be substituted for an extensional statement. In a separate key article, Evans and Gazdar [30] state that another way to view the difference between the extensional and definitional operators is the distinction between a query language and a database definition language. Extensional statements correspond to a query language, and definitional to a database definition language. Keller [52] also presents this viewpoint, and states that

extensional statements are treated by implementations of DATR as goal statements and are immediately evaluated whenever encountered.

Several syntactic shorthand devices are used in DATR. First, each equation has an implied `Node:<path> == value`. For example,

```
(47)  Word:
      <syn cat> == verb
      <syn type> == main
      <syn form> == passive participle
      <mor form> == love ed.
```

implies the following:

```
(48)  Word: <syn cat> == verb.
      Word: <syn type> == main.
      Word: <syn form> == passive participle.
      Word: <mor form> ==love ed.
```

Second, when a node/path is used as the value for the right-hand side of an equation, if the path is identical to the path in the left-hand side, it may be omitted. That is,

```
(49)  Word:
      <syn cat> == VERB:<syn cat>.
```

can be abbreviated as:

```
(50)  Word:
      <syn cat> == VERB.
```

This has to do with a concept in DATR known as *context*, to be discussed more fully below.

An important concept in DATR is default specification[46], also known as default definition or inference by default [29, 30]. In default specification, the left-hand portion of a

---

[46] Daelmans et al. [25] state that nonmonotonic inheritance is also known as default inheritance. In nonmonotonic inheritance, values inherited by a parent may be overridden by a child. Keller [52] states that the nonmonotonic character of DATR derives from DATR's default mechanism. Thus there is a relationship between default inheritance and default specification in DATR. In DATR, default specification allows all paths and values for a parent node to be inherited without having to explicitly specify each parent node/path from which to inherit. However, it would seem that the characteristic of a child's properties overriding those inherited from a parent has to do more with the use of local and global contexts in DATR than with inference by default.

query path[47] is used as a kind of wildcard to match against paths being searched in a node, effectively matching against far more paths than are actually specified. Thus far we have only seen query paths that take the value of an exactly matched path in a node. However, any query path whose left-hand portion matches a path in a node being searched, and whose right-hand portion extends or adds to the path in the node, will take on the value of the path in the node. This is known in DATR as default specification. Consider, for example, (51).

```
(51)  Do:
      <> == VERB
      <mor root> == do
      <mor past> == did
      <mor past participle> == done
      <mor present tense sing three> == does.
```

When a match for the query path `<mor past>` is sought in the node `Do`, it will exactly match `Do:<mor past>` and take on the value 'did'. This is the type of matching we have already seen. However, the query path `<mor past tense>` will also match `Do:<mor past>` and take on the value 'did'. This is because the left-hand side of `<mor past tense>` exactly matches `<mor past>`, and the right-hand side of `<mor past tense>` extends it. Another way to view default specification is that in a node being searched, if a path in the node is a substring of the left-hand side of the query path, the value will be used. Further examples are `<mor past tense plur>`, which extends `Do:<mor past>` and `<mor past participle plur>`, which extends `Do:<mor past participle>`. In the event that more than one path in a node is a substring of the query path (that is, the query path extends more than one path in the node), the longest path in the node wins. Because of default specification, there are effectively an infinite number of query paths that may match a path in a node. It should also be noted that all query paths extend the empty path <>.

---

[47] A query is an operation to find the value associated with a node/path.

```
(52)  VERB:
      <syn cat> == verb
      <syn type> == main
      <mor type> == root.

      Love:
      <> == VERB
      <mor root> == love.
```

Therefore, in (52), any query path failing to exactly match a path in the node `Love`, or failing to extend paths longer than `<>`, will take its value from the empty path `<>`. So, for example, the query `Love:<mor type>` has no exact match in `Love`, but it extends the empty path `<>`, and therefore takes on the value `VERB`, which causes the search to continue in the node `VERB`. On the other hand, `<mor root form>` extends both `Love:<>` and `Love:<mor root>`, but takes on the value 'love' because of the rule that the longest (most specific) path extended by the query path is the one that will be chosen.

One other aspect of default specification must be mentioned. In their article on inference in DATR [29], Evans and Gazdar state that when a path occurs in the right-hand side of an equation, a sub-path that extends the left-hand path also extends the right-hand side. As an example, Evans and Gazdar give `A2:<sing> == "A1:<plur>"`. The path `<sing fem nom>` would extend both sides, resulting in `A2:<sing fem nom> == "A1:<plur fem nom>"`. (However, in the DATR Standard Library RFC 2.0 [33], a "path cut" operator is defined, using a period. For example, if `A2:<sing> == "A1:<plur.>"`, then the path `<sing fem nom>` would result in `A2:<sing fem nom> == "A2:<plur>"`. The period (path cut operator) blocks the extension of the right-hand side.)

In DATR, when values are to be obtained by inheritance, the location of the inherited value may be indicated in one of three ways: a new path, a new node, or a new node/path. This is illustrated by Evans and Gazdar with the following:

```
(53)  Come:
      <> == VERB
      <mor root> == come
      <mor past> == came
      <mor past participle> == <mor root>
      <syn> == INTRANSITIVE:<>.
```

In (53), the right-hand side of an equation can indicate a new path from which to inherit (e.g., `Come: <mor past participle> == <mor root>`), a new node from which to inherit (e.g., `Come: <> == VERB`), or a new node/path from which to inherit (e.g., `Come: <syn> == INTRANSITIVE: <>`).

There are two types of inheritance in DATR: local inheritance and global inheritance. All examples seen thus far are instances of local inheritance. Global inheritance is indicated by quotation marks around a path. Both local and global inheritance utilize the notion of context. In DATR, the context stores a node name and a path. A context is saved for both local and global inheritance. When a query is put to DATR, the original query node and path are saved to both the local and global context. The local context is used for local inheritance, and the global context is used for global inheritance.

```
(54)  VERB:
      <mor past> == "<mor root>" ed

      Laugh:
      <> == VERB
      <mor root> == laugh.
```

Consider the query Laugh:<mor past>. When the query is given to DATR, the global context has the following values: Node: `Laugh`, Path: `<mor past>`. Initially, the local context is set identical to the global. In (54), the only path extended by `<mor past>` is the empty node `<>`. The value obtained is a node name, namely `VERB`. The new node name is copied to the local context, updating the local context Node to be `VERB`. Having updated the local context, the new local context is used for the search (Node = `VERB`, Path = `<mor past>`). An exact match is found in `VERB`. However, when DATR evaluates the path `VERB:<mor past>`, it encounters

"<mor root>" ed. The quotation marks around <mor root> tell DATR to use the global context instead of the local context. Before encountering the quoted path, the global context contains Node = Laugh, Path = <mor past>. The quoted path, "<mor root>" results in a new value for the path variable in the global context. The global context now becomes Node = Laugh, Path = <mor root>. Updating the global context also results in a change to the local context. The contents of the global context are copied to the local context, and control is now passed back to the context specified by the local context. DATR will now search the node Laugh, looking for a match for the path <mor root>. In Laugh, it finds that <mor root> == laugh. This value is returned back up to the node VERB to complete the valuation of VERB:<mor past>. Therefore, the value for the node/path VERB: <mor past> becomes laugh ed.

One way of looking at the global context is to think of it as global memory that contains a record of where we originally came from. It must be kept in mind that as each expression in the right-hand side of an equation is evaluated, the local context is updated. However, unless a quoted path is encountered, the global context remains unchanged. When a quoted path is evaluated, both the local and the global context are changed. Table 2 summarizes how local and global contexts are conceptually updated.

Table 2. How Local and Global Contexts are Updated in DATR

Descriptor Contents	Global Node Value	Global Path Value	Local Node Value	Local Path Value
Node2	Unchanged	Unchanged	Set to Node2	Unchanged
<Path2>	Unchanged	Unchanged	Unchanged	Set to Path2
Node2:<Path2>	Unchanged	Unchanged	Set to Node2	Set to Path2
"Node2"	Set to Node2	Unchanged	Set to Global Node	Set to Global Path
"<Path2>"	Unchanged	Set to Path2	Set to Global Node	Set to Global Path
"Node2:<Path2>"	Set to Node2	Set to Path2	Set to Global Node	Set to Global Path

So far, the examples provided have stated morphological variants at the root level of lexical entries. They fail to capture the generalization of morphological forms at higher levels. By use of a combination of both local and global inheritance, this can be achieved as seen in (55):

(55)
```
VERB:
<syn cat> == verb
<syn type> == main
<mor form> == "<mor "<syn form>">"
<mor past> == "<mor root>" ed
<mor passive> == "<mor past>"
<mor present> == "<mor root>"
<mor present participle> == "<mor root>" ing
<mor present tense sing three> == "<mor root>" s.

Love:
<> ==VERB
<mor root> == love.

Word1:
<> ==Love
<syn form> == present participle.

Word2:
<> ==Love
<syn form> == passive participle.
```

```
(56)  =0,0,0> LOCAL Word1:< || mor form > == Love
(57)          GLOBAL Word1:< mor form >
(58)  =1,0,0> LOCAL Love:< || mor form > == VERB
(59)          GLOBAL Word1:< mor form >
(60)  =2,0,0> LOCAL VERB:< mor form > == "< mor "< syn form >" >"
(61)          GLOBAL Word1:< mor form >
(62)  =3,1,0> LOCAL Word1:< syn form > == present participle
(63)          GLOBAL Word1:< syn form >
(64)  =3,0,0> LOCAL Word1:< || mor present participle > == Love
(65)          GLOBAL Word1:< mor present participle >
(66)  =4,0,0> LOCAL Love:< || mor present participle > == VERB
(67)          GLOBAL Word1:< mor present participle >
(68)  =5,0,0> LOCAL VERB:< mor present participle > ==
                                  "< mor root >" ing
(69)          GLOBAL Word1:< mor present participle >
(70)  =6,0,0> LOCAL Word1:< || mor root > == Love
(71)          GLOBAL Word1:< mor root >
(72)  =7,0,0> LOCAL Love:< mor root > == love
(73)          GLOBAL Word1:< mor root >
(74)  [Query 1 (12 Inferences)] Word1:< mor form > = love ing .
```

At this point, this author will digress from direct discussion of Evans and Gazdar [32] and provide some additional explanation of how DATR works. A trace from processing the query `Word1:<mor form>` against DATR theorem (55), using a PC implementation of DATR called ZDATR[48] [44] is shown in (56)-(74). The trace shows both the local and global context at each step. Initially the query starts at the node `Word1`, looking for a path named `<mor form>`. No exact match is found, but `<mor form>` extends the empty path `<>`, so the value will be obtained from the `<>` path. This can be seen in (56). The || symbol indicates the point at which the query path extends a node path. In (58), note that the local context has now been updated using the node name specified by `Word1:<> == Love`. Also note that the global context has not changed. As before, the query (in this case, `Love:<mor form>`) is not matched, but since it extends the empty path in the node `Love`, the local context is updated by changing the value for node to equal `VERB`. In (60), we have now been directed to `VERB:<mor from> == "mor "<syn form>">"`. Note the quoted paths. Also, note

something new: a path (`"<syn form>"`) embedded within a path. Embedded paths are called *evaluable paths* in DATR terminology. When a path itself contains another path to be evaluated, evaluation occurs inside-out. Thus, in this case, evaluation starts with the global path `"<syn form>"`. Note in (62) that the global path has been changed to `<syn form>`, and the global context has been copied to the local context. ZDATR is now attempting to satisfy the query by looking at `Word1:<syn form>`. The value of `Word1:<syn form>` is *present participle*. Note in (64) that the local context is the node `Word1`, and that the path value has prefixed *mor* to *present participle*, yielding `<mor present participle>`. How did *mor* get prefixed to the value obtained from `Word1:<syn form>`? And, how did the local context stay at `Word1`? The answer to both questions lies back up in `VERB:<mor form>`. The value there is `"<mor "<syn form>">"`. The inner-most path was evaluated first, i.e., `"<syn form>"`, and the value obtained from `Word1:<syn form>` (that is, *present participle*) was plugged into `"<mor "<syn form>">"`, replacing `"<syn form>"` with *present participle*. Effectively, then the value for `VERB:<mor form>` became `"<mor present participle>"`. In this case, path extensions are carried over from the left-hand side of the query path to the right-hand side. Why did the context stay at `Word1`? Because the new descriptor was itself quoted (just like the inner path was), the context was switched back again to that specified by the global context, namely, to the node `Word1`. Evans and Gazdar do not discuss what would have happened if the outer path had not been quoted (i.e., if the descriptor was `<mor "<syn form>">` without outer quotes). However, if the outer path had not been quoted, the node for the local context would have switched back to `VERB`. This presumably would be undesirable for the following reason. Default inheritance allows children to override properties inherited

---

[48] ZDATR is discussed in a section below.

from parents. By quoting the entire value for VERB:<mor form>, the node Word1 becomes the starting point to find the value for <mor present participle>. If that path existed in Word1, the value would be obtained from the node Word1, overriding the value inherited from VERB. Or, instead, if the path existed in Love, the value would be obtained from the node Love, overriding the value inherited from VERB. If any child of VERB specified the path being searched for, failure to quote the entire value specified by VERB:<mor form> would effectively short-circuit the ability of children to override properties inherited from parents.

The DATR theorem shown in (55) generalizes the lexicon to cover the past and present tenses of all English regular verbs and the three participle forms. This example clearly shows the power of inheritance to capture generalizations and to reduce redundancy. Without an inheritance mechanism, each leaf node (e.g., Word1 and Word2) would have an additional nine path/value equations. The use of inheritance significantly reduced the size of leaf nodes in this case. Also, the notion of default specification significantly reduces the number of path/values that must be specified.

So far, the examples shown have only dealt with simple inheritance and regular verbs. Evans and Gazdar [32] provide examples of how DATR can handle subregular and irregular verbs. Verbs such as *mow* and *sew* have *mown* and *sewn* for the past participle. By use of default inheritance, where an inherited property is overridden by the child node, it is possible to correctly handle the so-called EN-verbs:

```
(75)  VERB:
      <syn cat> == verb
      <syn type> == main
      <mor form> == "<mor "<syn form>">"
      <mor past> == "<mor root>" ed
      <mor passive> == "<mor past>"
      <mor present> == "<mor root>"
      <mor present participle> == "<mor root>" ing
      <mor present tense sing three> == "<mor root>" s.
```

```
EN_VERB:
<> == VERB
<mor past participle> == "<mor root>" en.

Mow:
<> == EN_VERB
<mor root> == mow.

Sew:
<> == EN_VERB
<mor root> == sew.
```

In (75), the path in `EN_VERB: <mor past participle>` overrides the `<mor past>` that would be inherited from the node `VERB` (because of default specification). Instead, the value for `EN_VERB: <mor past participle>` is evaluated from the global context (e.g., `Mew: <mor root>`), yielding *mew en.* (This underlying form would be converted to the appropriate orthographic form *mown* via spelling rules).

The verb *do* is used by Evans and Gazdar as one example of how DATR handles irregularity:

(76)
```
Do:
<> == VERB
<mor root> == do
<mor past> == did
<mor past participle> == done
<mor present tense sing three> == does.
```

Assuming that an abstract node, e.g., `VERB`, specifies values that apply to regular verbs, then by use of default inheritance, the values in the node `Do` would override the values that would otherwise be inherited from the node `VERB`. In this way, irregularity can be handled in DATR.

Multiple inheritance can be coded in DATR by referencing multiple nodes from which to inherit as seen in (77), an abstract example:

(77)
```
A:
<x y> == z.

B:
<h> == I
```

```
C:
<> == A
<> == B.
```

In (77), the node C inherits from both nodes A and B. As noted earlier, DATR provides built-in support for orthogonal multiple inheritance. This means that the properties inherited from multiple parents must be disjoint, thus avoiding conflicts. Evans and Gazdar state that orthogonality is enforced via DATR's syntactic notion of functionality [32]. They formally define this notion as follows:

> A DATR description is **functional** if and only if (i) it contains only definitional statements and (ii) those statements constitute a (partial) function from node/path pairs to descriptor sequences.

The practical consequence of functionality in DATR[49] is that a syntax error will be generated if a specific node/path pair occurs more than once. For example, the following would produce a syntax error:

```
(78)  Love:
      <mor root> == love
      <mor root> == love.
```

Although an individual developing a lexicon with DATR might not directly write a statement such as (78), through multiple inheritance a node could inherit <mor root> from multiple parents, with a result that has the effect of (78). However, the notion of functionality would generate an error, thereby enforcing orthogonal multiple inheritance.

Although DATR was designed to be a lexical knowledge representation language, it has turned out to be powerful enough to encode a variety of techniques. Evans and Gazdar describe how to code case statements, Boolean logic, finite state transducers, lists, lexical rules, ambiguity, alternation, and directed acyclic graphs (DAGs).

---

[49] Also, it should be noted that one would seldom, if ever, write extensional statements in DATR.

It should be noted that DATR is linguistic theory neutral, and theoretically capable of encoding a variety of approaches to linguistic phenomena. Indeed, Evans and Gazdar note that DATR has been used to encode lexicons for GPSG, PATR, and other approaches [32]. Although intended for unification-based approaches, there is no reason to think that DATR could not be used to encode lexical information required for a morphological parser such as AMPLE. Evans and Gazdar explicitly suggest that DATR can be used for item-and-arrangement approaches to morphology, the approach used by AMPLE.

2.3.5 The DATR Literature

Having introduced many of the main concepts of DATR drawing from Evans and Gazdar [32], a survey will now be provided of some of the other important DATR literature. Early work on DATR was brought together as a single volume in *The DATR Papers* [31]. This volume contains seven papers, DATR lexicon samples for nine natural languages, eighteen pieces of DATR code that illustrate various techniques, and source code for a Prolog implementation of DATR. Two major web-sites provide information on DATR, one at the University of Sussex (http://www.cogs.susx.ac.uk/lab/nlp/datr/datr.html), and the other at Leuven University (http://www.ccl.kuleuven.ac.be/LKR/html/datr.html). The web-sites provide extensive bibliographies of the DATR-related literature, as well as a library of DATR code fragments that illustrate the use of DATR with a variety of languages and for a variety of purposes.

2.3.5.1 DATR Semantics and Inference

The semantics of DATR was initially described by Evans and Gazdar in [30] and DATR inference was described by the same authors in [29]. The initial description of the semantics of DATR was deemed inadequate, and a better coverage was offered by Keller

[52]. In his improved coverage of inference in DATR [53], Keller states that in both his paper and Evans and Gazdar's paper on inference in DATR, the goal was to define rules that allow one to determine what may be legally inferred from DATR theories.

In their paper on inference [29], Evans and Gazdar define seven rules used for inference in DATR. Rule I states that for every definitional statement there is a corresponding extensional sentence that may be inferred. Rules II – IV cover local inheritance. Rule II states that if a given node/path is associated with another node/path (on the right-hand side of the equation), the value obtained resulting from an evaluation of the node/path may be substituted for the right-hand node/path, and another evaluation may be made if necessary. Rule III does the same for a node specified on the right-hand side, and Rule IV for a path. Rules V-VII also establish substitution of values for a node/path, a node, or a path, but do so for a global context. In the last section of their paper on inference, Evans and Gazdar cover inference by default. Evans and Gazdar state that their notion of default inference was derived from Moore [59, 60]. Default inference (specification or definition) was explained above. However, it must be noted that whereas inference by Rules II – VI allow a value to be inherited from a specific node/path, node, or path, inference by default allows all values from another node to be inherited unless overridden by a more specific local path. If it were not for inference by default, every node/path to inherit from would have to be explicitly stated.

Keller [52] sought to rectify recognized shortcomings of Evans and Gazdar's formal description of the semantics of DATR [30]. Evans and Gazdar's description focused on local and global inheritance and default inference. They did not, however, adequately cover the notion of global context, list values, or evaluable paths. Furthermore, Keller points out that

there are constructs available in DATR that do not fit the semantic network viewpoint of DATR. The semantic network paradigm works well for viewing the organization of lexical information encoded by DATR, but is not adequate to fully describe the language itself. Instead, Keller seeks to present DATR "as a language for defining certain kinds of partial functions by cases." From this perspective, the theories encoded in DATR are viewed as sets of partial functions that map paths onto values. In the case of the semantics of inference by default, the mapping is from paths "…to functions from extensions of those paths to values."

As an improvement on the inference paper by Evans and Gazdar [29], Keller [53] presents "the first fully worked out, formal system of inference for DATR theories." In his paper, Keller formulates an evaluation semantics for DATR that contains a complete set of inference rules covering values, definitions, sequences, evaluable paths, quoted descriptors, and path extensions.

2.3.5.2 Prioritized Multiple Inheritance

In another paper by Evans and Gazdar [36], they demonstrate three techniques to encode prioritized multiple inheritance in DATR. However, the authors argue against the advisability of abandoning orthogonal multiple inheritance for prioritized multiple inheritance.

2.3.5.3 Morphophonology

In [12], Cahill describes a new approach to previous work she did on English verbal morphology. In her previous work, Cahill relied on a two-tiered approach that combined DATR and the language MOLUSC (used for defining morphological alternations), but could not make use of DATR's inheritance mechanism. In her new approach, Cahill gains access

to DATR's inheritance mechanism and shows how a single-tiered (DATR only) approach may be used, though still relying on the theory of MOLUSC. She sets forth her new approach as a demonstration that a single lexicon can integrate all levels of description--orthography, phonology, morphology, syntax, and semantics. Cahill's approach to morphological alternation is based on mappings between sequences of syllable-based tree structures. Each syllable has a hierarchical structure containing an onset and a rhyme. The rhyme is composed of a peak and a coda. Whereas in traditional segmental phonology features are defined for each segment, Cahill makes use of an approach that assigns features at levels in the hierarchy that are above the segmental level. Specifically, features can be assigned at the syllable level, the rhyme level, the onset level, the peak level, and the coda level. Features at higher levels are inherited by lower levels, and lower levels may override a feature inherited from a higher level. Each feature consists of a name, a value, and a timing (scope of applicability). For example, in the feature [+ voice 1-2], the feature name is *voice*, the value is "+" (on), and the timing is segments 1-2. Although in Cahill's approach features are associated with syllables, rhymes, onsets, peaks, and codas rather than with segments, note that the timing of a feature is described in terms of segment timing points.

```
(79)  Spell:
      <> VERB
      <rhyme_feats> == ([ + voice 2-4])
      <onset> == ([ - voice 0-2 ]
                  [ + sibilant 0-1 ]
                  [ + alveolar 0-1]
                  [ + stop 1-2 ]
                  [ + labial 1-2 ])
       <peak> == ([ - round 2-3 ]
                  [ - high 2-3 ]
                  [ - low 2-3 ]
                  [ + front 2-3 ])
       <coda> ==  ([ + lateral 3-4 ]).
```

Take for example, the DATR code fragment for *spell* (79) from [12]. The word *spell* consists of four segments, 's', 'p', 'e', and 'l'. The onset contains the first two segments, the peak contains the third segment, and the coda contains the fourth segment. The rhyme, therefore, contains the third and fourth segments. Although not shown in this example, the hierarchical relationship between syllable, rhyme, onset, peak, and coda is defined in the node `VERB`. In this example, rhyme has a feature named *voice* that has a value of '+' (on) and a timing of 2-4. This means that for the duration of timing points 2-4, the feature *voice* is turned on. Because *voice* is defined at the rhyme level, the peak, and coda inherit the feature. However, note that the onset contains a feature named *voice* that has a value of '-', and a timing of 0-2. This means that for the duration of timing points 0-2[50], the feature *voice* is turned off.

Example (79) is not the final form of entries in Cahill's approach, but she uses this intermediate form as a convenient way to illustrate the use of syllable structures and features. Feature names are actually stated at a higher level (e.g., at an abstract node `VERB`), and feature values and timings are stated at a lower level. Also, a default value ('-' for 'off') and a default timing (the whole length of a verb root) are defined so that leaf nodes state values and timings only for those features they actually use. This is shown in (80), which provides only that part of Cahill's example that is relevant to this author's discussion. (Comments, '%', were added by this author). Example (81) shows the final form developed by Cahill for a leaf-level lexical entry. Note in (81) that values and timings are split into separate sets that must be considered together. For example, `<val lab onset> == +` "turns on" the labial feature, but `<time lab onset>` specifies that it is only on during timing points 1-2.

---

[50] An interpretation of this is that segments are related to timing points. Timing starts at point zero in time. The first segment therefore occurs from points 0 to 1 in time.

Cahill also covers context-dependent morphological alternation and feature value alternation. For example, (82) would handle the case of alternations for words such as *bereave* versus *bereft* and *cleave* versus *cleft*, where the peak is {ii} for the present tense form, and {e} for the past tense form.

```
(80)  VERB:
      <> == ()
      <root> == <struct "<sylls>"
      <sylls> == ()
      <struct pref> == ("<syll pref>" <struct>)
      <struct> == <syll>
      <syll> == ([ <feats syll> ]              % features
                 [ <feats onset> ]
                 [ <rhyme> ])

      <rhyme> == ([ <feats rhyme> ]
                  [ <feats peak> ]
                  [feats coda> ])
      <feats> ==
            ([alv "<val alv>" "<time alv>"         % alveolar
              approx  "<val approx>" "<time approx>"   % approximant
              fric "<val fric>"   "<time fric>"        % fricative
              high "<val high>"   "<time high>"       % high
              lab  "<val lab>"    "<time lab>"        % labial
              lat  "<val lat>"    "<time lat>"        % lateral
              low  "<val low>"    "<time low>"        % low
              nasal "<val nasal>"  "<time nasal>"     % nasal
              round "<val round>"  "<time round>"     % round
              sib   "<val sib>"    "<time sib>"       % sibilant
              stop  "<val stop>"   "<time stop>"      % stop
              vel   "<val vel>"    "<time vel>"       % velar
              voice "<val voice>"  "<time voice>")    % voice
      <val> == -
      <time> == r1.                        % rl = root length[51]


(81)  Spell:
      <> == VERB_A
      <val sib onset> == +
      <val lab onset> == +
      <val stop onset> == +
      <val front peak> == +
      <val voice peak> == +
      <val lat coda> == +
```

[51] This is a default timing value. If no timing value is specified elsewhere, the timing defaults to the length of the entire root.

```
<val voice coda> == +
<time sib onset> == 0-1
<time lab onset> == 1-2
<time stop onset> == 1-2
<time front peak> == 2-3
<time voice peak> == 2-3
<time lat coda> == 3-4
<time voice coda> == 3-4.
```

```
(82)   <peak past> == <peak_change "<peak pres>">
       <peak_change ii> == e
       <peak_change> == "<peak pres>"
```

The example code also handles cases of alternate forms where the peak for past tense is {e} if the peak for present tense is {ii}, and past tense is {e} otherwise. Cahill gives several examples of feature value-based alternations. The example of the English plural morpheme given in Chapter One is repeated here to illustrate Cahill's approach to feature-based alternation.

```
(83)  bets        {-s}
(84)  beds        {-z}
(85)  places      {-  z}
```

Cahill provides (86) as a set of traditional feature-based, ordered rules that cover (83) - (85).

```
(86)  S → /  z/ / [+ sib] ___
      S → /s/ / [- voice] ___
      S → /z/ / [+ voice] ___
```

That is, the English plural morpheme is realized as {- z } following a root that ends in a sibilant, {-s} following a root that ends in an unvoiced non-sibilant, and {-z} following a root that ends in a voiced non-sibilant. The rules for plural nouns in English shown in (86) could be covered by the following DATR code developed by this author using code fragments from Cahill [12] for the node NOUN:

```
(87)  NOUN:
      <noun plural> == ("<root>" <ssuff  "<val sib coda>"
                        "<val voice coda>">)
```

```
<ssuff +> == iz
<ssuff - +> == z
<ssuff -> == s.

Word1:
<> == NOUN
<root> == bet
<val sib coda> == -
<val voice coda> == -.

Word2:
<> == NOUN
<root> == bed
<val sib coda> == -
<val voice coda> == +.

Word3:
<> == NOUN
<root> == place
<val sib coda> == +
<val voice coda> == -.
```

This example relies on default specification. If `<val sib coda>` is '+', any value of `<val voice coda>` will extend the path for `NOUN:<ssuff +>`, setting the value to 'iz'. If `<val sib coda>` is '-', then if `<val voice coda>` is '+', a precise match will occur with `<ssuff - +>`, setting the value to 'z'. If `<val sib coda>` is '-', and `<val voice coda>` is '-', this extends the path `<ssuff - >`, setting the value to 'z'.

Before leaving this discussion of Cahill's work on morphophonology [12], one more technique will be covered. Cahill points out that particularly in the case of roots, it is possible that a morpheme will have multiple syllables. It is therefore necessary to have a technique that allows any number of syllables to be specified for a morpheme in the DATR code. In her approach, each root entry contains a path with a value representing each syllable beyond the first. The number of value occurrences matches the number of syllables beyond the first. So, for example, a disyllabic syllable would have `<struct> == ext`, and a trisyllabic syllable would have `<struct> == ext ext`, where *struct* is an arbitrary path name, here meaning *structure*, and *ext* is an arbitrary name meaning *extension*. Each syllable

beyond the first is an extension, and the number of *ext* occurrences assigned to the path *struct* matches the number of syllables occurring in addition to the first syllable.

```
(88)  VERB:
      <root> == <struct "<sylls>">
      <struct ext> == (<struct> "<syll ext>")
      <struct == <syll>
      <syll> == ("<onset>" "<rhyme>")

      Root1:
      <struct> == ext ext.
```

In (88), a subset of the paths for the node `VERB` is shown, with the paths taken from an example from Cahill. In combination with the `<struct>` path and values from the node `Root1`, the DATR code for `VERB` is sufficient to handle any number of syllables. The reader is referred to [12] for an explanation of how the code works, but suffice it to say that (88) relies heavily on the use of default specification, and particularly on the fact that in DATR, extensions of paths on the left-hand side of a DATR equation are added to the right-hand side also. A query of `Root1:<root>` would result in `VERB:<root> == <struct "<sylls>">` evaluating to `VERB:<root> == ("<onset>" "<rhyme>" "<onset ext>" "<rhyme ext>" "<onset ext ext>" "<rhyme ext ext>")`.

2.3.5.4 DATR Techniques for Phonology-based Lexicons

Cahill et al. [16] present a tutorial on the representation of phonology-based lexical knowledge using DATR. In this context, phonology based lexicons are lexicons that use non-orthographic (i.e., phonetic or phonological) representations for words. The authors note that most NLP research focuses on orthographic representations. Topics covered by Cahill et al. include segmental phonology, inflectional phonology, morphophonology, nonsegmental phonology, and lexica for speech. The section on lexica for speech covers linguistic word recognition, delayed synchronization, and multi-tape finite state transducers (FSTs). Cahill

et al. provide DATR techniques for each area of discussion. Only a few of the techniques will be discussed here.

As in [12], Cahill et al. present a technique to handle polysyllabic words. However, the technique is slightly different. Based on (89), the query result for `Sew:<phn root form>` would be "s @ U ", and `Zeitung:<phn root form>` would be "t s a I t U N'. The main difference between this technique and the one given by Cahill in [12] is that Cahill et al. use abstract nodes for syllable, disyllable, and trisyllable.

Cahill et al. also present several techniques that rely on DATR's built in boolean operators. Because these operators have not been demonstrated thus far in our discussion of DATR, an example will now be provided from [16].

(89)
```
# vars $yll: syll syl2 syl3.

Syllable:
<> == Null
<phn root> == <phn syl1>
<phn $yll form> == "<phn $yll onset>"
                   "<phn $yll peak>"
                   "<phn $yll coda>".
Disyllable:
<> == Syllable
<phn root> == <phn syl2> <phn syl1>[52].

Trisyllable:
<> == Syllable
<phn root> == <phn syl3> <phn syl2> <phn syl1>.

Sew:
<> == Syllable
<phn syl1 onset> == s
<phn syl1 peak> == @ U[53].
```

[52] Because the languages used for the tutorial (English and German) are primarily suffixing languages, reference is typically made to the last syllable of a root. Therefore the authors chose to number the syllables of roots in reverse order, with syl1 used for the last syllable.

[53] The authors use SAMPA, a machine-readable phonetic alphabet. See www.phon.ucl.ac.uk/home/sampa/home.

```
Zeitung:
<> == Disyllable
<phn syl2 onset> == t s
<phn syl2 peak> == a I
<phn syl1 onset> == t
<phn syl1 peak> == U
<phn syl1 coda> == N.

(90)  Suffix_ED:
       <> == Affix
       <phn root form> == IF:<ALVEOLAR:<FINAL_SET:<Root>>
                          THEN I d
                          ELSE IF:<VOICED:<FINAL_SEG:<Root>>
                               THEN d
                               ELSE t >>.
```

In (90), a rule is provided for the addition of the suffix –ed to roots. If the final segment of the stem is an alveolar consonant, then the suffix will be realized as {I d}[54]. Otherwise, if the final segment is voiced, the suffix will be realized as {d}. If it is not voiced, it will be realized as {t}.

2.3.5.5 DATR Encoding of LTAG Lexicons

The use of DATR to encode trees of a Lexicalized Tree Adjoining Grammar (LTAG) is described in two articles by Evans, Gazdar, and Weir [37, 38]. An LTAG specific LKRL was proposed by Vijay-Shanker and Schabes [69] to tackle the problem of redundancy in lexicons for LTAG, and was based on Flickinger's work on inheritance in the lexicon [39]. However, in [37], Evans et al. propose using DATR for the LKRL for LTAG lexicons, rather than developing a new LKRL tailored specifically for the needs of LTAG. Evans et al. argue the advantages of using an "off-the-shelf" LKRL (specifically DATR) and demonstrate how DATR can encode LTAG lexicons. Whereas Vijay-Shanker and Schabes utilize subcategorization trees with dominance, immediate dominance and linear precedence statements, Evans et al. use only local relations to specify the tree features of each node. For

the work presented in [37], Evans et al. investigated how four lexical rules would be handled with their DATR encoding of an LTAG lexicon: passive, dative, subject-auxiliary inversion, and wh-questions.

In [38], Evans et al. present an expanded and refined version their earlier work [37]. As before, they note the advantage of using a well established LKRL (DATR) for encoding LTAG lexicons, but also note that LTAG presents interesting problems for DATR, since grammars in LTAG are totally lexically based. In their DATR-based approach, the authors describe a tree relative to an anchor node (an LTAG leaf-node labeled with a lexical category). Each anchor node is described by relating it to its parent and sister subtrees, using as features *parent*, *left*, and *right*. Therefore, tree structures are embedded as features in DATR nodes. Furthermore, the tree structure encodings are described bottom-up (remembering that the anchor node is a leaf (bottom) node). For each anchor node, there is a DATR node path equation for the parent, the parent's parent, the node left or right of the parent, and any nodes to the left or right of the anchor node. Evans et al. note that encoding more than merely parent and sibling nodes allows one to make generalizations not otherwise possible.

In addition to the four LTAG lexical rules researched for their original work (passive, dative, subject-auxiliary inversion, and wh-questions), in [38] the authors also report their work on relative clauses and topicalization. In both LTAG, and in Evans et al.'s encoding of LTAG nodes in DATR, lexical rules are stated in terms of an input and output tree. In the DATR encodings, the input and output paths that represent these trees are linked together via DATR's inheritance mechanism.

---

[54] Note again, that the authors are using SAMPA for the phonetic alphabet.

In summary, Evans et al. demonstrate not only the applicability of DATR to encoding lexicons for LTAG grammars, but also demonstrate the power of DATR in general to capture lexical rules and generalizations for fully lexicalized grammars. They present a number of techniques that could be useful elsewhere, such as tree transformations during mappings from input to output trees.

2.3.5.6 Multilingual Lexicons using DATR

The use of DATR for encoding multilingual lexicons is described in three articles by Cahill and Gazdar [11, 17, 18] and an algorithm for generating PolyLex entries from the CELEX lexical database is described Cahill [15]. The problem addressed by Cahill and Gazdar is illustrated by lexicons typically produced for machine translation (MT). MT lexicons are usually monolingual (as has been the focus of most theoretical work on LKRLs). However, in the case of related languages (in this case English, Dutch, and German), there are commonalties that could potentially be captured. In their approach, Cahill and Gazdar use DATR to encode lexicons that contain entries for all three languages, and capture generalizations that may be inherited by each individual language. Past MT work on multilingual lexicons has focused only on semantics, which Cahill and Evans argue is adequate for multilingual dictionaries of unrelated languages (i.e., English and Japanese), but not for related languages. Given that an LKRL such as DATR increases the ability to capture generalizations and decreases maintenance for monolingual lexicons, Cahill and Gazdar assert that these advantages can also be applied to multilingual dictionaries for related languages In their series of articles, Cahill and Evans demonstrate that not only semantic generalizations can be captured across related languages, but also orthographic, phonological, morphological, and syntactic information may be captured, thereby facilitating

the capture of generalizations and decreasing maintenance effort. Information common to all words in the language are pushed to the top in monolingual hierarchical lexicons. In the case of multilingual lexicons, Cahill and Evans propose and demonstrate pushing to the top information common across languages.

In [11], Cahill and Evans propose an architecture for multilingual lexicons. They discuss and provide examples of the similarities between English, Dutch, and German, and in terms of orthography, phonology, morphology, morphophonology, and syntax. For example, they note that often all three languages have identical argument slots for the subcategorization frames for verbs. Hierarchical monolingual lexicons make use of inter-connected (but mostly disjoint) hierarchies for all levels of description (orthography, phonology, morphology, morphophonology, and syntax). The authors show how the same concepts may be applied to hierarchical multilingual lexicons of related languages. In their proposed architecture, by use of orthogonal multiple inheritance, a lexeme may inherit from both its language specific hierarchies, and from hierarchies representing those features that are common to all the languages in the lexicon. Information unique to a language is encoded in the language specific hierarchies, and information common across languages is inherited from the common hierarchies. Cahill and Gazdar note that by inheritance from the common set of hierarchies, the size of the language specific hierarchies may be reduced[55]. Finally, Cahill and Gazdar show how hierarchical multilingual lexicons can provide information for educated guesses about lexical incompleteness—that is, missing entries. Clues may be gleaned based on information contained in related lexical entries. According to the authors,

---

[55] In [15], Cahill also notes that lexemes in each language specific hierarchy can have common non-semantic features (i.e., phonology and morphology) without being semantically related.

the ability to support educated guesses about lexical incompleteness adds robustness to lexicons.

In [18], Cahill and Gazdar describe the architecture of PolyLex, a trilingual lexicon of English, Dutch, and German that is an implementation of the architecture the authors proposed in [11]. Much of the PolyLex article is redundant to the later article. However, the authors report statistics from the PolyLex project to support their claim that inheritance-based hierarchical lexicons can be used to capture information common across languages. In the case of PolyLex, they note "…a significant degree of shared information across all levels of lexical representation." [18]. At the time of writing, most of the work for PolyLex had focused on morphology and morphophonology. They were able to capture common information about syllable structure, word structure, and morphology. Regarding phonology, they found that in the case of non-verbal lexemes, "A simple count of the number of substantial equations in each hierarchy for the nouns and other categories shows that around 45% of Dutch phonology is defined in the common hierarchy; around 40% of English and around 38% of German." [18]. Maintenance of such a lexicon would be significantly easier than maintenance of three separate lexicons. Cahill and Gazdar believe that if PolyLex used a featural rather than segmental approach to Phonology, the percentage of phonology defined in the common hierarchy would be significantly larger.

In [17], Cahill and Gazdar report on how morphological alternation (allomorphy) is handled in PolyLex. In their approach to inflectional morphology in PolyLex, Cahill and Gazdar use the lexeme as the central notion, rather than a morpheme or a word. In PolyLex, words exist as realizations of lexemes, and morphemes are morphosyntactic information used in those realizations. Phonological information in PolyLex is restricted to aspects relevant to

inflection, and, as noted above, follows a segmental rather than featural approach. As in Cahill's earlier paper [12], discussed above, syllables are described as context-free phrase structure rules, and consist of an onset and rhyme, with a rhyme consisting of a peak and a coda. Additionally, a coda is defined as a body and tail, with the tail being the final (consonantal) segment. This is done to make it easier to refer to the final coda segment in rules. Also, as described in [12], nodes are posited for disyllables and trisyllables. In PolyLex, SAMPA is used for the phonetic alphabet.

In their approach to allomorphy in PolyLex, Cahill and Gazdar make use of path extensions (default specification) on the left-hand side of equations, and conditional (if-then-else) statements on the right-hand side. They note that these two methods sometimes must be combined. Path extensions are used to handle allomorphy involving variant properties of a class of lexemes (e.g., case and number), conditional statements to handle allomorphy involving inherent properties (e.g., gender), and a combination of the two methods to handle allomorphy involving both the variant and inherent properties of a class of lexemes.

The building of lexicons is often a labor intensive effort. In [15], Cahill describes an algorithm developed for semi-automated generation of lexical entries for PolyLex from the CELEX lexical database [4]. For each CELEX entry, an English, Dutch, and German lexical entry is generated. Potentially, a common lexical entry is also generated. At the time of writing, the algorithm had been used to generate entries for PolyLex from CELEX for some 2,300 lexemes. While it would have been relatively straight forward to generate separate monolingual lexicons from CELEX, it was more challenging to generate a single multilingual lexicon that avoided redundancy—the point of the PolyLex project. Avoiding redundancy during automated extraction of entries from CELEX required an algorithm that

could abstract generalizations across multilingual entries. The algorithm focused mainly on the extraction of phonological information for English, Dutch, and German, along with some limited class and semantic information.

An initial step was to create a list of the 5,000 most frequently occurring English words, according to CELEX frequency data. Each word was manually translated to its Dutch and German counterpart. The nouns were also semantically classified. As part of the extraction process, secondary stress (recorded in CELEX, but not PolyLex) was removed from each CELEX entry, and primary stress was shifted from the beginning of syllables to the rhyme (to fit PolyLex). Each word was loaded into an array, with array values for each syllable, and array values for each segment in each syllable. An array was created for each of the three languages. Also, words were automatically analyzed to assign them to an appropriate inflection-related declension class. Cahill notes that out of 1,300 nouns processed, the appropriate declension class could not be automatically determined for only about 5-6 entries for each language. In addition to phonological and declension information, the semantics of each entry was also recorded (i.e. the "meaning" of each word extracted from CELEX). Next, the language specific arrays were analyzed in order to populate an array representing commonality across the three languages. Note that the automated detection of commonality is in terms of phonology only. Once the four arrays were populated for each entry, they were output to a DATR code format. At the time of writing, some 1,300 nouns had been extracted, and 1,000 other non-verbal words. Future work was to include expansion of the algorithm to include verbs.

2.3.5.7 Summary

This section has discussed some of the important literature pertaining to DATR. Four articles covering DATR inference and semantics were discussed. Approaches to phonology and morphophonolgy in DATR were also discussed. These approaches centered on the notion of syllable structure. The ability to encode feature-based phonology in DATR was demonstrated. Other literature discussed included two articles on the encoding of LTAG grammars in DATR, and four articles on the use of DATR to encode multilingual lexicons. This literature discussed in this section has shown that DATR is a powerful language for encoding lexically-centered analyses, and can encode linguistic data at all levels of description.

2.3.6 Implementations of DATR

Evans and Gazdar [32] state that DATR may be implemented with about a page of PROLOG code. A number of free implementations of DATR are available. A relatively recent list of implementations is provided at a DATR web site at the University of Leuven [35], most of which were coded in PROLOG. The DATR implementation used for this author's research is ZDATR [44], and is available for download at http://coral.lili.uni-bielefeld.de/DATR/.

ZDATR 2.0 was developed by the Linguistics and Literary Studies department at the University of Bielefeld, Germany. The original version was developed by Christoph Schillo, but version 2.0 is mainly due to the efforts of Dafydd Gibbon and Grigoriy Strokin. ZDATR 2.0 was written in ANSI C, targeting UNIX and Windows 95. Though copyrighted, the source code is available under the guidelines of the GNU Public License. Binaries are available for both UNIX and Windows 95. ZDATR consists of two modules, *zdatrtok* and

zdatrinf. Plain text files containing lexical entries written in DATR (i.e. DATR theory files) are tokenized by *zdatrtok*. The output of *zdatrtok* is used by *zdatrinf*, which contains a DATR inference engine. In addition to theory files and token files, other file types used or produced by ZDATR are query files (lists of queries to be batch processed by the inference engine), batch query declaration files (complex query statements, including conditional queries, for batch processing by the inference engine), and log files. ZDATR 2.0 includes a library of built-in functions that include all functions defined by the DATR standard library RFC 2.0 [33], as well as some additional functions. There is also a library of external functions, with a one-for-one correspondence to those defined by the DATR standard library RFC 2.0.

The DATR Standard Library RFC 2.0 [33] was developed by Evans and Gazdar, but with contributions from seven named colleagues. The functions specified by the library fall into four major categories: functions that an implementation can supply, though they can be coded by a user in DATR itself; functions that either cannot be coded or are difficult to code in DATR; arithmetic functions; and metalevel functions (e.g., functions to create descriptors from parts). The library functions are intended to be defined in the implementation language, rather than as extensions of the DATR language. The library also defines directives, such as an *include* directive that in-lines one DATR file into another.

2.4 Related Research

Based on a search of the literature and discussion with members of the SIL computational linguistics community, it is this author's belief that no past or present research addresses the issue of specifically interfacing AMPLE with DATR-based lexicons. There has been, however, research that has some relationship to that proposed by this author.

Cahill and Evans [10, 13, 14] report on the use of DATR with the Traffic Information Collator (TIC). TIC is a prototype message understanding system to report traffic to commuters. The TIC lexicon did not originally use DATR. The authors report on their work to convert the lexicon to one based on DATR. Cahill [13] reports that the aim of converting the lexicon to DATR was to improve maintenance and to improve the integration and accuracy of linguistic information in the lexicon. The number of entries in the lexicon was decreased from 1,094 to 908. Of the 908 entries, 161 were abstract. Cahill also reports that the use of a DATR-based lexicon with TIC proved the feasibility of coding a real application using DATR and that maintenance was improved because changes can be made at a few places high in the hierarchy rather than at the leaf node level. Finally, Cahill reports that run-time performance of the DATR-based lexicon was not as good as that of the previous lexicon. However, it must be kept in mind that the initial work was on the feasibility of using DATR, and did not attempt to optimize performance.

This author's research differs from that of Cahill and Evans in a number of ways. First, this research focuses on an interface between a DATR-based lexicon and a morphological parser rather than a message understanding system. Second, although a message understanding system needs some knowledge of morphology, the language in question was English, which has a relatively simple morphology. This research investigates the use of DATR with a morphological parser that has a proven track record with numerous morphologically complex non-Indo-European languages. The type of lexical information required by an English message understanding system differs from that required by a morphological parser designed to work with any of the world's languages. Third, it is not clear from the TIC literature that Cahill and Evan's project involved interfacing a DATR-

based lexicon without modification to the message understanding system itself. If they did modify the system to use a DATR-based lexicon, this author's research differs in that the proposed interface does not require modification to the legacy parser.

Duda [28] presents a real-time, bi-directional interface between DATR and PATR that he calls DUTR. The power of DATR is as a declarative LKRL. The power of PATR is as a language for processing linguistic data. Although designed to support unification-based grammars, DATR is not directly usable by PATR. One approach to interfacing DATR with PATR is static. A DATR lexicon is "compiled" to a PATR form before it is used. Another approach is a dynamic (run-time) interface. In both of these approaches, the interface is in one direction, from DATR to PATR. Duda's approach is both dynamic and bi-directional. Duda's interface differs from that proposed by this author because PATR is a language for syntactic parsing, not morphological parsing. Indeed, SIL has incorporated an AMPLE morphological parser into PC-PATR, SIL's PATR tool.

Andry et al. describe DIALEX, a tool for developing inheritance-based lexicalized grammar knowledge bases [1]. DIALEX uses DATR to encode lexical information. DIALEX is part of the SUNDIAL project to develop prototype telephone dialog systems. DIALEX uses a lexicon that is partitioned into DATR nodes that are application-specific and those that are application independent, termed *base definitions*. The application-specific entries inherit from the base definitions. A single lexicon is used to generate (compile) multiple lexicons designed for differing purposes. The research reported for DIALEX differs from that proposed by this author in the following way. The authors state that entries in the compiled lexicon were fully inflected. This avoided the need for morphological parsing since the languages targeted were English and French, which are not morphologically

complex. The authors believe that the needs of speech processing ruled out real-time morphological processing. However, the lexicon used for input to the DIALEX lexicon generator (compiler) did not use fully inflected forms. Regularity was captured in higher, abstract nodes, and only irregularity was captured in leaf nodes. The lexicon generator compiled the input lexicon into a new one with fully inflected forms, thus avoiding the need for morphological parsing. The authors concluded that static compilation is preferable over dynamic in the case of speech recognition applications. This author's research, however, is morphologically centered both in the main DATR-based lexicon and the AMPLE lexicon that is be compiled-out by the interface. The compiled-out lexicon does not avoid morphological parsing, it supports it.

2.5 Summary

This chapter has presented a review and analysis of the literature relevant to this author's research. The review and analysis began with an exposition of the nature of morphology as a sub-field of grammar, and the role of the morphological parser in NLP. Next, a specific morphological parser, AMPLE, was examined. It was noted that AMPLE's handling of morphophonemics is based on traditional American Structuralist views of language, and its handling of morphotactics is based on a variety of approaches. The role of the lexicon in linguistic theory was examined. The trend toward lexicalization was discussed, and the nature of lexicalization was explored from the perspective of HPSG, unification-based formalisms, and PATR II. It was shown that grammar has been simplified at the expense of increased complexity in the lexicon. Next, the use of inheritance was explored as a mechanism to manage complexity in the lexicon and to better capture linguistic generalizations. The role of the lexicon in NLP was discussed, then a specific LKRL was

examined, namely, DATR, and its ability to support default multiple inheritance was demonstrated. It was noted that DATR was specifically developed to support feature-based unification approaches to grammar. Finally, related research was examined.

CHAPTER 3

USING DATR WITH AMPLE

3.1 Introduction

This chapter discusses the author's research on the use of DATR with AMPLE within the parameters defined by the research hypotheses. The chapter begins by describing the foundational work that was done as the basis of the research. The discussion then switches to how AMPLE lexical data may be encoded in DATR. Subsequent sections in the chapter discuss how generalizations may be captured in DATR, how the author's interface between DATR and AMPLE works, and how other kinds of lexicons may be generated from the base DATR lexicon. The discussion is centered around the research hypotheses. The chapter mostly focuses on AMPLE and DATR lexicons built to support morphological parsing of data from the Ogea language. However, the Yalálag Zapotec language of Mexico is also used for some of the discussion, specifically in the section 3.5 Capturing Generalizations and Reducing Redundancy with DATR. The chapter also addresses other topics beyond the hypotheses: how to handle many-to-many relationships in the lexicon, factors that affect the decrease in redundancy, ease of use issues, and general versus linguistically motivated abstractions.

3.2 Foundational Work

As foundational work to investigate the research hypotheses, this author did the following. First, the morphophonemics and morphotactics of the Ogea language of Papua New Guinea were analyzed. The analysis is presented in Appendix 1. Next, an AMPLE lexicon was created for the Ogea language based on the analysis described in Appendix 1.

This lexicon uses the unified dictionary approach, combining both roots and suffixes into a single database file. The Ogea AMPLE lexicon is presented in Appendix 2. The Ogea AMPLE lexicon was tested by using all examples from Appendix 1 as input to AMPLE for parsing. The lexicon presented in Appendix 2 accurately parses all the examples. Next, an Ogea DATR lexicon was created, along with a mechanism within the lexicon to support an interface between DATR and AMPLE. In addition, this author developed DATR code to support the generation of other lexicons from the base DATR lexicon, e.g., AMPLE lexicons using different fields as the record field (i.e., the English gloss, the English morph name, Ogea, or Tok Pisin), or an English Dictionary. The Ogea DATR lexicon is presented in Appendix 3. The Ogea DATR lexicon combines the root and suffix entries into a single file. In addition to the Ogea DATR lexicon (theory) file, Appendix 3 also presents the DATR query file and a DOS batch file used to generate an AMPLE lexicon from the DATR lexicon. Finally, Appendix 3 presents a DATR query trace file, which is the raw output from running the query file.

Whereas the work done on Ogea was complete both in breadth and depth, the work done using Yalálag Zapotec was selective. However, Appendix 4 presents a fully functional AMPLE root database file along with the DATR equivalent, using a subset of records from specific order classes. AMPLE records were selected from a set of Yalálag Zapotec AMPLE database files supplied to this author by Dr. H. Andrew Black.

Appendix 5 presents a number of Perl scripts, including one which was used to compare the Ogea AMPLE lexicon generated from DATR against the original Ogea AMPLE lexicon, and does final cleanup and generation of a database file that can be run in AMPLE. The final output produced by the Perl script was tested to verify that the parse results are the

same. This verification was done programmatically using a second Perl script, also presented in Appendix 5.

In summary, the foundational work for the research was the development of a full Ogea AMPLE and DATR lexicon and a partial Yalálag Zapotec DATR lexicon. Except for the Yalálag Zapotec AMPLE files supplied by Dr. Black, all work was original work by this author.

In terms of software tools used, the AMPLE software supplied in version 1.06 of SIL's CARLA Studio was used for the morphological parsing of Ogea. ZDATR 2.0 was used as the DATR implementation. ZDATR was described above in the literature chapter. Active State's Active Perl build 517 was used to run the Perl scripts.

3.3 Managing Name Space

AMPLE and DATR manage name space differently. In AMPLE, each record is individually identified by the morphname field. The value for the morphname must be unique across the lexical name space for records. Also, in AMPLE, certain fields may repeat within a record. For example, both the allomorph field and the morpheme property field may have multiple occurrences.

In DATR, each node must have a unique name. (A node is the analog of a record in AMPLE). Nodes start with an initial uppercase letter. On the right-hand side of a path, any string that begins with an uppercase letter is interpreted by DATR to be the name of a node. If a string is node a node name, and if the string starts with an initial uppercase letter, the string must be written with single quotation marks around it. The same applies to strings that start with a DATR reserved character. Also, within an individual node, all paths listed must have unique names on the left-hand side. This means that unlike AMPLE, multiple

occurrence fields cannot use the same "field marker". This creates issues in mapping between AMPLE and DATR as will be seen in the following section.

3.4 Encoding AMPLE Lexical Data in DATR

The purpose of this section is demonstrate that each and every field type found in AMPLE dictionary records can be encoded in DATR. This is done in support of the hypothesis:

> H_1: All types of lexical information expressible in the AMPLE legacy LKRL are also expressible in DATR.

Per the AMPLE documentation for AMPLE 3.3, there are 13 basic field codes[56] used for dictionary files: Allomorph (\a), Category (\c), Elsewhere allomorph (\e), Feature descriptor (\f), Infix location (\loc), Order class (\o), Morpheme co-occurrence constraint (\mcc), Morphname (\g), Morpheme property (\p), Morpheme type (\type), No-Load (\no) Root gloss (\ge), and Underlying form (\u). In AMPLE, morpheme type fields are used with unified dictionaries. A unified dictionary is a single file that combines prefixes, roots, infixes, and suffixes. When separate dictionary files are used, the morpheme type field is not used. When the files are separated, it is customary to use start the record with a field marker that indicates the morpheme type. That is, \p for a prefix record, \r for a root record, \i for an infix record, and \s for a suffix record. This author has successfully generated separate AMPLE database files from DATR, however, the research presented here is based on a unified dictionary in order to make use of the full set of AMPLE dictionary fields.

---

[56] Note that the 'do not load' field in AMPLE is probably not necessary in a DATR lexicon, since an undesired field can be suppressed by omitting that field in the DATR to AMPLE interface. Also note that the actual field codes used are under the user's control. The ones presented here are ones used by convention.

How, then, can AMPLE dictionary fields be encoded in DATR? DATR is a free-form language in that there are no predefined names for nodes[57], descriptors, or paths. Therefore, the basic types of AMPLE information listed above, along with properties, constraints, and comments can easily be encoded in DATR. In a simplistic approach, a root record could be specified as a DATR node simply by using standard AMPLE field codes, with or without backslashes:

```
(91)  R_X[58]:
      <\a> ==
      <\c> ==
      <\e> ==
      <\f> ==
      <\loc> ==
      <\o> ==
      <\mcc> ==
      <\g> ==
      <\p> ==
      <\type> ==
      <\no> ==
      <\ge> ==
      <\u> ==.

(92)  R_X:
      <a> ==
      <c> ==
      <e> ==
      <f> ==
      <loc> ==
      <o> ==
      <mcc> ==
      <g> ==
      <p> ==
      <type> ==
      <no> ==
      <ge> ==
      <u> ==.
```

As an alternative, the following could be used:

[57] Except for a few reserved names used for the DATR library functions.

[58] The R_ prefix (for Root) or the S_ prefix (for suffix) is just a convention to manage namespace in the lexicon. It has no special meaning to DATR.

```
(93)  R_X:
      <allomorph> ==
      <category> ==
      <elsewhere allomorph> ==
      <feature descriptor> ==
      <infix location> ==
      <order class> ==
      <morpheme co-occurrence constraint> ==
      <morphname> ==
      <morpheme property>  ==
      <morpheme type> ==
      <no-Load> ==
      <root gloss> ==
      <underlying form> == .
```

However, there are at least three problems with (91)-(93). First, merely mimicking AMPLE records in DATR does not allow one to capture generalizations and reduce redundancy. This will be addressed in the following section. Second, no provision is made to take advantage of DATR's default specification. Third, this simplistic approach does not handle the problem of repeating fields found in AMPLE records. For example, there may be more than one allomorph. For sake of discussion, we will now proceed with two alternatives means of specifying AMPLE dictionary field data in DATR:

```
(94)  R_X:
      <> ==                    % everything else is undefined
      <lex type> ==            % record type
      <lex gloss eng> ==       % English gloss
      <lex gloss nat> ==       % national language gloss
      <lex gloss eng morname> == % morphname
      <lex cat> ==             % category
      <lex else allo> ==       % elsewhere allomorph
      <lex feat> ==            % feature
      <lex under> ==           % underlying form
      <lex loc> ==             % infix location
      <lex prop> ==            % property
      <lex noload> ==          % no load
      <lex order class> ==     % order class
      <lex mcc> ==             % morpheme co-occurrence constraint
      <lex com> ==             % comment
      <lex form> == <allo 1> <allo 2>
      <allo 1> ==              % first allomorph
      <allo 2> == .            % second allomorph, etc.
```

```
(95)  R_X:
      <> ==                    % everything else is undefined
      <lex type> ==            % record type
      <lex gloss eng> ==       % English gloss
      <lex gloss nat> ==       % national language gloss
      <lex gloss eng morname> == % morphname
      <lex cat> ==             % category
      <lex else allo> ==       % elsewhere allomorph
      <lex feat> ==            % feature
      <lex under> ==           % underlying form
      <lex loc> ==             % infix location
      <lex prop> ==            % property
      <lex noload> ==          % no load
      <lex order class> ==     % order class
      <lex mcc> ==             % morpheme co-occurrence constraint
      <lex com> ==             % comment
      <lex form> == <form 1> <form 2>
      <form> == <allo form> <allo prop>
                <allo sec> <allo mec> <allo com>
      <allo form 1> ==           % allomorph form
      <allo prop 1> ==           % allomorph property
      <allo sec 1> ==            % allomorph SEC
      <allo mec 1> ==            % allomorph MEC
      <allo com 1> ==            % allomorph comment
      <allo form 2> ==           % allomorph form
      <allo form 2> ==           % allomorph form
      <allo prop 2> ==           % allomorph property
      <allo sec 2> ==            % allomorph SEC
      <allo mec 2> ==            % allomorph MEC
      <allo com 1> == .          % allomorph comment
```

The node structures presented in (94) and (95) do, in fact, encode all DATR dictionary field data. (However, they are neither elegant nor the final proposed structure. In later sections we will see the use of inheritance, which greatly simplifies matters.) The difference between the two above structures has to do with how they handle the fact that multiple allomorphs may occur.

In (94), the fact that multiple allomorphs may occur is handled by the path `<lex form>`, which can contain one or more paths as values, each specifying a numeric extension (e.g., `<allo 1> <allo 2>`) that indicates an allomorph. This allows each path in the node to be uniquely named. The number of paths specified must match the number of allomorphs required. By adding the path `<lex form> == <allo 1> <allo 2>`, it is possible to write a query for the `<lex form>` of a node and have all allomorphs returned as a query result.

A shortcoming of the structure in (94) is that querying for an allomorph will return a single string that combines the allomorph form, properties, string environment constraints (SECs), and morpheme environment constraints (MECs). It would be better if the node structure allowed each of the four parts to be identifiable and therefore separately retrievable without parsing a string. This is accomplished in the version seen in (95).

In (95), the fact that multiple allomorphs may occur is again handled by the use of the path <lex form>, which contains one or more paths as values, each specifying a number (e.g., <form 1> <form 2>) that indicates an allomorph. Again, the number of paths specified must match the number of allomorphs. Note that when the right-hand side is evaluated (e.g. <form 1>), by default specification the search path will then match the left-hand path <form>. (For example, the path <form 1> extends the path <form>, therefore the value from <form> will be used.) The path <form> specifies as its value the structure of an allomorph. That is, an allomorph is composed of the allomorph form, allomorph properties, allomorph SECs, and allomorph MECs. Because of the way default specifications work in DATR, when a query path such as <form 1> matches the left-hand path <form>, the '1' gets added by DATR to the paths specified on the right-hand side. Thus, when the query path <form 1> reaches the path <form>, the right-hand paths get converted to <allo form 1> <allo prop 1> <allo sec 1> <allo mec 1>. By creating separate paths, the information can be easily obtained during a query instead of having to parse it out of a literal string as in (94). That is, the DATR lexicon can be directly queried for, say, the SEC for allomorph 1.

If an allomorph property path, SEC path, or MEC path contain multiple values, these can either be encoded as a single literal on the right-hand side, or broken into independent paths using a technique similar to that shown above for encoding multiple allomorphs.

Using the node structure defined in (94) and (95), following are some examples. Each example is presented in three parts. Part (a) is the AMPLE record, (b) is the DATR version using the structure in (94), and (c) is the DATR version using the structure in (95).

(96) Example from Ogea

(a)

```
\g hit.O1s
\ge hit me
\type r
\a yari
\a yar / _ [V]
\u yari
\mp MC1
\c VR
```

(b)

```
R_hit_O1s:
<lex type> == r
<lex gloss eng> == 'hit me'
<lex gloss nat> == 'paitim mi'
<lex gloss eng morname> == 'hit.O1s'
<lex cat> == 'VR'
<lex under> == yari
<lex prop> == 'MC1'
<lex order class> == 0
<lex form> == <allo 1> '\n' <allo 2>
<allo 1> == 'yari / _ [C]'
<allo 2> == 'yar / _ [V] '.
```

(c)

```
R_hit_O1s:
<lex type> == r
<lex gloss eng> == 'hit me'
<lex gloss nat> == 'paitim mi'
<lex gloss eng morname> == 'hit.O1s'
<lex cat> == 'VR'
<lex under> == yari
<lex prop> == 'MC1'
<lex order class> == 0
<lex form> == <allo 1> <allo 2>
<allo> == <allo form> <allo prop>
<allo sec> <allo mec> '\n'
<allo form 1> == 'yari '
<allo sec 1> == '/ _ [C] '
<allo form 2> == 'yar '
<allo sec 2> ==  '/ _ [V] '.
```

(97) Example from Yalálag Zapotec

(a)

```
\p a
\_no 00001
\u a
\a 0 takes_class2  / _ [C-lenis]  +/ [Modal] _  | e.g.
                                                  zaddirj
\a ey / y _ [V]  +/ P _
\a ay / _ [V]
\a a  / _ [C]
\a e  / _ :s     +/ _ Caus      | in some dialects
\a e  / y _      +/ P _
\a i  +/ _ avergonzar
\c V/V VA/VA NumA/NumA
\g Rep          | Repetitive
\o -20
\mp class1   | Rep acts as if it were a
                 class 1 verb root
```

(b)

```
P_i:
<> ==
<lex type> == p
<lex gloss eng morname> == 'Rep | Repetitive'
<lex cat> == 'V/V VA/VA NumA/NumA'
<lex under> == 'a'
<lex prop> == 'class1 | Rep acts as if it were a class 1
                                          verb  root'
<lex noload> == '00001'
<lex order class> == '-20'
```

```
<lex form> == '\n' <allo 1> '\n' <allo 2> '\n' <allo 3>
              '\n' <allo 4> '\n' <allo 5>
              '\n' <allo 6> '\n' <allo 7>
<allo 1> == '0 takes_class2 / _ [C-lenis] +/ [Modal] _
                                | e.g. zaddirj '
<allo 2> == 'ey / y _ [V] +/ P _'
<allo 3> == 'ay / _ [V] '
<allo 4> == 'a / _ [C] '
<allo 5> == 'e / _ :s +/ _ Caus | in some dialects '
<allo 6> == 'e / y _ +/ P _ '
<allo 7> == 'i +/ _ avergonzar'.
```

(c)

```
P_i:
<> ==
<lex type> == p
<lex gloss eng morname> == 'Rep | Repetitive'
<lex cat> == 'V/V VA/VA NumA/NumA'
<lex under> == 'a'
<lex prop> == 'class1 | Rep acts as if it were a class 1
                                              verb root'
<lex noload> == '00001'
<lex order class> == '-20'
<lex form> == <form 1> <form 2> <form 3> <form 4>
              <form 5> <form 6> <form 7>
<form> == <allo form> <allo prop> <allo sec>
          <allo mec> <allo com> '\n'
<allo form 1> == '0 '
<allo prop 1> == 'takes_class2 '
<allo sec 1> == '/ _ [C-lenis] '
<allo mec 1> == '+/ [Modal] _ '
<allo com 1> == '| e.g. zaddirj '
<allo form 2> == 'ey '
<allo sec 2> == '/ y _ [V] '
<allo mec 2> == '+/ P _'
<allo form 3> == 'ay '
<allo sec 3> == '/ _ [V] '
<allo form 4> == 'a '
<allo sec 4> == '/ _ [C] '
<allo form 5> == 'e '
<allo sec 5> == '/ _ :s '
<allo mec 5> == '+/ _ Caus '
<allo com 5> == '| in some dialects '
<allo form 6> == 'e '
<allo sec 6> == '/ y _ '
<allo mec 6> == '+/ P _ '
<allo form 7> == 'i '
<allo mec 7> == '+/ _ avergonzar'.
```

Running the query `R_hit_O1s:<lex form>` against either (96) (b) or (96) (c), the following is returned by DATR:

```
(98)  yari / _ [C]
      yar / _ [V]
```

Running the query `P_i:<lex form>` against either (97) (b) or (97) (c), the following is returned:

```
(99)  0 takes_class2 / _ [C-lenis] +/ [Modal] _ | e.g. zaddirj
      ey / y _ [V] +/ P _
      ay / _ [V]
      a / _ [C]
      e / _ :s +/ _ Caus | in some dialects
      e / y _ +/ P _
      i +/ _ avergonzar
```

Note the following about (96) and (97). First, note the use of single quotation marks. As mentioned earlier, this is to prevent DATR from thinking a string beginning with an uppercase letter is a node name when it is not. Second, note that DATR allows quoted strings to contain formatting codes, such as ones routinely found in languages such as C. For example, note the use of '\n' to insert a line break in the output of the query. Also note the encoding of comments. Comments may be added directly into quoted string literals, or they may be encoded using the `<lex com>` path.

This section has demonstrated the truth of the hypothesis

> H_1: All types of lexical information expressible in the AMPLE legacy LKRL are also expressible in DATR.

The truth of the hypothesis is claimed based on the proof criteria specified for H_1. All AMPLE dictionary fields were listed. Then, corresponding DATR code was presented. DATR code was provided that encoded all types of AMPLE dictionary record field data. Because of the flexibility of DATR, there are numerous ways to do so. Each type of data may be easily encoded, as may repetitive data. However, so far, the DATR code presented to encode AMPLE dictionary field data has been verbose and inelegant. In the sections that

follow additional ways of encoding AMPLE data in DATR will be presented--ways that capture generalizations and reduce redundancy in the lexicon.

3.5 Capturing Generalizations and Reducing Redundancy with DATR

This section addresses the hypotheses:

> H_3: There are generalizations not possible to capture in the AMPLE legacy LKRL that can be captured by use of DATR's inheritance mechanism.
>
> H_4: Such generalizations can be exploited by 5% or more of lexical entries in the lexicon for a specific language.

Per proof criterion 3, H_3 will be considered true if examples can be shown from AMPLE lexicons for at least two languages where lexical entries contain information that may be generalized and DATR code is presented showing how the generalizations may be captured. Per proof criterion 4, H_4 will be considered true if for some DATR lexicon, $I / A * 100 \geq 5$, where *I* is the count of all inheriting nodes (lexical entries that inherit information from another node), and *A* is the count of all nodes (all lexical entries).

This section presents AMPLE and DATR lexical entries from the Ogea and the Yalálag Zapotec languages to satisfy proof criteria 3 and 4. It will be shown that through DATR there are a number of ways to capture generalizations and reduce redundancy in the AMPLE lexicons for these languages. The capturing of generalizations in DATR will be shown in three major areas: the use of abstract lexical entries, the use of variable-like constructs, and the use of morphophonemic rules.

3.5.1 The Use of Abstract Lexical Entries

Abstract lexical entries are entries that capture generalizations applicable to some category or class of lexemes. This authors analysis of AMPLE database files for both Yalálag Zapotec and Ogea revealed AMPLE records that contained redundant information--

information that could be factored out and placed in an abstract DATR node from which the information could subsequently be inherited. The first language examples will come from Yalálag Zapotec.

The AMPLE root database file, *yalrt.db*, contains approximately 1,442 records. Of these, some 30 records belong to Order Class[59] 9, and 44 belong to Order Class 10. Examination of the Order Class 10 records revealed co-occurrence of certain field values. Take for example the following two root records:

```
(100) \r sera'll
(101) \_no 00764
(102) \u sela'ch
(103) \a sela'ch
(104) \c VA
(105) \g desear
(106) \m 10
(107) \mp vt
(108) \mp class2
(109) \mp takes_null_S

(110) \r la'll
(111) \_no 00498
(112) \u la'ch
(113) \a la'ch
(114) \c VA
(115) \g vagar
(116) \m 10
(117) \mp vi
(118) \mp class2
(119) \mp takes_null_S
```

Note that both records have the following field values in common: `\c VA` (lines (104) and (114)), `\m 10` (lines (106) and (116)), `\mp class2` (lines (108) and (118)), and `\mp takes_null_S` (lines (109) and (119)). Analysis of all 44 Order Class 10 records showed that 43 of the 44 records have these same field values, and that one has all the same field values but omits `\mp takes_null_S`. Analysis of all 30 Order Class 9 records showed that

[59] Morphemes are assigned an order class number. This number is used by AMPLE to determine if a given morpheme may occur before another morpheme.

the field values `\c VA` (lines (104) and (114)) and `\mp class2` (lines (108) and (118)) occurred in all Order Class 9 and Order Class 10 records. Here, then, is redundant, predictable data which may be abstracted or generalized, and placed into an abstract node from which each Order Class 9 or 10 node may inherit.

This author created a DATR theory file that contains nodes for all 44 Order Class 10 records, and four Order Class 9 records. The following abstract nodes come from the section titled 'Categories' in the file *yal.dtr*, listed in Appendix 4:

```
(120) C_MORPHEME:
(121) <> ==
(122) <lex> ==
(123) <amp lex> == I_AMPLE
(124) <mor> == ALLOMORPH
(125) <lex name> == "<phon under>"
(126) <lex under> == "<phon under>"
(127) <lex order class> ==
(128) <lex form> == DEFAULT
(129) <lex add prop> ==.
(130)
(131) C_ROOT:
(132) <> == C_MORPHEME
(133) <lex type> == r.
(134)
(135) V09_10:
(136) <> == C_ROOT
(137) <lex cat> == 'VA'
(138) <lex prop> == 'class2 ' "<lex add prop>".
(139)
(140) V09:
(141) <> == V09_10
(142) <lex order> == 9.
(143)
(144) V9i:
(145) <> == V09
(146) <lex prop> == 'vi ' V09:<lex prop>.

(147) V9t:
(148) <> == V09
(149) <lex prop> == 'vt ' V09:<lex prop>.

(150) V10:
(151) <> == V09_10
(152) <lex order> == 10
(153) <lex prop> == V09_10:<lex prop> 'takes_null_S '.
```

```
(154) V10i:
(155) <> == V10
(156) <lex prop> == 'vi ' V10:<lex prop>.
(157)
(158) V10ic:
(159) <> == V10
(160) <lex prop> == V10i:<lex prop> 'takes_:_Caus'.
(161)
(162) V10t:
(163) <> == V10
(164) <lex prop> == 'vt ' V10:<lex prop>.
```

The ten abstract nodes shown above contain information that is inherited by actual root nodes in the Yal DATR theory file. Note node `V09_10`, that begins at (135). This node states that all Order Class 9 and 10 records contain the values 'VA' and 'class2'.

In addition to the node `V09_10`, which is an combined abstraction for Order Class 9 and 10, there are other abstract nodes that are abstractions specifically for Order Class 9 and for Order Class 10. The node `V09`, starting at line (140), states that all nodes inheriting from node `V09` have the value '9' for their order class. Note in (141) that node `V09` inherits from `V09_10`. Thus, all nodes inheriting from `V09` will also inherit from `V09_10`.

In order to handle the fact that some Order Class 9 records have the value 'vi' for \mp, and some have 'vt' for \mp, two additional nodes occur, both of which also inherit from `V09`. These nodes, `V9i` and `V9t` (starting at (144) and (147)) are used to encode either the value 'vi' or 'vt' depending on which of the two nodes a root node inherits from. The paths `<lex prop> == 'vi ' V09:<lex prop>` and `<lex prop> == 'vt ' V09:<lex prop>` (see (146) and (149)) tell DATR that if it reaches either of those nodes when looking for a lexeme property, it should return either 'vi' or 'vt' (depending on which node it is in) and go to the node `V09` and get whatever properties are found in that node. However, note that the node `V09` does not have a path `<lex prop>`. By default specification, the path `<lex prop>` extends the empty path `<>` (141), which redirects the query to the `path <lex prop>` in node

V09_10 (138). The lexeme properties for node V09 are in the path (138). Note that the value for the path is <lex prop> == 'class2 ' "<lex add prop>". A root node that inherits through V9i will have the following morpheme properties that apply at the record level: 'vi' (from (146)) and 'class2' (from (138)). In addition to these two morpheme properties in common to all Order Class 9 intransitive verbs, notice that the right-hand value of the path in line contains "<lex add prop>". This instructs DATR to add additional properties to the two already specified by looking at the path "<lex add prop>". Because this is a quoted path, DATR will search for the path using the node name found in the global context. As will be seen below, at run time the global path will contain the name of a root node. An example of a root node that adds a lexical property (morpheme property) to those inherited is:

```
(165) R_chel:
(166) <> == V9i
(167) <lex add prop> == 'takes_null_S takes_:_Caus'.
```

The node R_chel actually has more paths than those shown above, but for purposes of discussion, only two paths are required. The path in (174), <lex add prop> == 'takes_null_S takes_:_Caus', instructs DATR to add the properties 'takes_null_S' and 'takes_:_Caus' to those properties already inherited through the path <> == V9i in (166).

How all this occurs will be shown below by use of a trace. But first, it is time to introduce a full root node example:

```
(168) R_chel:
(169) <> == V9i
(170) <lex name> == 'chel'
(171) <lex noload> == '00158'
(172) <phon under> == 'che:l'
(173) <lex gloss> == 'enrollar              | take group 2 pns?? '
(174) <lex add prop> == 'takes_null_S takes_:_Caus'.
```

In addition to the two paths already discussed, the root node `R_chel` has paths that specify the name of the lexeme (170), a 'no load' field (171), the underlying (phonological) form (172), and the gloss for the lexeme (173). How these are mapped back into AMPLE fields will be shown below during the trace discussion. Some things to note at this point are as follows. For all Order Class 9 and 10 nodes, only the following paths are required to code a root node: the underlying form (171), the no_load value (172), and the gloss (173). If the name of the lexeme (170) had been identical to the underlying form, the lexeme name path would have been omitted. If there had been no properties to add to those inherited, (174) would have been omitted. The example below, for the node `R_lall`, demonstrates the least number of paths required to code a root node using the abstract nodes developed for the Yalálag Zapotec examples:

```
(175) R_lall:
(176) <> == V10i
(177) <lex noload> == '00498'
(178) <phon under> == 'la\'ll'
(179) <lex gloss> == 'vagar'.
```

In (178) notice what is in fact a short-coming in using DATR for languages with orthographies that use a single quote mark as an orthographic symbol. The "phonological" underlying form (which is, in fact the orthographic phonological form) is `la'll`. Without the use of an escape character (the backslash), DATR would think the string literal terminated after the first `a`. DATR incorrectly interpret the underlying form to be `la`, and would also generate an error since there would be an unmatched single quote mark at the end of the literal.

Now that the basic inheritance mechanism has been introduced for the Yalálag Zapotec examples, DATR traces will be discussed for queries against two nodes, `R_lall` and `R_chell`. The queries are as follows, using `R_lall` as the node name:

```
(180) R_lall:<lex name>
(181) R_lall:<lex noload>
(182) R_lall:<lex under>
(183) R_lall:<lex cat>
(184) R_lall:<lex gloss>
(185) R_lall:<lex order class>
(186) R_lall:<lex prop>
```

Following is the trace for the above set of queries:

```
(187) la'll.
(188) 00498.
(189) la'll.
(190) VA.
(191) vagar.
(192) 10.
(193) vi class2 takes_null_S .
```

Now, using a more "verbose" mode in DATR, here is the trace for only query (180):

```
(194) =0,0,0> LOCAL R_lall:< || lex name > == V10i
(195)         GLOBAL R_lall:< lex name >
(196) =1,0,0> LOCAL V10i:< || lex name > == V10
(197)         GLOBAL R_lall:< lex name >
(198) =2,0,0> LOCAL V10:< || lex name > == V09_10
(199)         GLOBAL R_lall:< lex name >
(200) =3,0,0> LOCAL V09_10:< || lex name > == C_ROOT
(201)         GLOBAL R_lall:< lex name >
(202) =4,0,0> LOCAL C_ROOT:< || lex name > == C_MORPHEME
(203)         GLOBAL R_lall:< lex name >
(204) =5,0,0> LOCAL C_MORPHEME:< lex name > == "< phon under >"
(205)         GLOBAL R_lall:< lex name >
(206) =6,0,0> LOCAL R_lall:< phon under > == la'll
(207)         GLOBAL R_lall:< phon under >
(208) [Query 1 (7 Inferences)] R_lall:< lex name > = la'll .
```

The query directs DATR to look for `<lex name>` in the node `R_lall`. Note that this path does not explicitly exist in the node, but by default specification the search path extends the empty path `< || lex name >`. The `||` symbol stands between the left-hand part of a path that matched the search path and the right-hand part that did not. As the trace progresses notice that the empty path in each successive node refers DATR up the inheritance hierarchy. The local context keeps changing, but the global remains the same. Finally, in (204) an explict path is matched in the abstract node C_MORPHEME. The right-hand side of the expression directs DATR to look for the quoted path `"<phon under>"`, which then results in

a search using the global context to find an explicit value (206) in node R_lall. Finally, in (208) DATR tells us that R_lall:<lex name> = la'll.

The queries (181), (182), and (184) are straight-forward since the search paths are found directly in the first node checked, namely R_lall. Query (186) is the most interesting query to examine at this point:

```
(209) =0,0,0> LOCAL R_lall:< || lex prop > == V10i
(210)         GLOBAL R_lall:< lex prop >
(211) =1,0,0> LOCAL V10i:< lex prop > == vi  V10:< lex prop >
(212)         GLOBAL R_lall:< lex prop >
(213) =2,0,1> LOCAL V10:< lex prop > == V09_10:< lex prop >
                                        takes_null_S
(214)         GLOBAL R_lall:< lex prop >
(215) =3,0,0> LOCAL V09_10:< lex prop > == class2  "< lex add prop
   >"
(216)         GLOBAL R_lall:< lex prop >
(217) =4,0,1> LOCAL R_lall:< || lex add prop > == V10i
(218)         GLOBAL R_lall:< lex add prop >
(219) =5,0,0> LOCAL V10i:< || lex add prop > == V10
(220)         GLOBAL R_lall:< lex add prop >
(221) =6,0,0> LOCAL V10:< || lex add prop > == V09_10
(222)         GLOBAL R_lall:< lex add prop >
(223) =7,0,0> LOCAL V09_10:< || lex add prop > == C_ROOT
(224)         GLOBAL R_lall:< lex add prop >
(225) =8,0,0> LOCAL C_ROOT:< || lex add prop > == C_MORPHEME
(226)         GLOBAL R_lall:< lex add prop >
(227) =9,0,0> LOCAL C_MORPHEME:< lex add prop > ==
(228)         GLOBAL R_lall:< lex add prop >
(229) [Query 7 (13 Inferences)]
      R_lall:< lex prop > = vi  class2  takes_null_S  .
```

Note how the search for <lex prop> (lexeme properties) starts in R_lall, gets the property 'vi' from node V10i in (211), the property 'takes_null_S' from node V10 in (213), and the property 'class2' from node V09_10 in (215). Finally, note that in (215) DATR is instructed to search at R_lall for "<lex add prop>", but DATR is eventually referred up the hierarchy to the top-most abstract node C_MORPHEME, where we find that <lex add prop> extends the empty path (225), and is thus undefined.

Looking at a trace of the query R_chel:<lex prop> we can see how a root node may add a property in addition to those inherited:

```
(230) =0,0,0> LOCAL R_chel:< || lex prop > == V9i
(231)         GLOBAL R_chel:< lex prop >
(232) =1,0,0> LOCAL V9i:< lex prop > == vi  V09:< lex prop >
(233)         GLOBAL R_chel:< lex prop >
(234) =2,0,1> LOCAL V09:< || lex prop > == V09_10
(235)         GLOBAL R_chel:< lex prop >
(236) =3,0,0> LOCAL V09_10:< lex prop > == class2  "< lex add prop
   >"
(237)         GLOBAL R_chel:< lex prop >
(238) =4,0,1> LOCAL R_chel:< lex add prop > ==
                      takes_null_S takes_:_Caus
(239)         GLOBAL R_chel:< lex add prop >
(240) [Query 1 (7 Inferences)]
  R_chel:< lex prop > = vi  class2  takes_null_S takes_:_Caus .
```

In this case, `R_chel` adds `takes_null_S` and `takes_:_Caus` to those properties it inherits (`vi` and `class2`).

Discussion will now turn to another way in which information may be factored into abstract nodes in not only Yalálag Zapotec, but likely any language. The query `R_lall:<lex form lex>` returns the result `R_lall:<lex form lex> == \a la'll`. For the moment, ignoring the `\a`, how did DATR know that `la'll` is an allomorph, since it was not explicitly defined in the node `R_lall`? The verbose trace shows the following:

```
(241) =0,0,0> LOCAL R_lall:< || lex form lex > == V10i
(242)         GLOBAL R_lall:< lex form lex >
(243) =1,0,0> LOCAL V10i:< || lex form lex > == V10
(244)         GLOBAL R_lall:< lex form lex >
(245) =2,0,0> LOCAL V10:< || lex form lex > == V09_10
(246)         GLOBAL R_lall:< lex form lex >
(247) =3,0,0> LOCAL V09_10:< || lex form lex > == C_ROOT
(248)         GLOBAL R_lall:< lex form lex >

(249) =4,0,0> LOCAL C_ROOT:< || lex form lex > == C_MORPHEME
(250)         GLOBAL R_lall:< lex form lex >
(251) =5,0,0> LOCAL C_MORPHEME:< lex form || lex > == DEFAULT
(252)         GLOBAL R_lall:< lex form lex >
(253) =6,0,0> LOCAL DEFAULT:< lex form lex > ==
                       < lex form lex default >
(254)         GLOBAL R_lall:< lex form lex >
(255) =7,0,0> LOCAL DEFAULT:< lex form lex default > == C_DLT
(256)         GLOBAL R_lall:< lex form lex >
```

```
(257) =8,0,0> LOCAL C_DLT:< lex form lex || default > ==
                    I_AMPLE:< amp allomorph >
(258)         GLOBAL R_lall:< lex form lex >
(259) =9,0,0> LOCAL I_AMPLE:< amp allomorph || default > ==
                  \a "< mor form >" "< mor prop >" "< mor sec >"
                     "< mor mec >"
(260)
(261)         GLOBAL R_lall:< lex form lex >
(262) =10,0,2> LOCAL R_lall:< || mor form default > == V10i
(263)         GLOBAL R_lall:< mor form default >
(264) =11,0,0> LOCAL V10i:< || mor form default > == V10
(265)         GLOBAL R_lall:< mor form default >
(266) =12,0,0> LOCAL V10:< || mor form default > == V09_10
(267)         GLOBAL R_lall:< mor form default >
(268) =13,0,0> LOCAL V09_10:< || mor form default > == C_ROOT
(269)         GLOBAL R_lall:< mor form default >
(270) =14,0,0> LOCAL C_ROOT:< || mor form default > == C_MORPHEME
(271)         GLOBAL R_lall:< mor form default >
(272) =15,0,0> LOCAL C_MORPHEME:< mor || form default > == ALLOMORPH
(273)         GLOBAL R_lall:< mor form default >
(274) =16,0,0> LOCAL ALLOMORPH:< mor form default > ==
                   "< phon under >"
(275)         GLOBAL R_lall:< mor form default >
(276) =17,0,0> LOCAL R_lall:< phon under > == la'll
(277)         GLOBAL R_lall:< phon under >
(278) =10,0,4> LOCAL R_lall:< || mor prop default > == V10i
(279)         GLOBAL R_lall:< mor prop default >
(280) =11,0,0> LOCAL V10i:< || mor prop default > == V10
(281)         GLOBAL R_lall:< mor prop default >
(282) =12,0,0> LOCAL V10:< || mor prop default > == V09_10
(283)         GLOBAL R_lall:< mor prop default >
(284) =13,0,0> LOCAL V09_10:< || mor prop default > == C_ROOT
(285)         GLOBAL R_lall:< mor prop default >
(286) =14,0,0> LOCAL C_ROOT:< || mor prop default > == C_MORPHEME
(287)         GLOBAL R_lall:< mor prop default >
(288) =15,0,0> LOCAL C_MORPHEME:< mor || prop default > == ALLOMORPH
(289)         GLOBAL R_lall:< mor prop default >
(290) =16,0,0> LOCAL ALLOMORPH:< mor || prop default > ==
(291)         GLOBAL R_lall:< mor prop default >
(292) =10,0,6> LOCAL R_lall:< || mor sec default > == V10i
(293)         GLOBAL R_lall:< mor sec default >
(294) =11,0,0> LOCAL V10i:< || mor sec default > == V10
(295)         GLOBAL R_lall:< mor sec default >
(296) =12,0,0> LOCAL V10:< || mor sec default > == V09_10
(297)         GLOBAL R_lall:< mor sec default >
(298) =13,0,0> LOCAL V09_10:< || mor sec default > == C_ROOT
(299)         GLOBAL R_lall:< mor sec default >
(300) =14,0,0> LOCAL C_ROOT:< || mor sec default > == C_MORPHEME
(301)         GLOBAL R_lall:< mor sec default >
(302) =15,0,0> LOCAL C_MORPHEME:< mor || sec default > == ALLOMORPH
(303)         GLOBAL R_lall:< mor sec default >
(304) =16,0,0> LOCAL ALLOMORPH:< mor || sec default > ==
(305)         GLOBAL R_lall:< mor sec default >
(306) =10,0,8> LOCAL R_lall:< || mor mec default > == V10i
(307)         GLOBAL R_lall:< mor mec default >
(308) =11,0,0> LOCAL V10i:< || mor mec default > == V10
(309)         GLOBAL R_lall:< mor mec default >
```

```
(310) =12,0,0> LOCAL V10:< || mor mec default > == V09_10
(311)         GLOBAL R_lall:< mor mec default >
(312) =13,0,0> LOCAL V09_10:< || mor mec default > == C_ROOT
(313)         GLOBAL R_lall:< mor mec default >
(314) =14,0,0> LOCAL C_ROOT:< || mor mec default > == C_MORPHEME
(315)         GLOBAL R_lall:< mor mec default >
(316) =15,0,0> LOCAL C_MORPHEME:< mor || mec default > == ALLOMORPH
(317)         GLOBAL R_lall:< mor mec default >
(318) =16,0,0> LOCAL ALLOMORPH:< mor || mec default > ==
(319)         GLOBAL R_lall:< mor mec default >
(320) [Query 1 (48 Inferences)] R_lall:< lex form lex > = \a   la'll
(321)  .
```

This trace introduces a new set of abstract nodes, a number of which are actually part of the interface between DATR and AMPLE and will not be discussed until a section below. However, it is sufficient at this point to focus on the fact that the node `C_MORPHEME` contains a path `<lex form> == DEFAULT`. This node and path are reached during a search for the allomorphs for `R_lall`. Because the lexeme has but one allomorph, and because that allomorph is identical to the underlying form, there is no need to explicitly encode an allomorph for `R_lall`. The search path `<lex form lex>` ultimately goes up the hierarchy and by default specification extends the path `<lex form>`, and is directed to a node name `DEFAULT` in (251). Ultimately DATR reaches an interface node (not yet discussed), which defines the components of an allomorph (259). The structure of an allomorph is defined as `\a "< mor form >" "< mor prop >" "< mor sec >" "< mor mec >"`. That is, an allomorph has a morphological form, property or properties, SECs, and MECs. When DATR searches for the `<mor form default>` (the `default` being added to `mor form` by extension), it is directed to use `<phon under>` in (274). Thus, without having to explicitly define a "default" allomorph, the inheritance hierarchy can be used to capture that generalization that if a lexeme has but one allomorph, and if that allomorph has the same form as the underlying form, there is no need to explicitly encode the allomorph--it can be obtained by inheritance.

The full listing of the Yalálag Zapotec DATR lexicon may be found separately in Appendix 4. The files shown in the appendix are capable of generating AMPLE records from DATR. Once the reader has finished this chapter, enough information will have been imparted to understand the entire DATR theory file for Yalálag Zapotec.

Proof criterion H_3 requires demonstration of two languages that may make use of DATR's ability to capture generalizations and reduce redundancy in the lexicon. With the first of the two languages, Yalálag Zapotec, it has been shown that generalizations may be made about entries in the AMPLE root database file, and DATR code has been presented to show how this may be done. Although the focus was only on Order Class 9 and 10 verbs, this was sufficient for the proof criterion. In addition to generalizations that may be made regarding Order Classes 9 and 10, it was also shown that inheritance may be used to define a default allomorph, doing away with the need to explicitly encode one in some cases.

Discussion will now turn to the Ogea language. The reader should refer to Appendices 1-3. Appendix 1 provides an overview of Ogea Morphophonemics and Morphotactics. Appendix 2 provides an AMPLE unified database file with the records required to parse all examples found in Appendix 1. Appendix 3 provides the Ogea DATR lexicon files to support the generation of the Ogea AMPLE unified database file. The interface will be discussed in a separate section below. The focus of the present section is on redundancy this author found in the Ogea AMPLE database file and the generalizations and abstractions that were made to reduce redundancy by use of DATR. This author's analysis and DATR code will be presented as the second language to satisfy proof criterion H_3.

Although there are many examples which could be given for Ogea, perhaps the most interesting one is the suppletive verb roots, object, and benefactive verbal suffixes. Since

each suppletive set presents the same opportunity for generalizations, only one needs to be focused on for the present discussion. Following are Ogea AMPLE records for the suppletive verb 'to hit'.

```
(322) \g hit.O1d
(323) \ge hit us (dual)
(324) \type r
(325) \a harire
(326) \a harir / _ [V]
(327) \u harire
(328) \mp MC1
(329) \c VR

(330) \g hit.O1p
(331) \ge hit us (plural)
(332) \type r
(333) \a harige
(334) \a harig / _ [V]
(335) \u harige
(336) \mp MC1
(337) \c VR

(338) \g hit.O1s
(339) \ge hit me
(340) \type r
(341) \a yari
(342) \a yar / _ [V]
(343) \u yari
(344) \mp MC1
(345) \c VR

(346) \g hit.O2d
(347) \ge hit you (dual)
(348) \type r
(349) \a tarire
(350) \a tarir / _ [V]
(351) \u tarire
(352) \mp MC1
(353) \c VR

(354) \g hit.O2p
(355) \ge hit you (plural)
(356) \type r
(357) \a tarige
(358) \a tarig / _ [V]
(359) \u tarige
(360) \mp MC1
(361) \c VR
```

```
(362) \g hit.O2s
(363) \ge hit you (singular)
(364) \type r
(365) \a nari
(366) \a nar / _ [V]
(367) \u nari
(368) \mp MC1
(369) \c VR

(370) \g hit.O3d
(371) \ge hit them (dual)
(372) \type r
(373) \a narire
(374) \a narir / _ [V]
(375) \u narire
(376) \mp MC1
(377) \c VR

(378) \g hit.O3p
(379) \ge hit them (plural)
(380) \type r
(381) \a narige
(382) \a narig / _ [V]
(383) \u narige
(384) \mp MC1
(385) \c VR

(386) \g hit.O3s
(387) \ge hit him/her/it
(388) \type r
(389) \a wari
(390) \a war / _ [V]
(391) \u wari
(392) \mp MC1
(393) \c VR
```

Note that two fields have the same values across all records shown above: the morpheme property is `MC1`, and the category is `VR`. Also note that for each lexeme, there are two allomorphs. One allomorph is identical to the underlying form, and the other allomorph drops the final vowel by application of a syncope rule. Is there a way to capture these generalizations in DATR? By now, the reader should not be surprised to learn that it can be done. First, consider the following abstract nodes:

```
(394) C_ROOT:
(395) <> == C_MORPHEME
(396) <lex type> == r.
(397)
(398) C_R_VR:
(399) <> == C_ROOT
(400) <lex prop> == 'MC2'
(401) <lex cat> == 'VR'.
(402)
(403) C_R_VR_MC1:
(404) <> == C_R_VR
(405) <lex form> == DEFAULT SYNCOPE
(406) <lex prop> == 'MC1'.
```

And consider the following root nodes:

```
(407) R_harige:
(408) <> == C_R_VR_MC1
(409) <phon under> == harige
(410) <lex gloss eng> == ENG_hit_O1p.

(411) R_harire:
(412) <> == C_R_VR_MC1
(413) <phon under> == harire
(414) <lex gloss eng> == ENG_hit_O1d.

(415) R_nari:
(416) <> == C_R_VR_MC1
(417) <phon under> == nari
(418) <lex gloss eng> == ENG_hit_O2s.

(419) R_narige:
(420) <> == C_R_VR_MC1
(421) <phon under> == narige
(422) <lex gloss eng> == ENG_hit_O3p.

(423) R_narire:
(424) <> == C_R_VR_MC1
(425) <phon under> == narire
(426) <lex gloss eng> == ENG_hit_O3d.

(427) R_tarige:
(428) <> == C_R_VR_MC1
(429) <phon under> == tarige
(430) <lex gloss eng> == ENG_hit_O2p.

(431) R_tarire:
(432) <> == C_R_VR_MC1
(433) <phon under> == tarire
(434) <lex gloss eng> == ENG_hit_O2d.
```

```
(435) R_wari:
(436) <> == C_R_VR_MC1
(437) <phon under> == wari
(438) <lex gloss eng> == ENG_hit_O3s.

(439) R_yari:
(440) <> == C_R_VR_MC1
(441) <phon under> == yari
(442) <lex gloss eng> == ENG_hit_O1s.
```

In the case of any path not explicitly defined or extending the other two paths, each of the root nodes shown above inherits from the node `C_R_VR_MC1`. Note that the eight fields in the AMPLE version were collapsed into four lines. However, things are not quite as compact as it appears since the English gloss and morph name are inherited from a set of entries that all begin with `ENG_`. For example:

```
(443) ENG_hit_O3s:
(444) <> == "R_wari"
(445) <lex gloss eng morname> == 'hit.O3s'
(446) <lex gloss eng> == 'hit him/her/it'.
```

The use of a separate set of `ENG_` entries was not necessary, but was done to support a multi-lingual query capability--a topic of a subsequent section. If the gloss and morph name information were encoded directly in the Ogea lexeme instead of via an `ENG_` entry, the DATR nodes for the verb 'to hit' would have five lines including the node name:

```
(447) R_wari:
(448) <> == C_R_VR_MC1
(449) <phon under> == wari
(450) <lex gloss eng morname> == 'hit.O3s'
(451) <lex gloss eng> == 'hit him/her/it'.
```

From these basic nodes (and, of course, the interface nodes) it is possible to generate the corresponding AMPLE records for the suppletive verb 'to hit'. To see how this is done, let us return to the abstract nodes found in (394) through (406). All the suppletive roots for 'to hit' inherit from `C_R_VR_MC1`, e.g. (440). The property `MC1` is inherited explicitly from this abstract node (406). This node also inherits from `C_R_VR`. The category `VR` is inherited

explicitly from this node (401). Thus, the insight that VR and MC1 co-occur in all the suppletive entries for the verb 'to hit' has been captured. In fact, it has been captured not only for the verb 'to hit', but for all suppletive entries, both roots and suffixes, that have the property MC1. Notice that the path <lex prop> occurs both in C_R_VR_MC1 (406) and in C_R_VR (400), but with different values. In the Ogea DATR theory file, any verb root that has the property MC2 inherits directly from C_R_VR, e.g.:

```
(452) R_agotete:
(453) <> == C_R_VR
(454) <phon under> == agotete
(455) <lex gloss eng> == ENG_teach.
```

Roots that are suppletive and have the property MC1 inherit directly from C_R_VR_MC1 and indirectly from C_R_VR. What this means is that the property MC2 is set as the default property for all verb roots, and in the case of roots that have the property MC1, this is obtained in C_R_VR_MC1 by overriding the value inherited from C_R_VR.

Next, note that the abstract node C_R_VR_MC1 captures the generalization that verb roots inheriting from this node have two allomorphs--a default allomorph and an allomorph resulting from the application of syncope. How these rules are applied will be discussed below in a separate section. The point here is that lexicon maintenance is greatly eased in Ogea for the suppletive verb roots (a total of 36 lexemes, each with two allomorphs and redundant information). As with some of the Yalálag Zapotec examples, the default allomorph does not need to be explicitly encoded in the node for the root lexeme. In addition, for suppletive Ogea verb roots with the morpheme property MC1, the syncope allomorph does not need to be explicitly encoded in the node for each root lexeme. Both allomorphs are handled in the abstract node C_R_VR_MC1 (405), and the nodes for the root lexemes merely inherit the appropriate allomorphs. In order to aid the reader in

understanding how this works, following is a trace for the query `ENG_hit_O1s:<lex form lex>`:

```
(456) \a yari
(457) \a yar  / _ [V]
(458) .
```

The verbose trace yields:

```
(459) =0,0,0> LOCAL ENG_hit_O1s:< || lex form lex > == "R_yari"
(460)         GLOBAL ENG_hit_O1s:< lex form lex >
(461) =1,0,0> LOCAL R_yari:< || lex form lex > == C_R_VR_MC1
(462)         GLOBAL R_yari:< lex form lex >
(463) =2,0,0> LOCAL C_R_VR_MC1:< lex form || lex > ==
                DEFAULT SYNCOPE
(464)         GLOBAL R_yari:< lex form lex >
[next 111 inferences omitted by author]
(465)  [Query 1 (114 Inferences)] ENG_hit_O1s:< lex form lex > =
           \a   yari
           \a   yar     / _ [V]
           .
```

Only the first three inferences (out of 114) are shown in order to keep the example focused. Note that the search for `<lex form lex>` in `ENG_hit_O1s` went up the inheritance hierarchy, first to the node `R_yari`, then to `C_R_VR_MC1`, where DATR was told to ultimately generate both a default and a syncope allomorph (463). The result was the correct allomorphs, generated from the underlying form specified in (441).

As with Yalálag Zapotec, it has been demonstrated that in the AMPLE database file for Ogea, there are generalizations that may be captured and redundancy that may be reduced. The technique shown with Ogea is particularly important for easing maintenance since in addition to the 36 lexical entries for suppletive roots, there are an additional 18 entries for suppletive verbal suffixes (the object and benefactive suffixes). For none of the 56 lexemes is it necessary to explicitly encode the allomorphs. Also, common properties and classes are captured in abstract nodes.

3.5.2 The Use of Variable-like Constructs

In continuation of the general topic of the capture of generalizations in DATR, one area in which DATR can provide generalizations and reduce redundancy is in the application of a tried and true principle in programming--the use of variables instead of literals. For the readers convenience the result of the query `ENG_hit_O1s:<lex form lex>` is repeated below:

```
(466) \a yari
(467) \a yar  / _ [V]
(468) .
```

Note the string environment constraint (SEC) `'/ _ [V]'` found in (467). Where did this come from? Consider the following section found in the Ogea DATR theory file in Appendix 3:

```
(469) % String Environment Constraints
(470)
(471) SEC:
(472) <mec environ> == Implode:<'/ ' "<environ>"' '>
(473) <allo sec> == <mec environ.>.   % if don't cut path,
                                      must specify each extension

(474) SEC_01:
(475) <> == SEC
(476) <environ> == '_ [V]'.

(477) SEC_02:
(478) <> == SEC
(479) <environ> == '[V] _'.

(480) SEC_03:
(481) <> == SEC
(482) <environ> == '[C] _'.

(483) SEC_04:
(484) <> == SEC
(485) <environ> == '_ [B]'.

(486) SEC_05:
(487) <> == SEC
(488) <environ> == '_ [A]'.
```

```
(489) SEC_06:
(490) <> == SEC
(491) <environ> == '_ [O]'.

(492) SEC_07:
(493) <> == SEC
(494) <environ> == '_ [X]'.

(495) SEC_08:
(496) <> == SEC
(497) <environ> == 'o _ #'.

(498) SEC_09:
(499) <> == SEC
(500) <environ> == 'e _'.

(501) SEC_10:
(502) <> == SEC
(503) <environ> ==      'o _'.

(504) SEC_11:
(505) <> == SEC
(506) <environ> == 'a _'.
```

The SEC section defines a set of variable-like nodes. They are termed 'variable-like' by this author because DATR's node definition mechanism is used rather than actual variables. These nodes act like variables, but technically are not. DATR does, in fact, allow one to define actual variables. However, the DATR literature indicates that using nodes is the preferred approach, and, as will be seen below, by using nodes it is easier to do certain things than would be the case if true variables were used.

The variable-like nodes listed above are invoked mainly through another abstract node:

```
(507) ALLOMORPH:
(508) <allo sec> ==
(509) <allo sec epenthetic_w> == "SEC_02"
(510) <allo sec glide> == "SEC_01"
(511) <allo sec syncope> == "SEC_01"
(512) <allo sec n_b> == "SEC_04"
(513) <allo sec n_a> == "SEC_05"
(514) <allo sec n_x> == "SEC_07"
(515) <allo sec n_v> == "SEC_06".
```

Note that a number of paths have been removed from the node ALLOMORPH in order to focus on the topic of variable-like nodes. Briefly, here is how the variable-like nodes work for SECs. When a morphophonemic rule is applied to a lexeme's underlying form, at some point the SEC will be specified. Ideally this occurs in an abstract node rather than in the actual root or suffix lexeme node. (This will be explained further in a subsequent section). Take for example, syncope, where DATR is given a variable-like node name, e.g., SEC_01. DATR will search the specified node for the path <allo sec syncope>. However, <allo sec syncope> is not defined in any of the nodes that begin with SEC_. Instead, each SEC_ node has an empty path pointing to the abstract node SEC (475). Note that the SEC node contains the path <allo sec>, which will be extended by <allo sec syncope> (473). The value for the path <allo sec> is defined as <mec environ.> (473). Notice the period just before the path terminator symbol '>'. This strips off the 'syncope' part of <allo sec syncope> to keep it from showing up in the final output as a literal. DATR has now been redirected back to the path <sec environ>. The value for this path is Implode:<'/ ' "<environ>"' '> (472). Implode is a function defined in the standard DATR library. It returns a string that is a concatenation of the function arguments. In this case it concatenates the SEC environment symbol '/' and the results of the quoted path "<environ>". By this point, the global context is the SEC_01 node, so it is searched for the value of "<environ>", which is '_ [V]' (476). The result is the string environment constraint '/ _ [V]'. The global context is switched to an SEC_ node in the ALLOMORPH node, e.g. (510), by the use of quotes around the SEC node name.

In the node starting at (507), notice that the SEC is defined not just for the syncope rule, but a number of other rules as well. The SEC for the 'epenthetic w' rule is defined as

SEC_02, for the glide rule is defined as SEC_01, etc. Any abstract node that tells DATR to apply a morphophonemic rule to the underlying form will ultimately look in the abstract node ALLOMORPH to find the appropriate SEC to use. Note that in the case of DEFAULT, the SEC is undefined. Any root or suffix lexeme that has an SEC for a DEFAULT allomorph can explicitly define it in the root or suffix node itself.

What are the advantages of using variable-like nodes? There are at least four. First, it eases maintenance. If the SEC for each allomorph is stated using a string literal paired with each allomorph of each lexeme (as it is in AMPLE), and if future analysis determines that a modification must be made to the SEC, every lexeme node would have to be searched and modified as needed. For example, if syncope in Ogea turned out to only occur before certain vowels and not all vowels, without the use of a variable-like node changes would have to be made in hundreds of places. By using a variable-like node, the change can be made in one place and applied where needed via inheritance. Of course, this advantage is only gained if an SEC is applied to more than one allomorph in the lexicon and the SEC is predictable based on some other feature (e.g., the Ogea syncope rule is tied to the SEC_01 environment). Second, note in (472) that the string environment start marker '/' is stated here in the abstract node SEC, rather than in each individual SEC_ node. It does not need to be redundantly coded in each SEC_ node. Third, if the environment start marker is changed, it may be done so in a single node rather than in each individual SEC_ node. Fourth, by separating the environment start marker from the actual environment it is possible to potentially support different systems that use the lexicon. That is, although AMPLE uses '/' to indicate the start of a string environment constraint, what if another tool uses a different symbol? By abstracting the start marker away from the actual environment value, it is

possible to modify the DATR code shown above and make the actual start marker dependent on the system for which the environment is being generated.

Despite the advantages to using a variable-like node, there is also at least one disadvantage. In AMPLE, since the SEC is explicitly stated, a human has an easier time reading an environment since it is not necessary to do a lookup of to determine the environment. It is easier for a human to read that syncope is applied in the environment '/ _ [V]' than to encounter the cryptic node name `SEC_01`.

In the Ogea DATR lexicon found in Appendix 3, variable-like nodes are used not only for SECs, but also for MECs:

```
(516) % Morpheme Environment Constraints
(517)
(518) MEC:
(519) <mec environ> == Implode:<'+/ ' "<environ>"' '>
(520) <allo mec> == <mec environ.>.   % if don't cut path, must
   specify each extension

(521) MEC_01:
(522) <> == MEC
(523) <environ> == '~_ TO'.

(524) MEC_02:
(525) <> == MEC
(526) <environ> == '~_ P3s'.

(527) MEC_03:
(528) <> == MEC
(529) <environ> == '~_ causative'.

(530) MEC_04:
(531) <> == MEC
(532) <environ> == '_ [Contra.Subj]'.

(533) MEC_05:
(534) <> == MEC
(535) <environ> == '_ Contra'.
```

In addition to the abstract node `MEC`, the above listing only shows the first five of 25 `MEC_` nodes found in the action Ogea DATR theory file. Note that whereas SEC environments are

more likely to be re-usable across languages, MEC environments are more likely to be language dependent and thus less re-usable across languages.

This section has shown how generalizations may be made, and redundancy reduced through the use of variable-like nodes. This was illustrated using DATR code from the Ogea DATR theory file which encodes both SECs and MECs in abstract nodes. In the next section, DATR code will be presented to show how morphophonemic rules are used in the Ogea DATR theory file to generate allomorphs. This will further illustrate the use of variable-like nodes.

3.5.3 The Use of Morphophonemic Rules

Consider again the suppletive verb root lexemes for 'to hit' in Ogea, shown above in (322) - (393). Each lexeme has two allomorphs--one identical to the underlying form, and one the result of applying a syncope rule:

```
(536) \a yari
(537) \a yar  / _ [V]
```

We have already discussed the fact that redundancy may be reduced by generating the allomorphs for these lexemes rather than coding each allomorph at the root node level. We have also seen above how the SEC can be encoded using variable-like nodes. This section will describe the actual mechanism used to automatically generate allomorphs and thereby reduce redundancy and increase generalizations. The following DATR code is from the 'Templates' section of the Ogea DATR theory file listed in Appendix 3. Some paths were removed since they pertain to the generation of alternative dictionaries and would add confusion at this point. The listing starts with a small extract from the AMPLE interface. The full code will be discussed in a subsequent section.

```
(538) I_AMPLE:
(539) <amp allomorph> == \a' ' "<allo form>"' ' "<allo prop>"' '
                         "<allo sec>"' ' "<allo mec>"'\n'
(540)
(541) ALLOMORPH:
(542) <allo form default> == "<phon under>"
(543) <allo form elision> == Chop:<"<phon under>">
(544) <allo form epenthetic_w> == Implode:<w "<phon under>">
(545) <allo form glide> == Implode:<Chop:<"<phon under>"> y>
(546) <allo form redup> ==  Implode:<First:<Explode:<"<phon
   under>">>
                      First:<Rest:<Explode:<"<phon under>">>>>
(547) <allo form syncope> == Chop:<"<phon under>">
(548) <allo form n_b> == Implode:<Chop:<"<phon under>"> m>
(549) <allo form n_a> == Implode:<Chop:<"<phon under>"> n>
(550) <allo form n_x> == Implode:<Chop:<"<phon under>"> ng>
(551) <allo form n_v> == Chop:<"<phon under>">
(552) <allo prop> ==
(553) <allo mec> ==
(554) <allo sec> ==
(555) <allo sec epenthetic_w> == "SEC_02"
(556) <allo sec glide> == "SEC_01"
(557) <allo sec syncope> == "SEC_01"
(558) <allo sec n_b> == "SEC_04"
(559) <allo sec n_a> == "SEC_05"
(560) <allo sec n_x> == "SEC_07"
(561) <allo sec n_v> == "SEC_06".
(562)
(563) DEFAULT:
(564) <lex form lex> == <lex form lex default>
(565) <lex form lex default> == C_DLT.
(566)
(567) ELISION:
(568) <lex form lex> == <lex form lex elision>
(569) <lex form lex elision> == C_DLT.
(570)
(571) EPENTHETIC_W:
(572) <lex form lex> == <lex form lex epenthetic_w>
(573) <lex form lex epenthetic_w> == C_DLT.
(574)
(575) GLIDE:
(576) <lex form lex> == <lex form lex glide>
(577) <lex form lex glide> == C_DLT.

(578) NASAL:
(579) <lex form lex> == <lex form lex n_v> <lex form lex n_b>
                 <lex form lex n_a> <lex form lex n_x>
(580) <lex form lex n_b> == C_DLT
(581) <lex form lex n_a> == C_DLT
(582) <lex form lex n_x> == C_DLT
(583) <lex form lex n_v> == C_DLT.
```

```
(584) REDUP:
(585) <lex form lex> == <lex form lex redup>
(586) <lex form lex redup> == C_DLT.
(587)
(588) SYNCOPE:
(589) <lex form lex> == <lex form lex syncope>
(590) <lex form lex syncope> == C_DLT.
(591)
(592) % Default Lexical Template
(593) C_DLT:
(594) <lex form lex> == I_AMPLE:<amp allomorph>.
```

First, note that the nodes shown above have a node for each morphophonemic rule in Ogea, plus a node name ALLOMORPH and one named C_DLT. The actual morphophonemic rules are specified and applied to the underlying form of a lexeme via the ALLOMORPH node. The abstract nodes named after morphophonemic rules serve the purpose of redirection so the appropriate path may be found in the ALLOMORPH node. This is best seen by looking again at a verbose trace, this time with all inferences shown. The trace of the query

ENG_hit_O1s:<lex form lex> is:

```
(595) =0,0,0> LOCAL ENG_hit_O1s:< || lex form lex > == "R_yari"
(596)         GLOBAL ENG_hit_O1s:< lex form lex >
(597) =1,0,0> LOCAL R_yari:< || lex form lex > == C_R_VR_MC1
(598)         GLOBAL R_yari:< lex form lex >
(599) =2,0,0> LOCAL C_R_VR_MC1:< lex form || lex > == DEFAULT
   SYNCOPE
(600)         GLOBAL R_yari:< lex form lex >
(601) =3,0,0> LOCAL DEFAULT:< lex form lex > ==
                          < lex form lex default >
(602)         GLOBAL R_yari:< lex form lex >
(603) =4,0,0> LOCAL DEFAULT:< lex form lex default > == C_DLT
(604)         GLOBAL R_yari:< lex form lex >
(605) =5,0,0> LOCAL C_DLT:< lex form lex || default > ==
                       I_AMPLE:< amp allomorph >
(606)         GLOBAL R_yari:< lex form lex >
(607) =6,0,0> LOCAL I_AMPLE:< amp allomorph || default > ==
              \a   "< allo form >"   "< allo prop >"
                   "< allo sec >"   "< allo mec >"
(608)         GLOBAL R_yari:< lex form lex >
(609) =7,0,2> LOCAL R_yari:< || allo form default > == C_R_VR_MC1
(610)         GLOBAL R_yari:< allo form default >
(611) =8,0,0> LOCAL C_R_VR_MC1:< || allo form default > == C_R_VR
(612)         GLOBAL R_yari:< allo form default >
(613) =9,0,0> LOCAL C_R_VR:< || allo form default > == C_ROOT
(614)         GLOBAL R_yari:< allo form default >
```

```
(615) =10,0,0> LOCAL C_ROOT:< || allo form default > == C_MORPHEME
(616)         GLOBAL R_yari:< allo form default >
(617) =11,0,0> LOCAL C_MORPHEME:< allo || form default > ==
   ALLOMORPH
(618)
(619)         GLOBAL R_yari:< allo form default >
(620) =12,0,0> LOCAL ALLOMORPH:< allo form default > ==
                    "< phon under >"
(621)         GLOBAL R_yari:< allo form default >
(622) =13,0,0> LOCAL R_yari:< phon under > == yari
(623)         GLOBAL R_yari:< phon under >
(624) =7,0,4> LOCAL R_yari:< || allo prop default > == C_R_VR_MC1
(625)         GLOBAL R_yari:< allo prop default >
(626) =8,0,0> LOCAL C_R_VR_MC1:< || allo prop default > == C_R_VR
(627)         GLOBAL R_yari:< allo prop default >
(628) =9,0,0> LOCAL C_R_VR:< || allo prop default > == C_ROOT
(629)         GLOBAL R_yari:< allo prop default >
(630) =10,0,0> LOCAL C_ROOT:< || allo prop default > == C_MORPHEME
(631)         GLOBAL R_yari:< allo prop default >
(632) =11,0,0> LOCAL C_MORPHEME:< allo || prop default > ==
   ALLOMORPH
(633)         GLOBAL R_yari:< allo prop default >
(634) =12,0,0> LOCAL ALLOMORPH:< allo prop || default > ==
(635)         GLOBAL R_yari:< allo prop default >
(636) =7,0,6> LOCAL R_yari:< || allo sec default > == C_R_VR_MC1
(637)         GLOBAL R_yari:< allo sec default >
(638) =8,0,0> LOCAL C_R_VR_MC1:< || allo sec default > == C_R_VR
(639)         GLOBAL R_yari:< allo sec default >
(640) =9,0,0> LOCAL C_R_VR:< || allo sec default > == C_ROOT
(641)         GLOBAL R_yari:< allo sec default >
(642) =10,0,0> LOCAL C_ROOT:< || allo sec default > == C_MORPHEME
(643)         GLOBAL R_yari:< allo sec default >
(644) =11,0,0> LOCAL C_MORPHEME:< allo || sec default > == ALLOMORPH
(645)         GLOBAL R_yari:< allo sec default >
(646) =12,0,0> LOCAL ALLOMORPH:< allo sec || default > ==
(647)         GLOBAL R_yari:< allo sec default >
(648) =7,0,8> LOCAL R_yari:< || allo mec default > == C_R_VR_MC1
(649)         GLOBAL R_yari:< allo mec default >
(650) =8,0,0> LOCAL C_R_VR_MC1:< || allo mec default > == C_R_VR
(651)         GLOBAL R_yari:< allo mec default >
(652) =9,0,0> LOCAL C_R_VR:< || allo mec default > == C_ROOT
(653)         GLOBAL R_yari:< allo mec default >
(654) =10,0,0> LOCAL C_ROOT:< || allo mec default > == C_MORPHEME
(655)         GLOBAL R_yari:< allo mec default >
(656) =11,0,0> LOCAL C_MORPHEME:< allo || mec default > == ALLOMORPH
(657)         GLOBAL R_yari:< allo mec default >
(658) =12,0,0> LOCAL ALLOMORPH:< allo mec || default > ==
(659)         GLOBAL R_yari:< allo mec default >
(660) =3,0,1> LOCAL SYNCOPE:< lex form lex > ==
                    < lex form lex syncope >
(661)         GLOBAL R_yari:< lex form lex >
(662) =4,0,0> LOCAL SYNCOPE:< lex form lex syncope > == C_DLT
(663)         GLOBAL R_yari:< lex form lex >
(664) =5,0,0> LOCAL C_DLT:< lex form lex || syncope > ==
                    I_AMPLE:< amp allomorph >
(665)         GLOBAL R_yari:< lex form lex >
(666) =6,0,0> LOCAL I_AMPLE:< amp allomorph || syncope > ==
```

```
        \a   "< allo form >"   "< allo prop >"
             "< allo sec >"   "< allo mec >"
(667)         GLOBAL R_yari:< lex form lex >
(668) =7,0,2> LOCAL R_yari:< || allo form syncope > == C_R_VR_MC1
(669)         GLOBAL R_yari:< allo form syncope >
(670) =8,0,0> LOCAL C_R_VR_MC1:< || allo form syncope > == C_R_VR
(671)         GLOBAL R_yari:< allo form syncope >
(672) =9,0,0> LOCAL C_R_VR:< || allo form syncope > == C_ROOT
(673)         GLOBAL R_yari:< allo form syncope >
(674) =10,0,0> LOCAL C_ROOT:< || allo form syncope > == C_MORPHEME
(675)         GLOBAL R_yari:< allo form syncope >
(676) =11,0,0> LOCAL C_MORPHEME:< allo || form syncope > ==
   ALLOMORPH
(677)         GLOBAL R_yari:< allo form syncope >
(678) =12,0,0> LOCAL ALLOMORPH:< allo form syncope > ==
                  Chop:< "< phon under >" >
(679)         GLOBAL R_yari:< allo form syncope >
(680) =13,1,0> LOCAL R_yari:< phon under > == yari
(681)         GLOBAL R_yari:< phon under >
(682) =13,0,0> LOCAL Chop:< yari > ==
                Implode:< Reverse:< Rest:
                < Reverse:< Explode:< $a  >  >  >  >  >
(683)         GLOBAL R_yari:< allo form syncope >
(684) =14,4,0> LOCAL Explode:< || yari > == ::Explode
(685)         GLOBAL R_yari:< allo form syncope >
(686) =14,3,0> LOCAL Reverse:< || y a r i > == ::Reverse
(687)         GLOBAL R_yari:< allo form syncope >
(688) =15,3,0> LOCAL ::Reverse:< y || a r i > == <  > $x
(689)         GLOBAL R_yari:< allo form syncope >
(690) =16,3,0> LOCAL ::Reverse:< a || r i > == <  > $x
(691)         GLOBAL R_yari:< allo form syncope >
(692) =17,3,0> LOCAL ::Reverse:< r || i > == <  > $x
(693)         GLOBAL R_yari:< allo form syncope >
(694) =18,3,0> LOCAL ::Reverse:< i > == <  > $x
(695)         GLOBAL R_yari:< allo form syncope >
(696) =19,3,0> LOCAL ::Reverse:< > ==
(697)         GLOBAL R_yari:< allo form syncope >
(698) =14,2,0> LOCAL Rest:< || i r a y > == ::Rest
(699)         GLOBAL R_yari:< allo form syncope >
(700) =15,2,0> LOCAL ::Rest:< i || r a y > == ::Idem:<  >
(701)         GLOBAL R_yari:< allo form syncope >
(702) =14,1,0> LOCAL Reverse:< || r a y > == ::Reverse
(703)         GLOBAL R_yari:< allo form syncope >
(704) =15,1,0> LOCAL ::Reverse:< r || a y > == <  > $x
(705)         GLOBAL R_yari:< allo form syncope >
(706) =16,1,0> LOCAL ::Reverse:< a || y > == <  > $x
(707)         GLOBAL R_yari:< allo form syncope >
(708) =17,1,0> LOCAL ::Reverse:< y > == <  > $x
(709)         GLOBAL R_yari:< allo form syncope >
(710) =18,1,0> LOCAL ::Reverse:< > ==
(711)         GLOBAL R_yari:< allo form syncope >
(712) =14,0,0> LOCAL Implode:< || y a r > == ::Implode
(713)         GLOBAL R_yari:< allo form syncope >
(714) =7,0,4> LOCAL R_yari:< || allo prop syncope > == C_R_VR_MC1
(715)         GLOBAL R_yari:< allo prop syncope >
(716) =8,0,0> LOCAL C_R_VR_MC1:< || allo prop syncope > == C_R_VR
(717)         GLOBAL R_yari:< allo prop syncope >
```

```
(718) =9,0,0> LOCAL C_R_VR:< || allo prop syncope > == C_ROOT
(719)         GLOBAL R_yari:< allo prop syncope >
(720) =10,0,0> LOCAL C_ROOT:< || allo prop syncope > == C_MORPHEME
(721)         GLOBAL R_yari:< allo prop syncope >
(722) =11,0,0> LOCAL C_MORPHEME:< allo || prop syncope > ==
   ALLOMORPH
(723)         GLOBAL R_yari:< allo prop syncope >
(724) =12,0,0> LOCAL ALLOMORPH:< allo prop || syncope > ==
(725)         GLOBAL R_yari:< allo prop syncope >
(726) =7,0,6> LOCAL R_yari:< || allo sec syncope > == C_R_VR_MC1
(727)         GLOBAL R_yari:< allo sec syncope >
(728) =8,0,0> LOCAL C_R_VR_MC1:< || allo sec syncope > == C_R_VR
(729)         GLOBAL R_yari:< allo sec syncope >
(730) =9,0,0> LOCAL C_R_VR:< || allo sec syncope > == C_ROOT
(731)         GLOBAL R_yari:< allo sec syncope >
(732) =10,0,0> LOCAL C_ROOT:< || allo sec syncope > == C_MORPHEME
(733)         GLOBAL R_yari:< allo sec syncope >
(734) =11,0,0> LOCAL C_MORPHEME:< allo || sec syncope > == ALLOMORPH
(735)         GLOBAL R_yari:< allo sec syncope >
(736) =12,0,0> LOCAL ALLOMORPH:< allo sec syncope > == "SEC_01"
(737)         GLOBAL R_yari:< allo sec syncope >
(738) =13,0,0> LOCAL SEC_01:< || allo sec syncope > == SEC
(739)         GLOBAL SEC_01:< allo sec syncope >
(740) =14,0,0> LOCAL SEC:< allo sec || syncope > ==
                    < mec environ >
(741)         GLOBAL SEC_01:< allo sec syncope >
(742) =15,0,0> LOCAL SEC:< mec environ > ==
                 Implode:< /  "< environ >"   >
(743)         GLOBAL SEC_01:< allo sec syncope >
(744) =16,1,0> LOCAL SEC_01:< environ > == _ [V]
(745)         GLOBAL SEC_01:< environ >
(746) =16,0,0> LOCAL Implode:< || /  _ [V]   > == ::Implode
(747)         GLOBAL SEC_01:< allo sec syncope >
(748) =7,0,8> LOCAL R_yari:< || allo mec syncope > == C_R_VR_MC1
(749)         GLOBAL R_yari:< allo mec syncope >
(750) =8,0,0> LOCAL C_R_VR_MC1:< || allo mec syncope > == C_R_VR
(751)         GLOBAL R_yari:< allo mec syncope >
(752) =9,0,0> LOCAL C_R_VR:< || allo mec syncope > == C_ROOT
(753)         GLOBAL R_yari:< allo mec syncope >
(754) =10,0,0> LOCAL C_ROOT:< || allo mec syncope > == C_MORPHEME
(755)         GLOBAL R_yari:< allo mec syncope >
(756) =11,0,0> LOCAL C_MORPHEME:< allo || mec syncope > == ALLOMORPH
(757)         GLOBAL R_yari:< allo mec syncope >
(758) =12,0,0> LOCAL ALLOMORPH:< allo mec || syncope > ==
(759)         GLOBAL R_yari:< allo mec syncope >
(760) [Query 1 (114 Inferences)] ENG_hit_O1s:< lex form lex > =
    \a   yari
    \a   yar     / _ [V]
    .
```

Although the above trace is quite lengthy, an understanding of it will tie things together for the reader, will illustrate how morphophonemic rules may be used to automatically generate allomorphs, and will set the stage for the next section. The trace shown above generates a

default allomorph and a syncope allomorph. Lines (601)-(659) apply to the generation of the default allomorph, and lines (660)-(759) apply to the generation of the syncope allomorph. Each of the two major sections is broken into five sub-sections. The first sub-section is the redirection of the query to the appropriate rule and the invocation of the AMPLE interface that handles allomorphs (e.g., lines (601)-(608)). The interface defines an allomorph field as consisting of a form, properties, SECs, and MECs. Each of these is a sub-section in trace above. There is a section to find the allomorph form (e.g., lines (609)-(623)), a section to find the allomorph properties (e.g., lines (624)-(635)), a section to find the allomorph SECs (e.g., lines (636)-(647)), and a section to find the allomorphs MECs (e.g., lines (648)-(659)).

DATR begins searching for the `<lex form lex>` in the node `ENG_hit_O1s`. Through the empty path it is sent up the inheritance hierarchy until it reaches the node `C_R_VR_MC1` (599). At this point DATR is told to search in both `DEFAULT` and `SYNCOPE`. These nodes are listed starting at (563) and (588). The path `<lex form lex>` in `C_R_VR_MC1` (405) is critical. It lists all allomorph rules that are to be applied to the underlying form of root nodes that have the property `MC1`. The reason the trace has two major sections (one for DEFAULT, one for SYNCOPE) is because of this line.

The discussion will now focus on how the default allomorph is generated ((601)-(659)). In the first sub-section (601)-(608), DATR starts in the node DEFAULT, where two things happen. First, the search path `<lex form lex>` is changed to `<lex form lex default>` (601). This will be important as the trace continues because the extension will direct DATR to paths that exactly apply to the default allomorph. The new search path, `<lex form lex default>`, is explicitly defined within the `DEFAULT` node, and results in a redirection of the search to the node `C_DLT`. When `C_DLT` is reached, a subtle transformation

occurs. Notice in (605) that only the `lex form lex` portion of the search path `<lex form lex default>` matched. This is shown by `<lex form lex || default>`. Watch what happens to the part that was left over (i.e., `default`). Line (605) directs DATR to look in `I_AMPLE:<amp allomorph>`, but when DATR gets to I_AMPLE, it is now looking for `<amp allomorph default>` (607). Again, this will cause DATR to go to paths that are specific to `default` whenever such specificity is required.

The portion of the AMPLE interface shown in (538) has to do with the generation of an AMPLE allomorph field. Note that this specification applies no matter what the morphophonemic rule is--the path `<amp allomorph default>` matched the shorter path `<amp allomorph>` (607). Note in (607) that each of the parts to an allomorph field are defined as quoted path names. Each path name results in a corresponding sub-section in the trace, as described above. Because the paths are quoted, DATR will use the global context (the node `R_yari` (608)) to satisfy the query. The path `"<allo form>"` results in lines (609)-(623), the path `"<allo prop>"` results in lines (624)-(635), the `path "<allo sec>"` results in lines (636)-(647), and the path `"<allo mec>"` results in lines (648)-(659).

Discussion will now turn to the form portion of the default allomorph. In lines (609)-(623), the following occurs. DATR attempts to find the path `<allo form default>` in the node `R_yari`, but ultimately through inheritance winds up at the top of the inheritance hierarchy at the node `ALLOMORPH`. There DATR is told to use `"<phon under>"` (620), which results in the default allomorph form being set to `yari` (622).

The default properties are found in lines (624)-(635). This is not very interesting, since it turns out the value is undefined for this allomorph. The same happens for the default SECs (636)-(647) and MECs ((648)-(659)). They are undefined for the default allomorph.

At this point discussion switches to the generation of the allomorph using the syncope rule (660)-(759). The trace progresses as before, and nothing new happens until we get to the syncope path in the node ALLOMORPH. (678). There DATR is again told to use the underlying form, but to apply a user-defined library function to the result: Chop:<"<phon under>">. In lines (680)-(713), the underlying form is found, then the Chop function is applied. The Chop function merely strips off the final character of a string. This is exactly what is required to apply the syncope rule to the underlying form of the node R_yari. So, the result is yar.

Because there are no properties or MECs defined for the syncope allomorphs, there is nothing interesting in those sections of the trace. However, in (726)-(747) we see the generation of the SEC environment '/ _ [V]'. This was described in the previous section on the use of variable-like nodes.

So far we have seen that in the case of those root nodes that have the property MC1, it is possible to generate two allomorphs by inheriting the morphophonemic rules named default and syncope. However, the templates section in the Ogea DATR theory file defines nodes for other morphophonemic rules. Specifically, the node ALLOMORPH contains all the information for each rule:

```
(761) ALLOMORPH:
(762) <allo form default> == "<phon under>"
(763) <allo form elision> == Chop:<"<phon under>">
(764) <allo form epenthetic_w> == Implode:<w "<phon under>">
(765) <allo form glide> == Implode:<Chop:<"<phon under>"> y>
(766) <allo form redup> ==  Implode:<First:<Explode:<"<phon
                              under>">> First:<Rest:<Explode:
                              <"<phon under>">>>>

(767) <allo form syncope> == Chop:<"<phon under>">
(768) <allo form n_b> == Implode:<Chop:<"<phon under>"> m>
(769) <allo form n_a> == Implode:<Chop:<"<phon under>"> n>
(770) <allo form n_x> == Implode:<Chop:<"<phon under>"> ng>
(771) <allo form n_v> == Chop:<"<phon under>">
(772) <allo prop> ==
(773) <allo mec> ==
```

```
(774) <allo sec> ==
(775) <allo sec epenthetic_w> == "SEC_02"
(776) <allo sec glide> == "SEC_01"
(777) <allo sec syncope> == "SEC_01"
(778) <allo sec n_b> == "SEC_04"
(779) <allo sec n_a> == "SEC_05"
(780) <allo sec n_x> == "SEC_07"
(781) <allo sec n_v> == "SEC_06".
```

The reader should now be in a better position to understand the structure of the node ALLOMORPH. There are sets of paths for forms, properties (not used in Ogea, so undefined), SECs, and MECs. Within each set, a path extension is provided for each type of rule. The morphophonemic rules for Ogea described in Appendix 1 are all covered by the node ALLOMORPH. There are rules for elision, epenthesis, glide, reduplication of the initial consonant and vowel, syncope, and nasalization. The nasalization rules are split based on the point of articulation of the segment following the nasal. The functions Chop, Explode, First, Implode, and Rest are applied to the underlying form to obtain the proper form. Chop was discussed above. Explode splits a string into individual characters, Implode concatenates characters or strings, First provides the first character from a string, and Rest strips off the first character of a string and returns the rest of the string. Except for Chop, these functions all belong to the standard DATR function library. This author could not find a function to chop off the final character of a string, so a user-defined chop function was written.

This section has shown the capture of generalizations and the reduction of redundancy through the automated generation of allomorphic forms by application of morphophonemic rules applied to the underlying form of a lexeme. In addition to the capture of generalizations and reduction of redundancy, the use of morphophonemic rules to generate allomorphs provides a linguist with the opportunity to test the accuracy of the

morphophonemic rules hypothesized for a language. If the allomorphs are generated directly by rules, the results can be checked against actual data to verify that the rules are correctly stated.

3.5.4 Reduction of Redundancy

Although many examples have been presented to demonstrate that generalizations can indeed be captured in DATR that cannot be captured in AMPLE, and that redundancy can be reduced, is the amount of reduction significant? For example, not all allomorphs may necessarily be capable of generation from a rule applied to an underlying form, and not all lexemes will have features that may be generalized and placed in an abstract node. More about this will be said in later. However, at this point the question before us is this: can data be presented that will satisfy proof criterion 4 for H_4? That is, for some language specific DATR lexicon, is it true that $I / A * 100 \geq 5$, where *I* is the count of all inheriting nodes (lexical entries that inherit information from another node), and *A* is the count of all nodes (all lexical entries). This can be demonstrated in the case of the Ogea DATR lexicon. As discussed in Chapter 2, inheritance in DATR may occur through the use of a named node, path, or combination of node and path. For example the node `S_tigi` inherits from `C_S_OBJECT_MC1` via the empty path (791), and inherits two `<lex gloss eng>` paths from `ENG_O2p` (793):

```
(782) C_S_OBJECT_MC1:
(783) <> == C_S_OBJECT
(784) <lex form> == DEFAULT SYNCOPE
(785) <lex prop> == 'MC1'.

(786) ENG_O2p:
(787) <> == "S_tigi"
(788) <lex gloss eng morname> == 'O2p'
(789) <lex gloss eng> == 'to you (plural)'.
```

```
(790) S_tigi:
(791) <> == C_S_OBJECT_MC1
(792) <phon under> == tigi
(793) <lex gloss eng> == ENG_O2p.
```

It turns out that in the case of the Ogea DATR lexicon, A = 911, I = 911, and I / A * 100 = 100. This means that the proof criterion for H_4 has been easily met. However, this is not a very satisfying situation. Although all nodes in the Ogea DATR theory file make use of inheritance, this tells us nothing about the degree to which redundancy was reduced.

One way to measure the reduction in redundancy is to count the number of occurrences of each field type in the original Ogea AMPLE database file, and then to count the number of occurrences of the corresponding information in the root and suffix nodes of the Ogea DATR theory file. This was done programmatically[60] with the following results shown in Table 3.

Although 100% of nodes use inheritance in the Ogea DATR lexicon, the actual reduction of redundancy varies depending on what type of information is involved. English glosses and underlying forms occur in all nodes. No reduction is possible. However, both the record type and the morpheme property were totally inherited--these were specified only in abstract nodes. The percent of reduction for MECs was significantly less than that of SECs. Of the 2,308 pieces of information counted in the Ogea AMPLE database file, only 830 of those pieces were not eliminated from the lower level root and suffix nodes through inheritance and the use of abstract nodes. Overall, there was a 64% reduction in redundancy

---

[60] See the two Perl scripts AMPStat.pl and DATStat.pl in Appendix 5. Some adjustments had to be made. The Ogea DATR theory files uses a separate set of English nodes that are not in the Ogea AMPLE database file. These were not actually required, but set up to demonstrate multi-language entry points into the theory file. The count of English glosses and morph names come from the English nodes. There are 280 AMPLE records, 280 English nodes, and 374 Ogea nodes. Extra Ogea nodes were built to support queries against homophonous allomorphs. Thus, the actual number of underlying forms is higher than it should be. Therefore the count was set to 280, since every DATR entry that corresponds to an AMPLE record must have an underlying form. No node inherits the underlying form.

Table 3 Reduction of Redundancy in Ogea Lexicon By Use of DATR

Information Type	AMPLE	DATR	Reduction[61]
Allomorph form	423	38	91.02%
Allomorph SEC	178	35	80.34%
Allomorph MEC	95	62	34.74%
Category	280	12	95.71%
English Gloss	280	280	0%
Morph Name (Gloss)	280	116	58.57%
Morpheme Property	130	0	100%
Order Class	82	11	86.59%
Type	280	0	100%
Underlying form	280	280	0%
Totals	**2,308**	**830**	**64.04%**

based on this method of measurement.

As for the Yalálag Zapotec lexicon, only a subset of the AMPLE root lexicon was used. The subset included all Order Class 10 records, and four Order Class 9 records (though by manual examination, the generalizations made for Order Class 9 were verified to hold for all Order Class 9 records). Comparing the pieces of information encoded in root records for the subset AMPLE lexicon versus the DATR version, the results are shown in Table 4.

There are a number of things to note about the results for Yalálag Zapotec. First, an information type had to be added to the DATR version, namely the lexeme name. This was because the AMPLE root field marker values for Yalálag Zapotec sometimes contained characters that are reserved in DATR. For example, the marker record marker `\r cha'ra'll` cannot be encoded in DATR as `R_cha'ra'll` because DATR interprets the single quotes as the start and stop of a string literal. Therefore it was necessary to create a

[61] Calculated as follows: 1-(AMPLE/DATR).

path <lex name> to hold the correct output for the AMPLE field marker values. This resulted in a 100% increase in this particular information type when the DATR lexicon is

Table 4. Reduction of Redundancy in Yalálag Zapotec Lexicon by Use of DATR[62]

Information Type	AMPLE	DATR	Reduction[63]
Allomorph form	50	26	48.00%
Allomorph SEC	45	5	88.89%
Allomorph MEC	31	1	96.77%
Category	24	0	100.00%
Comment	2	2	0%
Lexeme Name	0	20	-100%
Morpheme MCC	1	1	0%
Morph Name (Gloss)	24	24	0%
Morpheme Property	76	9	88.16%
No-Load	24	24	0%
Order Class	24	1	95.83%
Underlying form	24	24	0%
Totals	**325**	**137**	**57.85%**

compared to the AMPLE one. There were three information types that were not reduced in the DATR lexicon: the gloss, the no-load, and the underlying form. However, it would likely be possible in DATR to eliminate the no-load path. The Yalálag Zapotec AMPLE root database file uses the no-load field to encode a numeric identifier for the record. This could likely be generated by DATR during run-time, which would eliminate it as a path in the root DATR nodes. Note that there was a 100% reduction in category, and a near 100% reduction for the MEC and the Order Class. The Order Class should have been 100%, but for the record `\r nnalÄllej`, the order class field has a comment, so the DATR version included a <lex order> path so the comment could be captured. Taking into account the extra

[62] Only a subset of the entire AMPLE root file was used. All Order Class 10 records and four Order Class 9 records were used.
[63] Calculated as follows: 1-(AMPLE/DATR).

information required to encode the lexeme name, the overall reduction of redundancy for the Order Class 10 records and the four Order Class 9 records was 57.85%.

3.5.5 Summary

This major section addressed the hypotheses:

> H_3: There are generalizations not possible to capture in the AMPLE legacy LKRL that can be captured by use of DATR's inheritance mechanism.
>
> H_4: Such generalizations can be exploited by 5% or more of lexical entries in the lexicon for a specific language.

H_3 is considered true since per proof criterion 3, examples were shown from AMPLE lexicons for at least two languages (Yalálag Zapotec and Ogea) where lexical entries contained information that could be generalized and DATR code was presented showing how the generalizations may be captured. H_4 is considered true since per proof criterion 4, for the Ogea DATR lexicon, $I / A * 100 \geq 5$, where *I* is the count of all inheriting nodes (lexical entries that inherit information from another node), and *A* is the count of all nodes (all lexical entries). In the case of Ogea, the percentage of inheriting nodes is in fact 100%. It was also shown that redundancy in the Ogea lexicon was reduced by 64% when the lexicon was converted from AMPLE to DATR, and by 57.85% for Yalálag Zapotec.

3.6 An Interface Between DATR and AMPLE

This section addresses the hypothesis:

> H_2: It is possible to translate AMPLE-oriented DATR lexicons back into AMPLE legacy format for use by AMPLE.

Per proof criterion 2, H_2 will be considered true if it can be shown that a usable AMPLE version of a dictionary database can be generated from a DATR version. By *usable* it is meant that the generated AMPLE file(s) contain the lexical input required to correctly parse the words of a language.

This section presents proof for H_2 from the Ogea DATR theory file. First the interface itself is presented and described. Then the process of verification of the interface will be described.

3.6.1 Description of the Interface

The interface between DATR and AMPLE used for the Ogea lexicon is found in the Interface section of the Ogea DATR theory file listed in Appendix 3. The interface is listed below for the reader's convenience:

```
(794) I_AMPLE:
(795) % Generates records with English as first field
(796) <amp lex eng> ==     <amp gloss eng>
(797)                  <amp gloss morpheme name>
(798)                      <amp gloss national>
(799)                      <amp rest>
(800) % Generates records with the national language as first field
(801) <amp lex nat> ==     <amp gloss national>
(802)                      <amp gloss morpheme name>
(803)                      <amp gloss eng>
(804)                      <amp rest>
(805) % Generates records with the vernacular language as first
   field
(806) <amp lex ver> ==     <amp gloss vernacular>
(807)                      <amp gloss morpheme name>
(808)                      <amp gloss eng>
(809)                      <amp gloss national>
(810)                      <amp rest>
(811) % Generates records with the morphname as the first field
(812) <amp lex morname> == <amp gloss morpheme name>
(813)                      <amp gloss eng>
(814)                      <amp gloss national>
(815)                      <amp rest>

(816) <amp rest> ==        <amp type>
(817)                      "<lex form lex>"
(818)                      <amp underlying form>
(819)                      <amp allo property>
(820)                      <amp order class>
(821)                      <amp category>
(822)                      <amp elsewhere allomorph>
(823)                      <amp mcc>

(824)                      <amp features>
(825)                      <amp infix location>
(826)                      <amp noload>
(827)                      <amp comment>'\n'
```

```
(828) <amp allomorph> == \a' ' "<allo form>"' ' "<allo prop>"' '
                "<allo sec>"' ' "<allo mec>"'\n'
(829) <amp category> == \c' ' "<lex cat>"'\n'
(830) <amp comment> == \co' ' "<lex com>"'\n'
(831) <amp elsewhere allomorph> == \e' ' "<lex else allo>"'\n'
(832) <amp features> == \f' ' "<lex feat>"'\n'
(833) <amp gloss eng> == \ge' ' "<lex gloss eng>" '\n'  % not called
                                since used as mor name
(834) <amp gloss vernacular> == \gv' ' "<lex under>" '\n'
(835) <amp gloss national> == \gn' ' "<lex gloss nat>" '\n'
(836) <amp infix location> == \loc' ' "<lex loc>" '\n'
(837) <amp gloss morpheme name> == \g' ' "<lex gloss eng morname>"
   '\n'
(838) <amp allo property> == \mp' ' "<lex prop>"'\n'
(839) <amp noload> == \_no' ' "<lex noload>"'\n'
(840) <amp order class> == \o' ' "<lex order class>"'\n'
(841) <amp mcc> == \mcc' ' "<lex mcc>"'\n'
(842) <amp type> == \type' ' "<lex type>"'\n'
(843) <amp underlying form> == \u' ' "<lex under>"'\n'.
```

There are six major sections to the interface. In lines (795)-(814) there are a series of path definitions that control which field will be the first field in each AMPLE record. There are four choices: English, the national language, the vernacular language, and the morpheme name. Of course, as far as AMPLE is concerned, no matter which field occurs first, it is the morpheme name that uniquely identifies the record, and the first field of the record must be identified to AMPLE as the record identifier. Notice that in each of the four choices of which field comes first, the right-hand path list ends with the path `<amp rest>`. This path is defined in lines (816)-(827). The path `<amp rest>` provides a list of the additional parts to an AMPLE record. Note that the order in which the listed parts occur is the order in which the fields will occur in the generated AMPLE record. Also note that there is a quoted path that occurs--`"<lex form lex>"`. This is the path that generates allomorphs. Finally, note the quoted formatting code `'\n'` that will result in a line break between the result of each query. Except for the quoted path `"<lex form lex>"`, all other paths defined on the right-hand side of the path `<amp rest>` are found in the same node, i.e. `I_AMPLE`.

The last section of I_AMPLE contains a list of 16 paths. Each of these paths contains an AMPLE field marker, a quoted path name, and formatting codes. In the case of <amp allomorph> there are multiple path names on the right-hand side, one for each component of an AMPLE allomorph field. The quoted paths direct DATR to use the global context for its search. As will be seen, one problem with using ZDATR to generate AMPLE records is that ZDATR wants to put spaces between the results of each part of a query and a period at the end of each query. There is a command line switch to suppress the spaces, but no switch to suppress the period at the end of each query. Notice that each of the 16 paths contains single quotes with a space between them. This is necessary to put spaces in where needed since spaces have been suppressed via the command line. If the spaces are not suppressed, the result is spaces being inserted before each field marker. AMPLE does not allow a field marker to begin with a space.

An example will now be given, using the interface and the following nodes, some of which have been seen before:

```
(844) C_MORPHEME:
(845) <> ==
(846) <amp lex> == I_AMPLE
(847) <dict lex> == I_DICT
(848) <allo> == ALLOMORPH
(849) <lex under> == "<phon under>"
(850) <lex prop> ==
(851) <lex noload> ==
(852) <lex order class> ==
(853) <lex form> == DEFAULT
(854) <lex cat> ==
(855) <lex loc> ==
(856) <lex gloss nat> == 'No national language entry found'
(857) <lex gloss eng> == 'No English entry found'
(858) <lex gloss eng morname> == 'No English morph name found'.

(859) C_ROOT:
(860) <> == C_MORPHEME
(861) <lex type> == r.
```

```
(862) C_R_VR:
(863) <> == C_ROOT
(864) <lex prop> == 'MC2'
(865) <lex cat> == 'VR'.
(866)
(867) ENG_shove:
(868) <> == "R_nuu"
(869) <lex gloss eng> == 'shove'.
(870)
(871) R_nuu:
(872) <> == C_R_VR
(873) <phon under> == nuu
(874) <lex gloss eng> == ENG_shove.
```

Using the interface listed above, and the nodes listed above, the query `ENG_shove:<amp lex morname>` results in the following:

```
(875)  \g shove
(876)  \ge shove
(877)  \type r
(878)  \a nuu
(879)  \u nuu
(880)  \mp MC2
(881)  \o
(882)  \c VR
(883)  \e
(884)  \mcc
(885)  \f
(886)  \loc
(887)  \_no
(888)  \co
(889)
(890)  .
```

For comparison, here is the original AMPLE record:

```
(891) \g shove
(892) \ge shove
(893) \type r
(894) \a nuu
(895) \u nuu
(896) \mp MC2
(897) \c VR
```

The entry point to the query `ENG_shove:<amp lex morname>` is the node `ENG_shove` (867). Note that the query path `<amp lex morname>` does not exist in this node. Through inheritance, the path is eventually found in the interface node `I_AMPLE`. See in order, the following lines: (868), (872), (863), (860), (846). The result is that the morpheme name is

the first field in the output. Also notice that the I_AMPLE interface uses a unified dictionary, where instead of separate prefix, root, infix, and suffix database files, a single file is used with a `\type` field whose value indicates whether the record is for a prefix, root, infix, or suffix. In this case, the identification of the record type as a root was made by inheritance through the node `C_ROOT`. Notice in (867) that the node `ENG_shove` only specifies an English gloss, not a morph name. Through inheritance, if the English gloss and the morpheme name are identical, only the English gloss needs to be explicitly specified.

Finally, note that the result of a query is not directly usable by AMPLE. There are empty fields, a blank line, and a period. These must be removed by an external program, such as the Perl scripts shown in Appendix 5. (There probably is a way in DATR to suppress the empty fields, but this author has not yet figured it out. At any rate, there would still be a need for an external program to do a final cleanup of the DATR output.)

All other mechanisms used to generate the AMPLE record from DATR have been explained in previous sections. Discussion will now turn to how the interface was verified.

3.6.2 Executing the Interface

In order to generate an Ogea AMPLE database file from DATR, a batch file is run that instructs DATR to run a query file against the Ogea DATR theory file. The query file contains a query for each AMPLE record to be generated. The order of the generated records is a reflection of the order of the queries. The output of running the query file against the theory file is a DATR trace file. The trace file is processed by a Perl script to create the final Ogea AMPLE file by removing blank fields and periods inserted by DATR. The batch file *go.bat*, query file *ogea.qry*, and theory file *ogea.dtr* are listed in Appendix 3. The Perl script *DToDATR.pl* is listed in Appendix 5.

3.6.3 Verification of the Interface

Per the methodology defined in Chapter 1, verification of the interface occurs in two ways. First, the AMPLE file generated from DATR must be compared to the original AMPLE file to ensure that they are identical. Second, text must be parsed by AMPLE, first using the original AMPLE database file, then the version generated from DATR. The outputs of the two runs must be identical.

The code to compare the original Ogea AMPLE database file to the one generated by DATR is presented in Appendix 5 in the Perl script *compare.pl*. This Perl script normalizes the inputs by removing all extra spaces from the input of both files and by removing from the generated version all empty fields and periods inserted by DATR. The two files are compared line by line, using the normalized input. When the script is run, the two files are found to be identical[64].

The Perl script *C2.pl*, listed in Appendix 5, compares two AMPLE interlinear files. The interlinear files are the result of running AMPLE in CARLA Studio, using the file *list.txt* as the text file to be parsed. First AMPLE is run using a hand-created Ogea AMPLE database file *original.db*, which is copied to *ogea.db* for the run. The resulting interlinear file, *list.itx* is renamed (by hand) to *old.itx*. Next, the DATR to AMPLE interface is executed using the batch file *go.bat* (listed in Appendix 3), which creates a new version of *ogea.db*, generated from DATR (see Appendix 3 for the listing). AMPLE is run again, and this time the resulting interlinear file is named *new.itx*. Finally, the Perl script *C2.pl* is run. When this

[64] They were not identical the first time the comparison was run. This was due to errors in the interface and inconsistencies in the original AMPLE database file. The original file had to be corrected. For example, the order of fields was inconsistent across records. This had to be fixed in order for an automated comparison to work.

process was followed by this author, the Perl script verified that the interlinear files are identical.

3.7 Generating Other Lexicons from DATR

It is possible in DATR to not only create an interface to AMPLE, but also to create other interfaces. Thus, a single lexicon can be used to generate other lexicons and dictionaries, with format tailored for a specific purpose or requirement. In order to illustrate this, this author added a second interface in the interface section of the Ogea DATR theory file (see Appendix 3).

```
(898) I_DICT:
(899) <dict lex eng> ==
  "<lex gloss eng>"'\n'
(900)    '  Ogea:          ' "<lex under>"'\n'
(901)    '  Category:      '"<lex cat>"'\n'
(902)    '  Property:      ' "<lex prop>"'\n'
(903)    '  Allomorph(s): \n'
(904)    "<lex form dict>"'\n'
(905) <dict allomorph> ==
  '                 '"<allo form>"'\n'
(906)    '                   Property:     ' "<allo prop>"'\n'
(907)    '                   String Environment:   ' "<allo sec>"'\n'
(908)    '                   Morpheme Environment:   ' "<allo
   mec>"'\n'.
```

Using the dictionary interface, the query `ENG_hit_O1s:<dict lex eng>` results in:

```
(909) hit me
(910)    Ogea:          yari
(911)    Category:      VR
(912)    Property:      MC1
(913)    Allomorph(s):
(914)                   yari
(915)                     Property:
(916)                     String Environment:
(917)                     Morpheme Environment:
(918)                   yar
(919)                     Property:
(920)                     String Environment:   / _ [V]
(921)                     Morpheme Environment:
```

The point here is that once the basic lexicon has been built using DATR, there are any number of ways other lexicons or formats may be generated.

3.8 Handling Many-to-Many and One-to-Many Relationships

There are a number of situations in working with lexicons--especially multi-lingual ones--where many-to-many and/or one-to-many relationships may arise. For example, a particular surface form may be homophonous. Since DATR requires node names to be unique, various ad hoc techniques must be used to create a unique name. Take for example the Ogea suffix *ge*, that occurs both as a verbal suffix indicating contrafaction and as an inalienably possessed noun suffix indicating first person plural. It is not possible in DATR to have two nodes, both with the name `S_ge`. Instead, different node names are needed:

```
(922) S_ge_1:
(923) <> == C_S_CONTRA
(924) <phon under> == ge
(925) <lex gloss eng> == ENG_Contra
(926) <lex form> == DEFAULT
(927) <lex order class> == 100
(928) <lex cat> == 'Contra/Contra'
(929) <allo mec> == "MEC_25".
(930)
(931) S_ge_2:
(932) <> == C_S_PRONOUN_POSS
(933) <phon under> == ge
(934) <lex gloss eng> == ENG_P1p.
```

There is no problem with this, per se, however, if someone is not aware that there are two homophonous suffixes /ge/, if they attempt a query using S_ge, the query would fail. There is a one-to-many relationship between *ge* and nodes with that surface form. In order to support this one-to-many relationship, it is possible to construct a node `S_ge`, that points to the two homophones:

```
(935) S_ge:
(936) <> == "S_ge_1" "S_ge_2".
```

With this arrangement, the query `S_ge:<amp lex morname>` will return two AMPLE records, one for each homophone:

```
(937) \g Contra
(938) \ge Contrafactual
(939) \type s
(940) \a ge   +/ [Contra.Subj] _
(941) \u ge
(942) \mp
(943) \o 100
(944) \c Contra/Contra
(945) \e
(946) \mcc
(947) \f
(948) \loc
(949) \_no
(950) \co
(951)
(952) \g P1p
(953) \ge first plural possessive
(954) \type s
(955) \a ge
(956) \u ge
(957) \mp
(958) \o 10
(959) \c PNR/PN
(960) \e
(961) \mcc
(962) \f
(963) \loc
(964) \_no
(965) \co
(966)
(967) .
```

A similar situation occurs when dealing with a multi-lingual lexicon. For example, a particular English word might have many Ogea words of which it is the translation, and vice versa. English nodes could point to multiple vernacular nodes and vice versa. For example, the Ogea word /afo/ means either 'leg' or 'foot' in English. Therefore, the existing Ogea node `R_afo` could be modified so that the English gloss path points not to the node `ENG_leg/foot`, but to two English nodes `ENG_leg` and `ENG_foot`:

```
(968) R_afe:
(969) <> == C_R_PNR
(970) <phon under> == afe
(971) <lex gloss eng> == ENG_leg ENG_foot
(972) <lex gloss nat> == TP_lek.
```

3.9 Some Issues and Questions

3.9.1 Factors Affecting the Decrease of Redundancy

The results presented for Ogea regarding the degree to which redundancy was reduced by use of DATR over AMPLE will not be the same for other languages, and may not even be the same for lexicons built by other linguists working on Ogea. There are least two major factors that affect the degree to which redundancy is reduced. First, all languages vary in what information is regular, and therefore predictable. Second, the degree to which redundancy is reduced is dependent on the skill of the linguist. Not all people have equal skill in analyzing a language or in inventing new techniques to use in DATR to help reduce redundancy in the lexicon. For these two reasons, the results will vary across languages and linguists. Needless to say, there is enough predictable information in natural languages that there will likely always be the potential for significant reductions.

3.9.2 Ease of Use Issues

The question must be asked--is it easier to build a lexicon in AMPLE than in DATR? Generally speaking, it is probably easier to build a lexicon using AMPLE than it is to build one using DATR. Building an AMPLE-based lexicon does not require analysis of the records to develop the kind of abstractions one would desire to encode in DATR. Also, in AMPLE, it is not necessary to use single quote marks around strings to manage the name space. The fact that DATR reserves characters that may occur in an orthography (i.e., the single quote mark) makes life difficult for linguistics using special orthographies. There are work-arounds, of course, but these problems do make things more difficult for the builder of a lexicon in DATR.

Once the lexicon has been built, is it easier to maintain in AMPLE or in DATR? Assuming that a significant amount of information is stored in abstract nodes, and assuming that the information being added, changed, or deleted takes advantage of abstract nodes, a lexicon built using DATR will be easier to maintain than one built using AMPLE. Is it easier to capture linguistic generalizations in AMPLE or in DATR? It appears to be easier to capture linguistic generalizations in DATR than in AMPLE. However, the choice between AMPLE and DATR will likely depend on the goals and skills of the linguist and the purpose for which the lexicon is being built.

3.9.3 General Versus Linguistically Motivated Abstractions

There can be two goals in creating abstract nodes from which more concrete lexeme nodes inherit information. One goal might be to simply make maintenance of the lexicon easier. Another goal might be to create and test linguistic models of the language. It should be kept in mind that it is entirely possible to find co-occurring pieces of information that may be moved into an abstract node, but the builder of the lexicon might fail to determine a linguistic motivation for that abstraction. If one's goal is to ease maintenance, then any abstraction will do. But if one's goal is to build a linguistic model, and if one cannot determine the linguistic basis for an abstraction, one might not create an abstract node even if co-occurring information has been found.

With these tensions in mind, this author would like to suggest that in the analysis of existing AMPLE lexicons for conversion to DATR, some general principles of programming and logical database design apply. The principles can be used to abstract information, whether such abstractions are linguistically motivated or not. First, as shown in Chapter 3, it is possible to make an abstract node that acts like a variable. This need not be discussed

further. Second, in logical database design, a process known as normalization is often applied to a logical data model. Typically a logical data model is analyzed and modified to conform to what is known as third normal form. (There are forms beyond third normal form, but practitioners usually stop with third normal form). This process is based on principles originally developed by E. F. Codd [26:99]. If a data model is in first normal form, no repeating fields[65] occur in any record type. The fact that AMPLE allows multiple allomorph records and multiple morpheme property fields violates first normal form. For example, the following record, valid in AMPLE, has two `\a` fields:

```
(973) \g come.down
(974) \ge come.down
(975) \type r
(976) \a me
(977) \a m / _ [V]
(978) \u me
(979) \mp MC1
(980) \c VR
```

One of the techniques presented by this author to handle multiple occurrences of allomorphs was to have multiple paths, but with different extensions:

```
(981) R_hit_O1s:
(982) <lex type> == r
(983) <lex gloss eng> == 'hit me'
(984) <lex gloss nat> == 'paitim mi'
(985) <lex gloss eng morname> == 'hit.O1s'
(986) <lex cat> == 'VR'
(987) <lex under> == yari
(988) <lex prop> == 'MC1'
(989) <lex order class> == 0
(990) <lex form> == <allo 1> '\n' <allo 2>
(991) <allo 1> == 'yari / _ [C]'
(992) <allo 2> == 'yar / _ [V] '.
```

Following standard practices in the design of logical data models, a more attractive alternative would be to create a separate node for each allomorph:

---

[65] Data modeling uses different terminology for logical data models: record types are called entities, and fields are called attributes.

```
(993) R_hit_O1s:
(994) <lex type> == r
(995) <lex gloss eng> == 'hit me'
(996) <lex gloss nat> == 'paitim mi'
(997) <lex gloss eng morname> == 'hit.O1s'
(998) <lex cat> == 'VR'
(999) <lex under> == yari
(1000)      <lex prop> == 'MC1'
(1001)      <lex order class> == 0
(1002)      <lex form> == R_hit_O1s_1 R_Hit_O1s_2.
(1003)
(1004)      R_hit_O1s_1:
(1005)      <> == R_hit_O1s
(1006)      <allo form> == 'yari'
(1007)      <allo sec> == '/ _ [C]'.
(1008)
(1009)      R_hit_O1s_2:
(1010)      <> == R_hit_O1s
(1011)      <allo form> == 'yar'
(1012)      <allo sec> == '/ _ [V] '.
```

Second normal form is likely not applicable to our discussion. If a data model is in second normal form, no partial key dependencies occur. This requires a record that uses a composite key as the primary key--that is a primary key that concatenates two or more fields to form a key. However, it could be argued that in its effect, the separate nodes in (1004) and (1009) have a primary key that is like a concatenation of the primary key of the parent node plus a sequence number (e.g., `R_hit_O1s` + '`1`').

If a data model is in third normal form, no transitive relationships occur. That is, there are no field values that depend on the value of another non-primary key field in the same record. Another way to state this is that non-key field values should not co-vary within the same record. In such cases, one value may be dependent on the other, or the co-occurring values may in fact be dependent on some other value not present in the record. Application of the principle of third normal form to the analysis of AMPLE records can reveal the fields whose values may be inherited from an abstract node. In the case of Yalalog Zapotec, it was shown that for Order Class 9 and 10 verbs, the values of certain fields co-vary. The value for the morpheme property field ('class2') and the value for the category field ('VA') was

unchanging across these verbs, indicating that either the value of one was dependent on the other, or both were dependent on a third element. By positing an abstract node V09_10, these field values were abstracted from the lower level root nodes, and the values became dependent on a third element, namely the node name itself (which functions as the primary key of a node in DATR).

```
(1013)      V09_10:
(1014)      <> == C_ROOT
(1015)      <lex cat> == 'VA'
(1016)      <lex prop> == 'class2 ' "<lex add prop>".
```

3.10 Summary

This chapter discussed the author's research on the use of DATR with AMPLE within the parameters defined by the research hypotheses. The chapter began by describing the foundational work that was done as the basis of the research. The discussion then switched to how AMPLE lexical data may be encoded in DATR. Subsequent sections in the chapter discussed how generalizations may be captured in DATR, how the author's interface between DATR and AMPLE works, and how other kinds of lexicons may be generated from the base DATR lexicon. The discussion was centered around the research hypotheses. The chapter mostly focused on AMPLE and DATR lexicons built to support morphological parsing of data from the Ogea language. However, the Yalálag Zapotec language of Mexico was also used for some of the discussion, specifically in the section 3.5 Capturing Generalizations and Reducing Redundancy with DATR. The chapter also addressed other topics beyond the hypotheses: how to handle many-to-many relationships in the lexicon, factors that affect the decrease in redundancy, ease of use issues, and general versus linguistically motivated abstractions.

CHAPTER 4

CONCLUSION

4.1 Research Results

This author's research investigated the feasibility and desirability of using DATR with AMPLE, a legacy morphology exploration tool developed by the Summer Institute of Linguistics (SIL) [6, 9, 70], the world's largest linguistic organization. The research demonstrated the feasibility of using DATR with AMPLE by showing how DATR can be used to encode the lexical information required by AMPLE and by presenting an interface between AMPLE and DATR-based lexicons that does not require modification of AMPLE itself. The desirability of using DATR with AMPLE was shown by demonstrating that there are generalizations that cannot be captured using AMPLE's lexical knowledge representation language (LKRL) that can be captured in DATR, thereby reducing redundancy of lexical information.

The research centered around the following hypotheses:

Feasibility

H_1: All types of lexical information expressible in the AMPLE legacy LKRL are also expressible in DATR.

H_2: It is possible to translate AMPLE-oriented DATR lexicons back into AMPLE legacy format for use by AMPLE.

Desirability

H_3: There are generalizations not possible to capture in the AMPLE legacy LKRL that can be captured by use of DATR's inheritance mechanism.

H_4: Such generalizations can be exploited by 5% or more of lexical entries in the lexicon for a specific language.

This author's research demonstrated the truth of the hypotheses. In order to prove these hypotheses, this author analyzed the morphophonology and morphotactics of a Papuan language, Ogea, and built a comprehensive AMPLE unified database lexicon that supports the parsing of every example found in the author's write-up of Ogea. Next, the author developed a DATR-based version of the lexicon, as well as an interface to generate the AMPLE lexicon from DATR. It was found that linguistic generalizations not possible in AMPLE's LKRL could be made in DATR. Through these generalizations, redundancy in the Ogea AMPLE lexicon was reduced by 64% in the DATR version of the same lexicon. In the case of Yalálag Zapotec, the number of redundant pieces of information in the subset AMPLE lexicon was reduced 57.85% in the DATR version. The validity of the interface from the DATR lexicon to the AMPLE lexicon was verified by programmatically comparing the original version to the generated one. Also, a text file was parsed in AMPLE using both versions of the Ogea AMPLE database file, and the results were programmatically verified to be identical. An AMPLE root database file for a second language, Yalálag Zapotec, was also partially analyzed by the author. It was demonstrated that for that language also, generalizations not possible to make with the AMPLE LKRL could be made in DATR.

4.2 Future Research

This author’s research has demonstrated both the feasibility and desirability of using DATR to encode lexical information, and to generate AMPLE dictionary files from DATR lexicons. However, there are areas on which future research might focus. First, the DATR to AMPLE interface developed by this author for the Ogea language should be abstracted to make it more generally useful to other languages. This should be done by developing a DATR version of AMPLE database files for a minimum of two additional languages.

Complete dictionary files for each language should be converted to a DATR version in order to provide adequate breadth and depth. The interface can be placed in a separate file or files, and shared with language specific DATR theory files via the DATR `#load` directive. Those portions of the interface that are truly language independent could be placed in one file, and those portions that might or should be tailored for specific languages could be placed in yet another file. The abstracted interface should then be made available to the linguistic community for use with other languages and the interface should subsequently be modified based on feedback from actual usage.

Because AMPLE is often used for computer-aided language adaptation (CARLA), future work should also focus on developing techniques and examples of multi-lingual DATR lexicons of related languages that generalize cross-linguistic information to abstract nodes. These DATR-based lexicons could be used to generate the corresponding AMPLE lexicons for use in CARLA. Such a multi-lingual lexicon would be similar to the English-Dutch-German lexicon being developed by Cahill and Gazdar [18].

The use of DATR itself could be made easier through the development of a graphical-user-interface front-end to DATR. Perhaps a node browser could be developed analogous to class browsers currently available in modern integrated development environments (IDEs) for object-oriented programming languages. A DATR IDE could also provide lists of a node's paths that pop up when the node name has been written in another node's path statement. This is analogous to the help offered in Microsoft's Visual Basic, that displays a pop-up of the methods and properties available through a particular object when the object name has been typed as part of a line of code.

Lastly, DATR should be considered by the information systems community for the maintenance of data dictionaries. This community has often been confronted with the problem of combining databases from multiple sources into a new database. Record names, field names, formats, etc. are often found to vary for what should be the same pieces of information. DATR could be used to develop a unified data dictionary that also preserves database specific names, layouts, formats, and definitions.

4.3 Final Thoughts

This author found both AMPLE and DATR to be powerful tools, and intends to use both for his continuing work on the Ogea language. That is, the advantages to using these tools, and to using DATR to build and maintain lexicons, are not just academic. These tools have pragmatic value to those working on the analysis of languages around the world.

BIBLIOGRAPHY

[1] F. Andry, N. M. Fraser, S. McGlashan, S. Thornton, and N. J. Youd, "Making DATR Work for Speech: Lexicon Compilation in SUNDIAL.," *Computational Linguistics*, vol. 18, pp. 245-267, 1992.

[2] E. Antworth and S. McConnel, "PC-Kimmo Reference Manual--A Two-Level Processor for Morphological Analysis, version 2.1.0.," , vol. 1998, 1997.

[3] E. L. Antworth, *PC-KIMMO: A Two-Level Processor for Morphological Analysis*, vol. 16. Dallas, Texas: Summer Institute of Linguistics, 1990.

[4] H. Baayen, R. Piepenbrock, and H. van Rijn, "The CELEX Lexical Database," : Linguistic Data Consortium, University of Pennsylvania, 1993.

[5] D. Biber, "Co-occurrence Patterns Among Collocations: A Tool for Corpus-based Lexical Knowledge Acquisition.," *Computational Linguistics*, vol. 19, pp. 549-556, 1993.

[6] H. A. Black and C. A. Black, "A Conceptual Introduction to Morphological Parsing using AMPLE," in *LinguaLinks--Electronic Helps for Language Field Work*. Dallas, TX: Summer Institute of Linguistics, 1998, pp. 71.

[7] R. D. Borsley, *Modern Phrase Structure Grammar*. Oxford, UK: Blackwell, 1996.

[8] T. Briscoe, "Introduction," in *Inheritance, Defaults, and the Lexicon*, *Studies in Natural Language Processing*, T. Briscoe, V. de Paiva, and A. Copestake, Eds. Cambridge, England: Cambridge University Press, 1993, pp. 1-12.

[9] A. Buseman, D. J. Weber, H. A. Black, and S. R. McConnel, *Supplement to AMPLE: A Tool for Exploring Morphology*. Dallas, Texas: Summer Institute of Linguistics, 1992.

[10] L. Cahill and R. Evans, "An application of DATR: the TIC lexicon.," in *The DATR Papers.*, R. Evans and G. Gazdar, Eds. Brighton: The University of Sussex, 1990, pp. 31-39.

[11] L. Cahill and G. Gazdar, "Multilingual lexicons for related languages," presented at 2nd DTI Language Engineering Convention, London, England, 1995.

[12] L. J. Cahill, "Morphonology in the lexicon," presented at Proceedings of 5th European Conference on Computational Linguistics, 1993.

[13] L. J. Cahill, "Some Reflections on the Conversion of the TIC Lexicon," in *Inheritance, Defaults, and the Lexicon*, T. Briscoe, V. de Paiva, and A.

Copestake, Eds. Cambridge, England: Cambridge University Press, 1993, pp. 47-57.

[14] L. J. Cahill, "An Inheritance-based Lexicon for Message Understanding Systems," presented at Proceedings of Fifth European Conference on Computational Linguistics, 1994.

[15] L. J. Cahill, " Automatic Extension of a Hierarchical Multilingual Lexicon," presented at Second Multilinguality in the Lexicon Workshop, Brighton, 1998.

[16] L. J. Cahill, J. Carson-Berndsen, and G. Gazdar, "Phonology-based Lexical Knowledge Representation--a Tutorial," in *Lexicon Development for Speech and Natural Language Processing*, D. Gibbon, F. van Eynde, and I. Schuurman, Eds. Dordrecht: Kluwer, 1998.

[17] L. J. Cahill and G. Gazdar, " Allomorphy in PolyLex," in *Proceedings of the First Mediterranean Morphology Meeting*, A. Ralli and S. Scalise, Eds., 1998.

[18] L. J. Cahill and G. Gazdar, "The PolyLex Architecture: Multilingual Lexicons for Related Languages," *Traitement Automatique des Langues*, vol. 38, pp. 1-14, 1998.

[19] A. Cater, "Lexical Knowledge Required for Natural Language Processing," in *Machine Tractable Dictionaries: Design and Construction*, C.-M. Guo, Ed. Norwood, NJ: Ablex Publishing Corporation, 1995, pp. 31-53.

[20] N. Chomsky, "Remarks on Nominalization," in *Readings in Transformational Grammar*, R. Jacobs and P. Rosenbaum, Eds. Waltham, Mass.: Ginn, 1970.

[21] M. Colburn, "Erima Grammar Essentials," Summer Institute of Linguistics, Unpublished Manuscript. Ukarumpa, Papua New Guinea 1979.

[22] M. Colburn, "The Functions and Meanings of the Erima Deictic Articles," *Pacific Linguistics A-69, Papers in New Guinea Linguistics*, vol. 23, pp. 209-272, 1984.

[23] A. Copestate, A. Sanfilippo, T. Briscoe, and V. De Paiva, "The ACQUILEX LKB: an Introduction.," in *Inheritance, Defaults, and the Lexicon*, *Studies in Natural Language Processing*, T. Briscoe, V. de Paiva, and A. Copestake, Eds. Cambridge, England: Cambridge University Press, 1993, pp. 148-163.

[24] D. Crystal, *A Dictionary of Linguistics and Phonetics*, 4th. ed. Malden, Mass.: Blackwell Publishers, 1997.

[25] W. Daelemans, K. De Smedt, and G. Gazdar, "Inheritance in Natural Language Processing," *Computational Linguistics*, vol. 18, pp. 205-218, 1992.

[26] C. J. Date, *An Introduction to Database Systems*. Reading, MA: Addison - Wesley, 1986.

[27] W. Dolan, "A syllable-based parallel processing model for parsing Indonesian morphology.," in *Morphology as a Computational Problem*, K. Wallace, Ed. Los Angeles: Department of Linguistics, UCLA, 1988.

[28] M. Duda, "From DATR to PATR via DUTR," Institut fur Deutsche Sprache und Linguistik, Berlin 17, July 1994.

[29] R. Evans and G. Gazdar, "Inference in DATR," presented at Proceedings of the Fourth Conference of the European Chapter of the Association for Computational Linguistics, 1989.

[30] R. Evans and G. Gazdar, "The Semantics of DATR," presented at Proceedings of the Seventh Conference of the Society for the Study of Artificial Intelligence and Simulation Behavior, London, 1989.

[31] R. Evans and G. Gazdar, "The DATR Papers," University of Sussex, Brighton CSRP 139, 1990.

[32] R. Evans and G. Gazdar, "DATR: A Language for Lexical Knowledge Representation," *Computational Linguistics*, vol. 22, pp. 167-216, 1996.

[33] R. Evans and G. Gazdar, "The DATR Standard Library RFC, Version 2.00," 18 August 1998.

[34] R. Evans, G. Gazdar, and B. Keller, "A Bibliography of Papers on DATR," , vol. 1998, 1997.

[35] R. Evans, G. Gazdar, and B. Keller, "The DATR Web Pages at Leuven," , vol. 1998, 1997.

[36] R. Evans, G. Gazdar, and L. Moser, "Prioritized Multiple Inheritance in DATR," in *Inheritance, Defaults, and the Lexicon*, T. Briscoe, V. de Paiva, and A. Copestake, Eds. Cambridge, England: Cambridge University Press, 1993, pp. 38-46.

[37] R. Evans, G. Gazdar, and D. Weir, "Using Default Inheritance to Describe LTAG," presented at 3e Colloque International sur les grammaires d'Arbres Adjoints (TAG+3), Paris, 1994.

[38] R. Evans, G. Gazdar, and D. Weir, " Encoding Lexicalized Tree Adjoining Grammars with a Nonmonotonic Inheritance Hierarchy," in *Proceedings of the 33rd Annual Meeting of the Association for Computational Linguistics*, 1995, pp. 77-84.

[39] D. Flickinger, "Lexical Rules in the Hierarchical Lexicon," , vol. LexicalRulesHierarchicalLex.ps: Flickinger, Daniel, 1987.

[40] E. Gamma, J. Vlissides, R. Johnson, and R. Helm, *Design Patterns: Elements of Reusable Object Oriented Software*: Addison Wesley Longman, Inc., 1994.

[41] G. Gazdar, "Paradigm Merger in Natural Language Processing," in *Computing Tomorrow: Future Research Directions in Computer Science*, R. Milner and I. Wand, Eds. Cambridge: Cambridge University Press, 1996, pp. 88-109.

[42] G. Gazdar, E. Klein, G. Pullum, and I. Sag, *Generalized Phrase Structure Grammar*. Cambridge, MA: Harvard University Press, 1985.

[43] G. Gazdar and C. Mellish, *Natural Language Processing in Prolog--An Introduction to Computational Linguistics*. Wokingham, England: Addison-Wesley, 1989.

[44] D. Gibbon and G. Strokin, "ZDATR," , 2.0 ed. Bielefeld: Universität Bielefeld, Germany, 1998.

[45] L. Guthrie, J. Pustejovsky, V. Wilks, and B. M. Slator, "The Role of Lexicons in Natural Language Processing," *Communications of the ACM*, vol. 39, pp. 63-72, 1996.

[46] D. Hindle, "A Parser for Text Corpora," in *Computational Approaches to the Lexicon*, B. T. S. Atkins, B. Levin, and A. Zampolli, Eds. New York: Oxford University Press, 1994, pp. 103-151.

[47] D. Hull, "Stemming Algorithms - A Case Study for Detailed Evaluation," *Journal of the American Society for Information Science*, vol. 47, pp. 70-84, 1996.

[48] D. Hull and G. Grefenstette, "A Detailed Analysis of English Stemming Algorithms," Rank Xerox Research Centre, Meylan, France January 31 1996.

[49] R. Jackendoff, "The Boundaries of the Lexicon," in *Idioms: Structural and Psychological Perspectives*, M. Everaert, E. van der Linden, A. Schenk, and R. Schreuder, Eds. Hillsdale, New Jersey: Lawrence Erlbaum, 1995, pp. 29-44.

[50] R. T. Kasper and W. C. Rounds, "The Logic of Unification in Grammar.," *Linguistics and Philosophy*, vol. 13, pp. 35-58, 1990.

[51] F. Katamba, *Morphology*. New York: St. Martin's Press, 1993.

[52] B. Keller, "DATR theories and DATR models," in *Proceedings of the 33rd Annual Meeting of the Association for Computational Linguistics*, 1995, pp. 55-62.

[53] B. Keller, "An Evaluation Semantics for DATR Theories," presented at Proceedings of COLONG-96, Copenhagen, 1996.

[54] K. Koskenniemi, "A Two-Level Morphology: a General Computational Model for Word-Form Recognition and Production.," University of Helsinki Department of General Linguistics, Helsinki, Finland 11, 1983.

[55] D. D. Lewis and K. Sparck Jones, "Natural Language Processing for Information Retrieval," *Communications of the ACM*, vol. 39, pp. 92-101, 1996.

[56] T. G. Lewis, *Object-Oriented Application Frameworks*: Prentice Hall, 1995.

[57] J. McCarthy, "Formal Problems in Semitic Morphology and Phonology.," : MIT, 1979.

[58] S. R. McConnel, "AMPLE Reference Manual--A Morphological Parser for Linguistic Exploration," , vol. 1998: SIL, 1998.

[59] R. C. Moore, "Semantical considerations on nonmonotonic logic.," SRI International, Menlo Park, CA, Technical Note 284, 1983.

[60] R. C. Moore, "Possible-worlds semantics for autoepistemic logic.," Center for the Study of Language and Information, Stanford 1985.

[61] V. B. Y. Ooi, *Computer Corpus Lexicography*. Edinburgh, Scotland: Edinburgh University Press, 1998.

[62] C. Pollard and I. A. Sag, *Head-Driven Phrase Structure Grammar*. Chicago: The University of Chicago Press, 1994.

[63] G. Russell, A. Ballim, J. Carroll, and S. Warwick-Armstrong, "A Practical Approach to Multiple Default Inheritance for Unification-based Lexicons.," *Computational Linguistics*, vol. 18, pp. 311-338, 1992.

[64] S. M. Shieber, *An Introduction to Unification-Based Approaches to Grammar*. Sanford, CA: Center for the Study of Language and Information (CSLI), 1986.

[65] G. F. Simons, "Computing in Linguistics: A Tool for Exploring Morphology," *Notes on Linguistics*, vol. 44, pp. 51-59, 1989.

[66] A. Spencer, *Morphology Theory*. Cambridge, England: Basil Blackwell, 1991.

[67] A. Spencer and A. M. Zwicky, "Introduction," in *The Handbook of Morphology*, A. Spencer and A. M. Zwicky, Eds. Oxford: Blackwell Publishers, Ltd., 1998, pp. 1-10.

[68] R. Sproat, *Morphology and Computation*. Cambridge, MA: The MIT Press, 1992.

[69] K. Vijay-Shanker and Y. Schabes, "Structure sharing in lexicalized tree-adjoining grammars.," *COLING-92*, vol. I, pp. 205-211, 1992.

[70] D. J. Weber, H. A. Black, and S. R. McConnel, *AMPLE: A Tool for Exploring Morphology*, vol. 12. Dallas, Texas: Summer Institute of Linguistics, 1988.