Adaptive Parsing:
Self-extending Natural Language Interfaces

Jill Fain Lehman
28 August 1989
CMU-CS-89-191

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Submitted to Carnegie Mellon University in partial fulfillment of the requirements for the degree of Doctor of Philosophy in Computer Science.

This research was partially supported by the Defense Advanced Research Projects Agency (DOD), ARPA Order No. 4976, and monitored by the Air Force Avionics Laboratory under Contract F33615-87-C-1499. Jill Fain Lehman was partially supported by a Boeing Company fellowship.

The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency, the Boeing Company, or the U. S. government.

To my family

''I don't know what you mean by 'glory,' '' Alice said.

Humpty Dumpty smiled contemptuously. ''Of course you don't—till I tell you. I meant 'there's a nice knock-down argument for you!' ''

''But 'glory' doesn't mean 'a nice knock-down argument,' '' Alice objected.

''When I *use a word,'' Humpty Dumpty said, in rather a scornful tone, ''it means just what I choose it to mean—neither more nor less.''*

''The question is,'' said Alice, ''whether you can *make words mean so many different things.''*

''The question is,'' said Humpty Dumpty, ''which is to be master—that's all.''

— *Lewis Carroll*
Alice's Adventures in Wonderland

Acknowledgements

Many people have contributed to the creation of this document. As always with such efforts, their help has come in many forms, including contributions of ideas, guidance, inspiration, encouragement, environment, software, time, energy, friendship, love, and food. And, as always, due to limitations of space (but not appreciation) what follows is an incomplete list, both in those it mentions, and in the ways in which I counted on their support.

Jaime Carbonell acted as chairman of the dissertation committee, providing guidance and encouragement along the path through graduate school, research direction, and a wealth of advice and insight on the content of the final document.

Allen Newell's part in this research was to be...Allen Newell (a role in which he excels). Without fail, he provided a new and different point of view during our discussions. His global perspective and profound breadth of knowledge have affected my view of the world. I am most grateful, however, for his friendship throughout my graduate career.

Tom Mitchell lent perspective on CHAMP's role in the larger world of machine learning, and provided many useful comments on the written dissertation.

As the ''outside'' committee member, Wendy Lehnert acted from a distance to provide an alternative point of view on the work presented here. In a larger sense, however, she is the most ''inside'' member of my committee, having encouraged and supported me since my first endeavors in natural language research as an undergraduate.

Through long discussions or heated debates Angela Kennedy Hickman, Mark Perlin, Alex Hauptmann, and Wayne Ward all helped in fleshing out ideas and exorcising various research demons. They are characteristic of the generous environment at Carnegie Mellon University and its School of Computer Science.

Completion of the experimental work in this dissertation required help from a variety of sources. David Nichols and Kelly F. Hickel created software that made the ''hidden-operator'' and ''on-line'' experiments possible. Michael and Julie Shamos contributed some of the facilities that resulted in a wider subject pool than otherwise would have been available. Special thanks must go to Marion Kee for her invaluable assistance in

running the experiments (and keeping users calm). Ben McClaren also assisted. Finally, without the users' willing participation I would not have been able to present the arguments central to the dissertation.

Like Allen Newell, Sharon Burks helped by being herself: a source of guidance, encouragement, and assistance with everything from the trivial to the critical.

This document was improved by the painstaking reviews given it by Angela Kennedy Hickman and Philip Lehman. Angela also helped enormously simply by knowing exactly how to be a friend. The degree to which Philip contributed to this dissertation is immeasurable, as are his contributions to my life.

My mother and father instilled in me an appreciation for the beauty and intricacy of language. Together with my brother Kenneth and the rest of my family, my parents have been a source of love and support, and the providers of the underlying intellectual curiosity that was a major force in getting me through graduate school.

Abstract

A long-term goal of natural language processing is to provide users with an environment in which interaction with computers is as easy as interaction with human beings. In particular, the philosophy behind the design of natural language interfaces is to permit the user the full power and ease of her usual forms of expression to accomplish a task. At odds with this philosophy is the computational barrier created by natural language's inherent ambiguity; as the size of the permissible grammar increases, system performance decreases. Thus, current natural language interfaces tend to fail along two dimensions: first, they fail philosophically by limiting the user to an arbitrary subset of her natural forms of expression. Second, they fail theoretically by being non-extendible in any practical sense.

The research presented advances a new, alternative approach to interface design based on a model of language acquisition through automatic adaptation. The model depends on a key observation about frequent user behavior: although individual users differ significantly in their preferred linguistic expression, each frequent user is quite consistent in her usage over time. This *self-bounded* aspect of user behavior is demonstrated empirically in an initial set of experiments using a simulated system and a later set of experiments using an implementation of the model.

The regularity in each frequent user's linguistic behavior is exploited in the design of a general mechanism to learn idiosyncratic grammars. The mechanism explains perceived errors in the input utterance as deviations with respect to the system's current grammar. The explanation is then transformed into new grammatical components that are capable of recognizing the general structure of the deviation in future interactions. Thus, in contrast to most existing interfaces, the *adaptive interface* is self-extending and allows the user to employ her own natural language for accomplishing tasks.

The usefulness and robustness of adaptation is demonstrated by the implementation of the model in a working interface. The interface has been evaluated both on the data from the original simulations and in on-line interactions with real users. The results of the evaluation clearly show adaptation's effectiveness; the implementation was able to capture both the regularity and the idiosyncracy in the natural grammars of eight users.

Table of Contents

1. Introduction 1
1.1. The Uses of Regularity in Language Understanding 2
1.2. The Frequent User Problem 4
1.3. Contributions of the Dissertation 6
1.4. Reader's Guide 7
2. System Behavior in an Adaptive Environment 9
2.1. Foundations: Least-deviant-first Parsing and MULTIPAR 9
2.2. The Model 11
2.3. Assumptions and Hypotheses 16
2.4. Adaptation vs. Customizable and Instructable Interfaces 18
2.5. Prior Research in Adaptation 19
3. User Behavior in an Adaptive Environment 25
3.1. The Behavioral Hypotheses 25
3.2. The Experimental Condition 27
3.3. Control Conditions 30
3.4. User Profiles 31
3.5. Results and Discussion 31
4. System Architecture and Knowledge Representation for an Adaptive Parser 41
4.1. Representing Knowledge in CHAMP 43
4.2. Learning-Related Knowledge: The Lexicon and Formclass Hierarchy 45
4.2.1. The Formclass 47
4.2.2. The Form and its Steps 50
4.2.3. The Wordclass and the Lexical Definition 55
4.3. Application-Related Knowledge: The Concept Hierarchy 56
5. Understanding Non-Deviant Utterances 63
5.1. Segmentation 65
5.1.1. Creating Tokens 65
5.1.2. Partitioning Tokens into Segments 65
5.1.3. Resolving Unknown Segments as New Instances of Extendable Classes 68
5.1.4. Applying Segmentation-time Constraints 72
5.1.5. Seeding the Agenda 74

5.2. The Coalesce/Expand Cycle 77
5.3. A Detailed Example: The Complete Coalesce/Expand Cycle for "Cancel the 3 p.m. speech research meeting on June 16." 84
6. Detecting and Recovering from Deviation 91
6.1. Parsing as Least-deviant-first Search 93
6.2. The Cache 100
6.3. Error Detection 103
6.4. A Detailed Example: Error Detection during the Parse of "Schedule a meeting at 3 pm June 7" 106
6.5. Error Recovery 110
7. The Resolution Phase: Choosing Among Explanations 125
7.1. Converting Meaning to Action 130
7.2. Using Default Knowledge and Inference 133
7.3. Interacting with the Databases 137
7.4. Confirmation by Effect 142
8. Adaptation and Generalization 145
8.1. Adapting to Substitution Deviations 148
8.2. Adapting to Insertion Deviations 153
8.3. Adapting to Deletion Deviations 154
8.4. Adapting to Transposition Deviations 158
8.5. A Summary of Adaptation in CHAMP 159
8.6. Controlling Growth in the Grammar through Competition 165
8.7. The Effects of Adaptation on Performance 171
9. Evaluating the Interface 177
9.1. CHAMP's Performance on Data from the Hidden-operator Experiments 178
9.1.1. Differences between CHAMP and the Simulated System 179
9.1.2. Comparison of Performance: Simulation versus Implementation 184
9.1.3. Analysis of Differences in Acceptance/Rejection 187
9.1.4. Analysis of the Utility of Adaptation and Generalization 189
9.1.5. Adaptive versus Monolithic Performance 195
9.2. CHAMP's Performance in Real User Interactions 198
9.2.1. Synopsis of User and System Performance 199
9.2.2. Analysis of Rejections 201
9.2.3. Analysis of the Utility of Adaptation and Generalization 202
9.2.4. Adaptive versus Monolithic Performance 205
9.3. Summary 206
10. Critical Issues 209
10.1. The Effect of Kernel Design on Learning 209
10.2. The Effect of Kernel Design on Search 212
10.3. The Effect of Search Constraints on System Predictability 213
10.4. The Lexical Extension Problem 214
10.5. Summary 216

11. Conclusions and Future Directions 219
11.1. Main Results and Contributions 219
11.2. Future Directions 221
Appendix A. CHAMP's Kernel Grammar 225
A.1. The Kernel Lexicon 225
A.2. The Formclass Hierarchy 229
A.3. The Concept Hierarchy 243
Appendix B. Test Set Sentences 249
B.1. Utterances from User 1 250
B.2. Utterances from User 2 254
B.3. Utterances from User 3 259
B.4. Utterances from User 4 263
B.5. Utterances from User 5 268
B.6. Utterances from User 7 269
B.7. Utterances from User 9 272
B.8. Utterances from User 10 278
Appendix C. Performance Measurements for User Data 283
C.1. Users in Hidden-operator Experiments 283
C.1.1. Data for User 1 285
C.1.2. Data for User 2 286
C.1.3. Data for User 3 287
C.1.4. Data for User 4 288
C.1.5. Data for User 5 289
C.1.6. Data for User 7 290
C.2. Users in On-line Experiments 291
C.2.1. Data for User 9 292
C.2.2. Data for User 10 294

Chapter 1
Introduction

A long-term goal of natural language processing is to provide users with an environment in which interaction with computers is as easy as interaction with human beings. In particular, the philosophy behind the design of natural language interfaces is to permit the user the full power and ease of her usual forms of expression to accomplish a task. At odds with this philosophy is the computational barrier created by natural language's inherent ambiguity; as the size of the permissible grammar increases, system performance decreases. Thus, current natural language interfaces tend to fail along two dimensions: first, they fail philosophically by limiting the user to an arbitrary subset of her natural forms of expression. Second, they fail theoretically by being non-extendible in any practical sense.

The research presented in this dissertation advances a new, alternative approach to interface design based on a model of language acquisition through automatic adaptation. The model depends on a key observation about frequent user behavior: although individual users differ significantly in their preferred linguistic expression, each frequent user is quite consistent in her usage over time. This *self-bounded* aspect of user behavior is demonstrated empirically in an initial set of experiments using a simulated system and a later set of experiments using an implementation of the model.

The regularity in each frequent user's linguistic behavior is exploited in the design of a general mechanism to learn idiosyncratic grammars. The mechanism explains perceived errors in the input utterance as deviations with respect to the system's current grammar. The explanation is then transformed into new grammatical components that are capable of recognizing the general structure of the deviation in future interactions. Thus, in contrast to most existing interfaces, the *adaptive interface* is self-extending and allows the user to employ her own natural language for accomplishing tasks.

The usefulness and robustness of adaptation is demonstrated by the implementation of the model in a working interface. The interface has been evaluated both on the data from the original simulations and in on-line interactions with real users. The results of the evaluation clearly show adaptation's effectiveness; the implementation was able to capture both the regularity and the idiosyncracy in the natural grammars of eight users.

1.1. The Uses of Regularity in Language Understanding

It is common to view natural language understanding as a search problem in a space of partially instantiated meaning structures. Search is necessary because of the local ambiguity of language; in general, the greater the linguistic coverage of the system, the more ambiguity there will be in the grammar, and the less computationally effective will be a "naive" strategy such as exhaustive search. This is true regardless of whether the target meaning structures represent a semantic or syntactic analysis of the utterance, since sufficiently complex grammars tend to engender both kinds of ambiguity. Research in natural language processing has concentrated, therefore, on ways to exploit various combinations of syntactic and semantic knowledge to constrain the size of the space that must be examined.

Regularity is the feature of syntax and semantics that gives them the power to limit search. Through common use we have defined both what particular words can mean and how they can combine. Because these definitions holds across uses, we can factor them out and treat them as a source of predictions.[1] Predictions appear, implicitly or explicitly, as the system's expectations of what it will encounter in the future. Expectations may be strong, as when a partial interpretation dictates an event, or weak, as when a partial interpretation suggests an event whose later occurrence is taken as confirming evidence for the interpretation. Strong or weak, expectations serve a uniform purpose; they act as an heuristic method for focusing search.

There is another source of regularity, however: one which can be found in the consistent use of a particular subset of syntax and semantics over time. Such a set of restrictions on form and reference is usually defined as *style*. Given this definition, we can say that every fixed grammar we devise necessarily recognizes language used in a particular style (although not necessarily a style that conforms to the usage of any individual). Thus, in a trivial sense, every system already uses style to constrain search by virtue of understanding only a subset of natural language. When the syntax and semantics described by the underlying grammar are fixed, the style embodied in a parser is implicit. The question is whether there are gains to be had from explicating style. The answer requires that we examine how syntax and semantics, as static sources of regularity, fail in their predictive roles.

Left implicit, the style embodied by a parser is a constant. The system's usefulness

---

[1]Natural language systems tend to get into trouble exactly when an assumed regularity is not manifested by the user; for example, an assumption of adherence to strict grammaticality in interactive dialogue is more of a hindrance than a help. Nevertheless, the principle holds: we can extend the system's grammar to include those extragrammatical and ungrammatical constructs it is likely to encounter. By "compiling in" a source of regularity we get additional power to constrain search [14, 21].

under these conditions presupposes some attempt at a complete analysis of the linguistic phenomena associated with a given task. Although it is a goal of natural language systems to allow discussion of the domain in a variety of ways, when the number of permissible forms of reference grows, ambiguity generally grows as well. The lack of practical extendibility seen in most natural language systems results from this unfortunate fact. The nature of language is such that a complete analysis, even if possible, is hardly useful; if we try to anticipate all the linguistic forms the system might encounter and all the word meanings and inferences that might be intended, we will formalize enough of English to make the effects of ambiguity computationally intolerable.

The historical solution to this dilemma is to choose small enough domains and specific enough tasks such that a style can be fixed that has the elusive property of being both computationally tractable and "user-friendly." Yet a fixed style results in two kinds of mismatch between the language expected (the system's language) and the language encountered (the user's language): utterances the system can understand but that the user will never employ, and utterances that the user employs but that the system cannot understand. Because of these mismatches, tractability extracts a price: search is wasted both in allowing for utterances the system will never encounter and in trying to resolve utterances the system was not designed to interpret. In other words, as long as the predictions embodied in the grammar are static, the power of the system—its ability to constrain search—is fixed as well.

The problems intrinsic to restricting style seem to stem from the assumption that a single parser must be able to understand every user. If we discard this assumption, we can take advantage of the possibility that individuals do not always use all of language. In repetitive tasks—going to the store, eating at a restaurant—we are accustomed to finding repetitive behavior—scripts and plans [51]. It is a premise of this thesis that stylization need not apply only to action, but may apply to language as well, especially where the principal method for accomplishing a task is linguistic. We argue further that under appropriate circumstances people are both *idiosyncratic* and *self-bounded* in their language use; they naturally restrict their forms of expression to the preferred subset of language that they used to accomplish the task in the past.

If people are self-bounded in their language use, then a system that can learn those bounds incorporates regularity over time as an additional source of predictions. Such a system need not define all meaningful forms of expression *a priori*; it need only learn the particular forms preferred by a given user over time. We call the gradual augmentation of a kernel grammar to include the user's preferred forms of expression *adaptive parsing*. In essence, the purpose of adaptive parsing is to learn a user's idiosyncratic style.

The basic challenges in constructing an adaptive parser are similar to those involved in the design of any learning system. How do we decide when to learn? How do we decide

what to learn? What learning method will bring the new knowledge to bear during search only when that knowledge is appropriate? The model of adaptation presented in the next chapter answers these questions in the context of language acquisition in a goal-oriented, interactive environment. The model demonstrates how an expectation-driven parser can learn by noticing violated expectations, explaining them as deviations with respect to a current grammar, and then deriving new grammatical components based on the explanations. The system is able to learn an idiosyncratic style because it judges the grammaticality of an utterance in relation to a history of interactions with a particular user, rather than by any absolute measure.

An adaptive parser need not suffer from the problems described above for non-adaptive parsers. The process of adapting a grammar to reflect the language of an individual user directly addresses the issue of extendability; from some initial body of knowledge, the system increases its linguistic coverage in response to a particular set of experiences. If our goal were to understand every user with a single system, the limit on extendability would be the union of all idiosyncratic grammars and the ambiguity inherent in such a language description would place the goal computationally out of reach. In adaptive parsing, however, we are concerned with only one idiosyncratic grammar at a time. The crucial observation is that *the user's self-bounded behavior places a natural limit on the degree of extendability required of the system and, therefore, the amount of ambiguity that must be tolerated during search.*

Adaptation to a self-bounded grammar addresses the mismatch problem as well. As long as the user's preferred forms of expression are represented in an initial, minimally ambiguous grammar, or their representation is derivable from it through adaptation, mismatch is controlled because the system reflects only those inefficiences actually embodied in the ambiguity of the particular linguistic style. Tractability cannot be guaranteed absolutely—it is a function of the amount of stable ambiguity in the individual's idiosyncratic grammar. Still, by relying on the self-bounded behavior of the user, we trade the current impossibility of understanding every individual effectively for the likelihood of understanding any individual well.

1.2. The Frequent User Problem

Although much can be gained by considering style as a source of regularity, there are clearly language understanding tasks that remain intractable despite this additional power. Ernest Hemmingway and Herman Melville certainly differ in style, and although distinct parsers could take advantage of that fact, it is the need to encode and reason with a large amount of world knowledge that causes most of the computational difficulties in understanding narrative texts. Thus, in order to demonstrate the power of style, we have focused on the class of constrained, task-oriented, natural language understanding situations that involve interaction with a user of a software system.

Current research in designing natural language interfaces seems to derive much of its justification from the need to support an increasing number of naive users. The interests of these users lie solely in the semantics of the application domain with which they interact. They may have only the vaguest model of how software systems are organized and it may not be cost-effective to improve that model (see, for example, [34]). For these users, command languages often seem arbitrary and non-intuitive; depending upon the user's level of experience and the complexity of the interface, the technology intended to increase productivity often becomes the new bottleneck.

The existence of a system-supplied command language requires that the user learn both the system's task model and a communication, or command language, model. The burden is on the user to express her intent completely and unambiguously within the forms available. In theory, by allowing the user to interact via an English language front-end, the need for a separate communication model disappears, moving the burden of clarifying intent into the natural language interpreter. However, even natural language interfaces fall short of immediate utility by naive users; no existing interface permits all of the complexities of full natural language, including the full range of potentially relevant vocabulary, non-standard turns of phrase, and extra-grammatical utterances [9]. Thus, a user must learn the extent and limitations of an interface, mostly by trial and error, reintroducing the need to master communication as well as the underlying task.

Shneiderman [55] has divided potential users of natural language interfaces along two axes: degree of semantic (application) knowledge and degree of syntactic (formal language) knowledge. He has argued that as a method of interaction, natural language interfaces are appropriate only for "infrequent novice users" who have "semantic knowledge of a problem domain yet are not knowledgeable in the query language syntax." The "frequent professional user," he states, "with rich syntactic and semantic knowledge would probably prefer the precise concise query facility." It is clear from this characterization that Shneiderman equates formal language sophistication with frequency of use. His view ignores the fact that frequency of use is dictated by the task and need not by correlated with formal language experience; a task requiring frequent interactions may be approached by a user who is syntactically unsophisticated.

We agree with Shneiderman that the infrequent novice user is best served by a flexible natural language interface. With sporadic interactions, her model of how to use the system remains relatively constant and uninformed. We differ from Shneiderman's view, however, in distinguishing between the sophisticated frequent user and the naive frequent user. For the former, the time and energy required to learn a complex, system-supplied command language may not be prohibitive, while for the latter, it often is.

If a natural language interface best serves the naive user, an adaptive natural language interface best serves the naive *frequent* user. Through regular interaction the frequent

user's methods become stylized and her model of how to accomplish the task grows in particular ways. This stylization is exactly what an adaptive parser comes to reflect, at the very least increasing the efficiency of the frequent user's interactions over those of the infrequent user. As the naive frequent user becomes increasingly expert in her interactions, however, she may also want the convenience and speed of the command language available to the sophisticated frequent user [38] [24] [48]. The same process of adaptation that allows non-standard turns of phrase and ungrammatical constructions also permits the user to develop a concise and idiosyncratic command language as part of its natural function. Although we cannot guarantee that an idiosyncratic command language will be as efficient as a system-supplied one, there is also no guarantee that the user's experience has rendered an arbitrary system designer's choices more intuitive. In addition, a user who must interact frequently with a number of different application programs will rarely find any common philosophy in their interface designs. Even after reaching "expert" status she will be forced to keep a multitude of conventions in mind (or manuals in her pocket) if she is to take advantage of the conciseness and execution speed of simpler command parsers. In theory, a single adaptive interface to a variety of application programs solves this problem as well.

In the scenario of the naive frequent user, adaptation serves to develop the parser in the same directions the user is developing. Learning an idiosyncratic command language can be recast as learning a particular interface user's style. Under appropriate conditions the benefits of such a system include better performance over time and a smooth transition for the user from naive to expert status.

1.3. Contributions of the Dissertation

The thesis of this dissertation is that adaptive parsing is a desirable and expedient aspect of natural language interface design when users act in a linguistically idiosyncratic and self-bounded manner. Adaptation reflects the idea that regularity over time introduces compensatory search constraint for individual linguistic freedom. In other words, a grammar extended in response to a user's idiosyncratic language is naturally bounded in its growth by the user's reliance on forms that have been effective in the past.

Support for this thesis is established by three main results:

- *Idiosyncratic, self-bounded linguistic behavior under conditions of frequent use*. We demonstrate that with frequent interactions a user limits herself to a restricted subset of forms that is significantly smaller than the full subset of natural language appropriate to the task, and quite different from the restricted subsets chosen by others.
- *Adaptive parsing, in theory*. We present a model of adaptive language acquisition capable of learning different idiosyncratic grammars with a single, general mechanism.

- *Adaptive parsing, in practice.* We examine in detail a computational realization of the model as a system able to perform effectively in the presence of spontaneous user input.

1.4. Reader's Guide

This section gives a brief synopsis of each of the remaining chapters in the thesis for the convenience of readers with particular interests. A basic understanding of adaptive parsing can be acquired from Chapters 1 through 3 and Chapters 9 through 11. The remaining material (Chapters 4 through 8) describes an implemented adaptive interface at a level of technical detail that is likely to be beyond the interests of the casual reader. A more cursory understanding of the implementation can be acquired, however, by reading the introductory sections of each of those chapters as well as Sections 4.1 and 8.7.

Chapter 2 describes our model of adaptive parsing and introduces such notions as deviation, recovery, and least-deviant-first search. The chapter also states clearly the assumptions underlying the model and poses a number of testable hypotheses to justify the assumptions. In the final sections of Chapter 2, we discuss alternative approaches to adaptation and prior research in language acquisition.

Chapter 3 discusses a set of hidden-operator experiments in which users interacted with a simulated adaptive interface based on the model described in Chapter 2. The results of the experiments support the behavioral assumptions of self-bounded, idiosyncratic language use under conditions of frequent interaction. In addition, the experiments provided a large set of spontaneously generated utterances, a small subset of which helped guide the subsequent design of a working interface. All data collected from the experiments was used in the evaluation of that interface, as described in Chapter 9.

Chapter 4 begins the discussion of CHAMP, an implementation of the model as a working adaptive interface. This chapter presents the overall design of the system and examines its knowledge representations in detail.

Chapter 5 introduces the bottom-up parser. The chapter is designed to familiarize the reader with the basic parsing mechanism before introducing the complexities associated with error detection, error recovery, and adaptation. The final section of the chapter presents an extremely detailed example of parsing grammatical input.

Chapter 6 extends the basic parsing algorithms to include error detection and error recovery. The extensions turn the bottom-up parser described in the previous chapter into a least-deviant-first bottom-up parser.

Chapter 7 examines the portion of the interface responsible for resolving multiple

interpretations of an utterance. In particular, we examine the role of the user and of the domain databases in providing additional constraint for the understanding process.

Chapter 8 describes CHAMP's method of adaptation and generalization. We demonstrate how the explanations produced by error recovery and verified by resolution are transformed into the new grammatical components that recognize formerly deviant input directly in future interactions. Adaptation turns the least-deviant-first bottom-up parser described in Chapters 6 and 7 into the full adaptive parser found in CHAMP.

Chapter 9 evaluates CHAMP in two ways. First, we examine how the performance of the implementation compares to the performance displayed by the experimental simulation of the model for the same data. Second, we validate our preliminary results in on-line experiments with two new users.

Chapter 10 discusses general issues in adaptive interface design, outlining some problems and tradeoffs that seem to be inherent in learning language from real user interactions.

Chapter 11 concludes with a review of the main results of the research and directions for future work.

Chapter 2
System Behavior in an Adaptive Environment

The goal of an adaptive parser is to acquire an idiosyncratic grammar with minimal ambiguity and maximal coverage through repeated experience with a user. To attain this goal a system must have both a method for expanding the search space in the presence of an unparsable input and a method for augmenting the grammar to recognize the unfamiliar structure directly in future interactions. Using such methods the system can extend an initial grammar to handle new forms, or variants of existing forms, whether they be truly ungrammatical structures, or grammatically valid extensions to an initial grammar whose coverage was less than complete.

In the next section we introduce some of the notions fundamental to our model of adaptive parsing—deviation, recovery, and least-deviant-first search—in the context in which they first appeared. Section 2.2 then refines and generalizes these ideas and presents our model of adaptation. The assumptions that underlie the model are examined in Section 2.3, while the last two sections of the chapter compare the model to other adaptive approaches and natural language interface designs.

2.1. Foundations: Least-deviant-first Parsing and MULTIPAR

Our approach to adaptive parsing is based in part on ideas first explored in the flexible parser MULTIPAR [14, 42]. MULTIPAR was designed to overcome certain specific types of extragrammatical behavior frequently displayed by interface users: simple spelling errors, some word-order or phrase-order inversions, missing case markers, missing determiners, and semantic inconsistencies.

Like most natural language parsers, MULTIPAR's performance can be described as search through the virtual space of possible parses for a given input. In most parsers, search is constrained by the violation of *expectations* inherent in the grammar; for example, the expectation that a preposition introduces a postnominal case, or the expectation that a particular lexeme acts as verb. When an expectation is unmet, the search path is abandoned. In contrast, MULTIPAR could not necessarily limit search by rejecting a partial parse when an expectation was violated. Since some ungrammatical

constructions were permitted in MULTIPAR, the system could not necessarily limit search by rejecting a partial parse when an expectation was violated. If the linguistic *deviation* belonged to one of the anticipated types of extragrammatical behavior, the system invoked an associated *recovery strategy* to compensate—for example, by inserting a case marker where one was missing. Compensating for the deviation allowed the parse to continue, at the cost of some loss of confidence in the resulting meaning structure for the utterance. In order to produce the best meaning and to constrain search if too many expectations were violated, each type of violation was assigned a degree of deviance. Deviant partial parses were inserted into a queue and explored in a *least-deviant-first* manner. This means simply that grammatical parses were attempted first, parses containing minor deviations such as orthographic anomalies next, and paths with more serious grammatical deviations were examined only if no meaning could be constructed using less serious deviations.[2]

In MULTIPAR the loci at which expectations could go unmet were predetermined, fixed points in the grammar. These recovery points were empirically selected based on the most frequently encountered user deviations in a number of informal studies. In some sense, when we built MULTIPAR we simply extended our notion of what we would consider part of the grammar to include those classes of extragrammatical constructions most often produced by interface users. With increased flexibility, MULTIPAR was an improvement over stricter interfaces, especially for the sporadic user. In designing an interface for the frequent user, however, we want to take advantage of the stylization that occurs with frequent interaction to provide a system capable of capturing an idiosyncratic grammar developing over time. Unfortunately, the static model of deviance and recovery embedded in MULTIPAR is inadequate to this task. By predetermining the recovery points, MULTIPAR can anticipate only a small subset of the possible deviations in an utterance, denying the user true idiosyncracy of expression. In addition, because MULTIPAR cannot change its definition of what is grammatical, it also cannot improve its performance in response to regularity in the user's language; once a deviation, always a deviation.

Like its predecessors, MULTIPAR's limitations stem from the design goal of understanding everyone. Nevertheless, the system serves as an introduction to the fundamental point of view of parsing as deviation detection and recovery in a least-deviant-first search. In the next section we will see how these ideas can be extended to accommodate the frequent user.

---

[2]This description of MULTIPAR does not usefully distinguish the system from earlier work by Kwasny and Sondheimer [35], or Weischedel and Black [62] which explored the ideas of deviance and relaxation in Augmented Transition Network grammars (ATNs) [64]. Although MULITPAR can be said to share the same abstract point of view, it differed significantly from the other systems both in the generality of its approach and the scope of extragrammatical phenomena it tolerated.

2.2. The Model

Recall that the goal of an adaptive parser is to acquire an idiosyncratic grammar with minimal ambiguity and maximal coverage through repeated experience with a user. In essence, the system must remember the recovery actions performed in the presence of an unfamiliar utterance, and then augment the grammar with a representation that can parse the same structure directly. Thus, in order to take advantage of regularity in language use over time, we must generalize the key concepts of deviation and recovery to be independent of any particular grammar. We begin by introducing the following definitions:

- **Kernel grammar**: the lexicon and syntactic forms that are present in the system prior to interaction with any user.[3] The kernel grammar organizes information about typical forms of reference. For example, the pattern "<month> <day> , <year>" is a typical form of reference for a date.
- **Domain concepts**: the actions and objects that are meaningful referents in the task domain. A domain concept organizes the static information (such as semantic constraints and default values) that is associated with an action or object. Thus, the **day** concept might limit the value bound to <day> in "<month> <day> , <year>" to a number between one and thirty-one, while the **date** concept might enforce a tighter upper-bound depending upon the value of <month>.
- **System form/kernel form**: any form in the kernel grammar, lexical or syntactic.
- **Deviation**: the violation of an expectation as embodied in a syntactic form. Exactly four types of deviation, other than misspelling, are possible:
  - the lack of expected text (deletion error)
  - the existence of unexpected text (insertion error)
  - the presence of text instead of the text expected (substitution error)
  - the presence of expected text in an unexpected location (transposition error)

  Note that misspellings correspond to exactly the same four categories but the constituents are letters in words rather than phrases in sentences [13].
- **Recovery action**: one of insertion, deletion, substitution, transposition, or spelling correction. Each action compensates for one type of deviation. The recovery action for a deletion error is insertion; for an insertion error, it is deletion; transposition and substitution are their own inverses.
- **Deviation-level**: a value that indicates the number of deviations permitted at a given point in the search for a meaning representation for the utterance. Search at Deviation-level 0, for example, will succeed only for utterances directly parsable by forms already in the grammar. Adopting the least-deviant-first method, the

---

[3] "Syntactic" should be interpreted to mean only "a related set of expectations." In other words, the model may be applied to semantic grammars as well as traditional syntactic ones. A semantic grammar is one that uses semantic categories such as <meal> in place of, or in addition to, purely syntactic categories such as <noun phrase>.

deviation-level is increased whenever all paths at the current level have failed. Thus, if search for a "grammatical" interpretation is unsuccessful, Deviation-level 1 is considered, parsing utterances that require only one recovery action. At Deviation-level 2, two recovery actions are permitted along a search path, and so forth.

- **User form/derived form**: a new lexical or generalized syntactic form created in response to the successful application of one or more recovery actions in the presence of deviant input. These forms are added to the specific user's "adapted kernel" (the kernel grammar remains unchanged).
- **Adapted kernel**: a grammar that includes both system and user forms. An adapted kernel is specific to a single user.
- **Current grammar**: the lexical and syntactic forms available at a given point in time. Deviation is always defined relative to the current grammar which may be either the kernel grammar or an adapted kernel.

The set of four deviations and their corresponding recovery actions were chosen for two reasons. First, for completeness: every utterance can be mapped into a grammatical form by zero or more applications of insertion, deletion, transposition, and substitution. The second reason for the choice is the set's lack of commitment to any underlying linguistic theory stronger than a basic ordering assumption. Instead, the set gives the system a kind of *linguistic weak method* for understanding that seems appropriate for an initial exploration of language acquisition through adaptation.

Sample sentences at various levels of deviation and the recovery actions that correct them are shown in Figure 2-1. The system form in the example is given in standard BNF notation: terminals appear as tokens, non-terminals are enclosed in angle brackets, optional constituents are enclosed in square brackets, alternatives are separated by a bar and an asterisk is read as the Kleene star: "zero or more instances of." Thus, the system form in Figure 2-1 would be paraphrased as, "The word *add* or *schedule* optionally followed by *a* or *an*, followed by one of *meeting*, *appointment*, or the extension of the non-terminal *meal*, followed by zero or more marked cases." Beneath each deviant sentence we introduce the notational conventions used for indicating the four kinds of error: deleted segments are reintroduced in square brackets, inserted segments are surrounded by "><," transposed segments are underlined, and substitutions are italicized.

Using the definitions introduced above we describe the essence of our model for adaptive parsing in the following manner. For each utterance, we perform a least-deviant-first parse. If the utterance can be understood using only the domain concepts and the current grammar then a meaning structure is produced at Deviation-level 0 and no adaptation is necessary. If the utterance is deviant with respect to the current grammar then one or more recovery actions are required and a derived form is constructed to capture the effects of recovery. In short, the adaptive understanding process is one in

System form: add | schedule [a | an] meeting | appointment | <meal> <marked-case>*

0 deviations: "Schedule lunch with Bob at 12"

1 deviation: "There is a dinner at AISys at 7:00 p.m."
notation: "*There is* a dinner at AISys at 7:00 p.m."
Recovery:
1. substitute "add" for "There is"

2 deviations: "Meeting with Philip on Thursday, June 12 10 to 11 a.m."
notation: "[Add] Meeting with Philip on Thursday, June 12 [from] 10 to 11 a.m."
Recovery:
1. insert "add" before "Meeting"
2. insert "from" before "10"

3 deviations: "I have to be at AISys for a meeting at 7:00 p.m."
notation: "*I have to be* at AISys >for< a meeting at 7:00 p.m."
Recovery:
1. substitute "add" for "I have to be"
2. transpose "at AISys" to postnominal position
3. delete "for"

4 deviations: "There is an AISis appt. at 7 p.m. June 11"
notation: "*There is* an AISis *appt.* at 7 p.m. [on] June 11"
Recovery:
1. substitute "add" for "There is"
2. spelling correct "AISis"
3. substitute "appointment" for "appt."
4. insert "on" before "June"

Figure 2-1: Sample deviant sentences and recovery actions with errors indicated by notational convention.

which we look for the simplest combination of deviations that help explain a particular input utterance and store some generalization of that explanation as a new user form.[4] Figure 2-2 displays the model schematically.

[4]A word of caution is appropriate here for readers familiar with the machine learning literature. The term "explanation" is used throughout this work in an intuitive sense rather than in the technical sense associated with Explanation-Based Generalization [43]. In terms of our model, "explaining" an utterance simply means using the grammar and recovery actions to associate a meaning to the input.

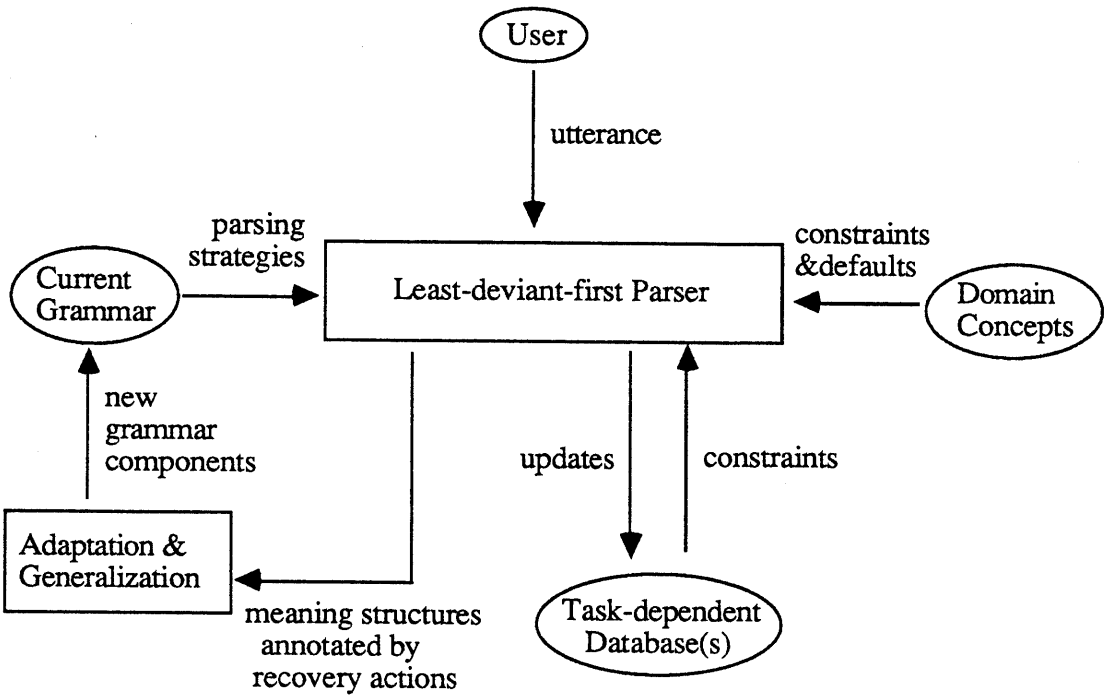

Figure 2-2: A schematic representation of the model for adaptive understanding (knowledge sources are indicated by ovals, processing components, by rectangles). The adaptation loop, in which deviations are explained by recovery actions then transformed into new grammatical components, appears at the left.

As an example of adaptive understanding let us consider the sentences in Figure 2-1 as sequential utterances from a single user. The first utterance is non-deviant with respect to the kernel grammar. As a result, a meaning can be assigned at Deviation-level 0 and the database updated to reflect the user's request. The next utterance requires one recovery action to explain the substitution of "There is" for "add". Thus, the second utterance produces two results: the database is updated and a new form is added to the grammar making "add" and "there is" synonymous. The new grammar component is generalized so that the two lexical entries act as synonyms regardless of the context in which the relation was learned (here, a <meal> followed by a marked location and a marked time). In this way, when the four deviation utterance in Figure 2-1 is encountered, only three recovery actions need be applied; the appearance of "There is" in the final sentence is now non-deviant with respect to the user's adapted kernel and the first substitution recovery in the list has become unnecessary.

What are the aspects of a user's idiosyncratic language that are amenable to adaptation? Note that although the model in Figure 2-2 makes no commitment for or against any particular kind of grammatical component, we are nonetheless limited to learning what is representable within the grammar and, more importantly, to what is explainable using our

general but knowledge-poor notions of deviation and recovery. Thus, we are theoretically free to represent the kernel so as to learn morphology, syntax, semantics, or even discourse phenomena. The degree to which the inclusion of any of these categories is appropriate, however, depends upon an affirmative response to two questions:

1. Is there regularity in the user's behavior along this dimension? For example, is she consistent in her method of abbreviation, her choice of imperative or declarative form, her word choice, or her order of subtask performance?

2. Can those regularities be captured in useful, predictive ways? In other words, can we design a representation for the grammar that, in conjunction with our model of deviance and recovery, is adequate to capture the user's actual conditions on usage?

In this thesis, we will show that an affirmative response to these questions can be given for a semantic grammar augmented with context-sensitive search constraints. The kind of grammatical components the system will learn, therefore, will be idiosyncratic vocabulary and syntactic structure.

How does our model achieve the goal of acquiring an idiosyncratic semantic grammar balancing minimal ambiguity and maximal coverage? In initial interactions with a user, we expect the number of deviant utterances to be fairly large, although the degree of initial disparity between the kernel grammar and the user's grammar clearly depends on both the choice of system forms and the user's idiosyncracies. Utterances that can be explained without too much uncertainty (for example, containing at most two deviations) will give rise to generalized user forms and thereby create an adapted kernel. By extending the grammar in response to real instances of behavior, we increase the system's linguistic coverage in useful directions while incorporating only the ambiguity intrinsic to the user's idiolect. As the adapted grammar and the user's language converge, the likelihood that an existing form serves as an adequate explanation of an utterance increases. At the same time, the average amount of search required to understand the user should decrease as more utterances are judged grammatical rather than deviant. Thus, as long as the user continues to rely primarily on those forms she has employed successfully in the past, the system's recognition capability and response time asymptotically approach the optimal performance given any stable inherent ambiguity in the user's grammar. When a novice user becomes more expert she may wish to move away from full sentences and toward the kind of terse utterances that resemble a command language. Should linguistic behavior begin to vary, there would follow a transition period of decreased system performance. The decrease occurs because the system must widen its search to explain the new, deviant utterances. As the new set of preferred forms stabilizes, however, the system's recognition capability and response time again asymptotically approach the optimal performance for the new adapted grammar.

2.3. Assumptions and Hypotheses

The model of adaptation introduced in the previous section relies implicitly on a number of assumptions. The purpose of this section is to make those assumptions explicit. Since assumptions about human behavior cannot be taken for granted, we pose those assumptions as testable hypotheses (proving the hypotheses is the subject of Chapter 3). We begin by adopting:

A1. The Single-user Assumption: Any instance of the parser is confined to interaction with a single user over time.

By assuming A1, we presuppose that the system is intended to benefit frequent users. Thus we restrict the set of pertinent tasks to those requiring recurring interaction with an individual. A1 serves as a precondition for the following hypotheses:

H1. The Regularity Hypothesis: In a natural language interface used by the same individual over time, the user will tend to rely on those forms of expression that she remembers as having worked in the past. In other words, we assume that frequent use leads the user to a *stylized* and *self-bounded* grammar.

H2. The Deviation Hypothesis: Self-bounded behavior is more appropriately described as asymptotic to a fixed set of structures. When a user deviates from her canonical utterances, she will do so in small, idiosyncratic ways.

H1 tells us that adaptation is a useful technique if we can guarantee that the user's grammar will not grow indefinitely. It leads us to expect that the amount of search performed by the system will drop sharply after an initial learning phase as the set of utterances understood by the system and those employed by the user converge. H2 reminds us, however, that we cannot simply choose a point at which to stop adapting; small changes may continue to occur sporadically. In Chapter 3 we will see strong empirical support for these hypotheses.[5]

Current natural language interfaces work, at least in part, because they restrict the search space of meaningful utterances to those to which the application program can respond. By assuming all utterances must make sense in the context of a fixed set of actions, objects, and descriptive categories, the parsing process becomes one of resolving the references in an utterance to the concepts in the domain. In other words, the utterance

---

[5]There may be times when true exploratory behavior will be employed by the user, violating H1 and H2. In effect these times are similar to the state of the parser before its first interactions with the user; many predictions based on history will not pertain. At times of exploration, as in the initial stage of use, we expect poorer performance. H1 and H2 imply that instances of exploratory behavior are relatively rare; the evidence in Chapter 3 supports this as well.

"Schedule a meeting at 4 p.m." is meaningful only if the domain concepts for **add-to-calendar**, **meeting** and **time** are well-defined. In an adaptive parser, restriction to a fixed set of referents is extremely important. For if an utterance cannot be recognized and new domain concepts can be introduced, then we are faced with the following dilemma: it will not be possible to distinguish between the case where the unrecognized utterance is a new form referring to known concepts (and thus guaranteed to be meaningful), and the case where the unrecognized utterance refers to an unknown concept (possibly one to which the application program cannot respond). Thus, we adopt:

A2: The Fixed Domain Assumption: The task domain is fully specified in the sense that no object or action meaningful to the application is unrepresented in the underlying system. The only exceptions permitted are categories explicitly designated as extendable—the members of such a category may change but the classification of a category as extendable or non-extendable may not.

Extendable categories in our domain include locations and people's names. As Kaplan [29] points out, these are the kinds of categories whose membership must be permitted to grow in a cooperative interface. If we did not include extendable categories as an exception to the Fixed Domain Assumption, every unknown segment would have to be resolved as either a substitution or insertion. Allowing extendable classes does add a certain degree of indeterminacy—an unknown segment must be considered a possible instance of a known extendable class as well as a substitution or insertion—but the benefits far outweigh the cost. It is unreasonable to assume, for example, that new people are never met and new places never visited; an interface capable of understanding real user input cannot afford such an assumption. In Chapter 5 we will show how learning new instances of extendable classes can be accomplished naturally within the adaptive parsing framework. Here we notes only that the exception to the Fixed Domain Assumption does not entail the dilemma we mentioned above. Since the number of meaningful types is fixed the introduction of a new token cannot correspond to a concept with which the application is unfamiliar.

In addition to providing a powerful mechanism for constraining search, Assumption A2 serves as the precondition for a final hypothesis:

H3: The Fixed Kernel Hypothesis: The kernel grammar for an adaptive parser need contain only a small set of system forms—in general only one form of reference for each meaningful domain action and object. Any user can be understood by then extending the grammar as a direct result of experience with the individual, using the paradigm of deviation detection and recovery in a least-deviant-first search.

The point of an adaptive parser is not just to recognize a large set of natural language utterances, but to control search by recognizing that set of utterances most likely to be

used by a particular individual. The Single-user Assumption (A1) says that we need a different instance of the interface for each user; the Fixed Kernel Hypothesis says that we need design only one initial interface for all individuals.

2.4. Adaptation vs. Customizable and Instructable Interfaces

Past solutions to the frequent user's problems have been limited to customizable or instructable interfaces. The difference between the two approaches rests upon whether changes to the interface occur during system building (customizable) or system use (instructable).

Customizable interfaces permit special vocabulary and grammatical forms to be integrated into a base system. In the case of systems like INTELLECT[6] [24, 25] and LanguageCraft[7], the extensions are added once by the interface designer. This kind of customization may make the system friendlier to the sporadic user (by increasing the probability that a naturally-occurring utterance is meaningful), but it does not actually address the frequent user's problems because the system's notion of what is grammatical cannot change over time. Although an effort is made to account for the most likely forms of reference, the user remains limited to a fixed subset of English.

Alternatively, customization may be an on-going process performed by an on-site expert, as with RAMIS-English [34], LDC [3], and TEAM [22]. The primary drawback to this kind of customization is that it suffers from the mismatch problems outlined in Chapter 1—every user pays the performance price for the customizations desired by others. In addition, dependence on an expert means a potentially lengthy wait for the user's idiosyncratic grammar to become part of the system. In contrast, adaptive parsing makes the interface itself the on-site expert—only the individual's customizations are added to her interface, and added at the moment they are needed.

Like an adaptive interface, an instructable system can change dynamically in response to a single user's demands. Unfortunately, instructable interfaces require a language of instruction (distinct from the task language) as well as an interface to talk about language—introducing yet another version of the problem we are trying to solve. In order to control the learning dialogue, some systems, such as ASK [59], NANOKLAUS [26], and UC Teacher [63] limit the instruction language to a fixed set of instructive forms. These forms, in turn, limit changes to the underlying subset of English—generally to the introduction of new vocabulary (either as synonyms or with corresponding concept

---

[6]INTELLECT is a trademark of Artificial Intelligence.

[7]LanguageCraft is a trademark of Carnegie Group, Incorporated.

definitions) or to a highly specialized set of alternate syntactic forms. In order to allow for true idiosyncratic variations in syntax, such systems would have to be extended significantly. At the very least, they would have to be given full access to their own grammar representations and an additional set of instructive forms that operate on that representation.

Even an extended instructable interface would be a poor solution to the frequent user's problems. For whenever the instruction language consists of a fixed set of instructive forms, the designer must choose those forms based upon some prior analysis. If all potential loci of change within the grammar are not equally available through the instruction language, it is possible to exclude the kinds of linguistic idiosyncracies the user finds natural. Since all loci must be available, the number of instructive forms is likely to be quite large. Presented with this diversity, a user may find the task of choosing the appropriate instruction uncomfortably complex—perhaps more complex than memorizing a fixed command language to the domain application and bypassing instruction altogether. On the other hand, an adaptive parser treats the violation of expectations at all loci uniformly, permitting syntactic as well as lexical variation and moving the burden of choosing the correct explanation of the deviation onto the system.

An alternative method for designing an instructable interface allows the user to employ natural language for the instruction task as well (for examples, see [45] and [28]). In theory, the user could teach any idiosyncratic grammatical variation she preferred. To adopt this approach one must assume that people are reflective about how they use language, that they know and can express the rules that govern their own linguistic behavior. One must also assume that the task of understanding a user's utterance about her own language is somehow less difficult than understanding a user's utterance about a restricted task domain. Neither assumption seems warranted.

2.5. Prior Research in Adaptation

Previous work in automatic language acquisition falls broadly into one of two categories: theories of child language learning that try to account for existing developmental data, or fundamentally non-cognitive models that argue for a particular learning mechanism. Both sets of research differ significantly from the model outlined in Section 2.2—either with respect to their underlying assumptions or with respect to their range of performance.

Since our model is not intended as a theory of human language acquisition, it is not surprising that it proceeds from a different set of assumptions than systems that are so intended. There are a number of psychological models that offer an explanation of aspects of the observed developmental progression in children, particularly phenomena

such as the omission of function words, gradual increase in utterance length, and order in which function words are mastered. Early research in this vein includes Langley's AMBER [36], an adaptive production system, and Selfridge's CHILD [52, 53], which evolved out of the conceptual dependency approach to language understanding [50]. The particulars of the acquisition process differ in these systems and differ from later research by Anderson [2], Berwick [5], and Pinker [47]. Yet despite various justifications, they all share one powerful assumption: the system (child) has access to the meaning of the utterance prior to any learning-related processing.[8] Even if the reasons for separating language understanding from language acquisition are well-founded in models of children, an adaptive interface's task is such that it cannot share this assumption. For an adaptive natural language interface the process of searching for the correct meaning structure and the process of learning new grammatical components are inextricably related.

In general, the second category of relevant research has been less concerned with the explanation of particular psychological phenomena than with advancing specific mechanisms or structures as integral to some aspect of language learning. While our model fits comfortably into this context, it differs from each of these systems along one or more dimensions—most notably with respect to the range of types of linguistic knowledge it can acquire.

In 1975, Miller [41] described a tic-tac-toe playing system with a natural language interface that could compensate for "inadvertent ungrammaticalities" and learn new words and expressions in its domain. Although expectation-driven, like the model we presented in Section 2.2, Miller's system lacked generality. New words were learned by one mechanism, new syntax by another, and only when an unknown verb was encountered. In addition, the system compensated for some types of ungrammaticality consistently but without learning—the constructions could not be viewed as systematic attempts by the user to employ alternate forms. In sum, the simplicity of the domain hid a number of computational difficulties that might have become apparent in a more realistic task.

Granger's FOUL-UP [20] was also expectation-driven. Relying on conceptual dependency-based knowledge structures, FOUL-UP used both script-based intersentential and intrasentential expectations to infer single word meaning. Because of the representational scheme, there were different learning procedures for different word classes. The ability of the learning procedures to recover from violated expectations varied enormously depending on the unknown word's syntactic role: FOUL-UP

---

[8]Anderson's LAS [1], which preceded the work cited above, was not explicitly a model of child acquisition and did not account for developmental data. It did, however, share the assumption mentioned.

recovered well in the face of unknown nouns, had difficulty with unknown verbs, and fell apart when confronted with unknown prenominal adjectives. Carbonell generalized much of Granger's work as part of his POLITICS system [7] [8], although he was concerned with learning only concrete and proper nouns. Extensions included adding expectations from goal-driven inferences and allowing the system to make potentially incorrect or overgeneral hypotheses and later recover from its own mistakes.

In a different vein, Berwick [4] presented an approach to single-word learning based on analogical matching between causal network descriptions of events. Similarly, Salveter [49] used successive event descriptions to extract the meaning of verbs.

Common assumptions among the single-word acquisition systems described above are that the input is both syntactically and semantically correct, and that only one word at a time will be unrecognized. In contrast, our model of adaptation contains a uniform learning procedure capable of more than single-word acquisition without those assumptions.

Davidson and Kaplan [12, 29] took a different view of single-word learning. They argued that it was unrealistic to assume a complete and unchanging lexicon for some types of databases. To aid interface users under these conditions, they used a semantic grammar to generate all legal interpretations of the unknown word, then explored category assignments in order of the estimated cost of verification. Similarly, in the presence of an unknown word or phrase, the only non-deviant interpretation our model will permit is the resolution of the segment as an instance of a known extendable class (see Section 2.3). In Kaplan and Davidson's work, however, the *only* extensions to the lexicon that were permitted were new ground instances of specific known classes (for example, a new ship name or port city); the query language itself was fixed. In contrast, our model of adaptation permits idiosyncratic vocabulary to be added to all word classes.

Zernik's RINA [67, 69, 70, 68] went beyond the single word to phrase acquisition. Specifically, RINA learned the meaning of idioms from multiple examples. Since idioms are a special kind of idiosyncratic language use, Zernik's work is of particular interest. Indeed, Zernik's approach to hypothesis formation and error correction is, in many ways, compatible with our own. Underlying RINA's design, however, is the assumption that the syntax of the utterance is correct, although it may be inadequate to resolve the meaning. Thus, an unparsable sentence is always taken as an indication of a gap in the system's semantic knowledge. With the exception of extendable classes, our model proceeds from the opposite assumption: an unparsable sentence indicates a new form of reference, not a new referent. Zernik's assumption provides his system with a strong mechanism for limiting search, just as the Fixed Domain Assumption provides a different but equally powerful method for search reduction in our model. The empirical studies discussed in the next chapter demonstrate that a fixed domain, variable language assumption is the

more realistic one for task-oriented natural language interfaces. From this point of view, RINA's approach to learning and our approach to adaptation take complementary paths, a fact we will reexamine in Chapter 10.

Berwick, too, explored multi-word constructions. In recent work [6] he has concentrated on learning finite subsystems of English (for example, auxiliary verbs and noun phrase specifiers) by induction of finite automata. To accomplish this he must assume that what constitutes a grammatical utterance can be captured at a single point in time by some core set of examples that spans the target linguistic space. He can then try successively more complex classes of automata until he arrives at the level that generates the core set without generating examples he knows to be outside that set. Thus, his techniques, though powerful, do not accord well with the conditions of language use under which an adaptive interface must perform—the minimally ambiguous set of utterances that must be accounted for does change over time (and vary across users). In addition, we could not bound the search for a class of automata using Berwick's method since we do not know *a priori* what the target linguistic space is.

Expanding our view to include learning sentential forms, we find the early work of Siklossy [56] whose system (ZBIE) induced vocabulary and syntactic structure. For ZBIE to learn it had to be given both a natural language input and a functional language input. The latter was a tree structure intended to describe the same scene referred to by the natural language input (for example, "This is a hat" would be paired with "(be hat)" and "The hat is on the table" would be paired with "(be (on table hat))"). The system used its current grammar to translate the functional language description into a hypothetical natural language utterance, replacing functional tokens with language tokens when possible, or with a token representing an unknown when no language token was available. ZBIE's primary learning strategy was to match, left-to-right, the natural language hypothesis against the natural language input, trying to resolve unknowns in the hypothesis with unmatched tokens in input. Using the primary learning strategy, ZBIE could acquire new vocabulary and more economical versions of its grammatical patterns for very simple sentences. When the primary learning strategy failed, the system had a rudimentary method for building a new grammar pattern. Although noteworthy for its scope, Siklossy's approach is incompatible in two ways with the constraints imposed on a natural language learning interface. Whereas a learning interface cannot assume grammaticality, ZBIE functioned under the assumption that the utterance was syntactically correct. More importantly, ZBIE was given both a language utterance and a meaning structure as input whereas a learning interface's search for the meaning of an utterance is its fundamental task.

Also at the sentential level is work by Harris [23]. Although not advanced as a psychological model of child language acquisition, Harris's intent was to teach his robot natural language under the same conditions children learn it. By breaking acquisition

down into distinct phases, he built a system able to learn both word meanings and sentential forms. In Phase I, the system received utterance-action description pairs and learned the meaning of individual words by building a correlation table. In Phase II, a kind of context-free-grammar was induced from a fixed set of sentences composed from the vocabulary learned in Phase I. Finally, in Phase III, the correlation table and grammar were used together to parse new utterances and respond appropriately. In order to expand the vocabulary, the system had to be returned to Phase I; similarly, to extend the grammar, it had to be returned to Phase II. This approach seems to lie somewhere between adaptation and instruction (the successor to this work, INTELLECT, was discussed in Section 2.4).

At the discourse level we find the recent work of Fink and Biermann, VLNCE [16], which notices and remembers regularities in interface behavior in order to help predict future actions. In contrast to our work, VLNCE tracks dialogue expectations in order to predict lexical items to correct for speech recognition failures. The system builds a *behavior graph* in which the nodes correspond to the meaning structures of utterances it has encountered—no record of the structure of the utterances themselves is kept. The *expectation parser* uses the behavior graph and an ATN-like network [64] to process the input. When lexical expectations are violated, the parser uses the expectations in the behavior graph to control the size of the search space during error recovery. Recovery corresponds to traversing portions of the network that correspond loosely to insertion and deletion errors; the error recovery procedures are hand-coded into the network in terms of the grammatical structure of each type of constituent (similar to MULTIPAR). The model of adaptation we have presented differs from this approach in three ways: first, it offers a uniform mechanism for error recovery that is defined independently of the grammar. Second, the model views errors as potentially reliable indicators of previously unknown regularities. In VLNCE an error is considered a transient result of the inadequacy of the speech recognizer. Finally, although this thesis examines only adaptation in a semantic grammar, we believe the same model could be used to learn discourse level phenomena as well (see the discussion in Section 2.2). In contrast, there is no readily apparent way to extend VLNCE to use grammar level predictions.

Young, Hauptmann, and Ward's MINDS system has a focus similar to VLNCE's [27] [66] [65]. Their approach also uses predictions derived from high level knowledge sources (the dialogue level, problem solving knowledge, pragmatics, and user goal representation) to limit the search space during speech processing. VLNCE uses its high level knowledge to correct errors the system generates. In contrast, MINDS uses knowledge to prevent the generation of interpretations that will violate the predictions. It does this by restricting the set of possible word choices to those that index concepts that have been predicted to be active. As with VLNCE, MINDS learns changes only to its high level knowledge sources; the syntactic and semantic information embedded in the grammar cannot change. In addition, the changes that are learned are expected to hold

only in the short-term because they reflect the fluidity of the user's goals over the course of a dialogue. Our adaptive approach takes the opposite point of view; changes to the system occur at the syntactic level of a semantic grammar and reflect long-term regularities in the user's linguistic behavior. Although it is possible to see how short-term, high level learning might be added to an adaptive interface (Section 2.2), it is less apparent how MINDS could be extended to learn long-term changes at the lexical and syntactic levels.

It is clear from our review of previous work that the model of adaptation described in Section 2.2 proceeds from a different, more realistic set of assumptions for natural language interfaces than those found in other language acquisition systems. The assumptions underlying our model enable the system to meeting the frequent user's need for a dynamic grammar. The model itself provides a uniform mechanism for constructing new grammatical components—from sentential forms down to dictionary entries—in direct reaction to the user's changing linguistic demands. The mechanism is robust enough to understand and learn in the face of multiple errors in an utterance, and to permit unknown but meaningful segments to be resolved as either synonyms or new instances of known extendable classes. The system achieves its goal in large part by taking advantage of the natural, self-bounding behavior of frequent users. In the next chapter we examine our expectations about user behavior in detail.

Chapter 3
User Behavior in an Adaptive Environment

In formulating the adaptation model in the previous chapter we made some strong assumptions about users' linguistic behavior. Before investing the resources required to implement an adaptive interface, it seemed appropriate to ascertain the validity of those assumptions. To accomplish this, we used a technique that is fairly common in evaluating natural language interface designs—the hidden-operator experiment. A hidden-operator experiment is one in which the user believes she is interacting with a computer system, when, in reality, the feedback is produced by the experimenter at a remote terminal simulating the system's behavior.[9]

In addition to allowing us to test our assumptions about user behavior in an adaptive environment, the hidden-operator experiments described below had two other uses. First, they demonstrated that the model is capable of learning different idiosyncratic grammars with a single, general mechanism. Second, the interactions between the user and the simulated system provided a set of spontaneously generated utterances that could be used as part of an evaluation of the implementation (see Chapter 9).

In the next section we state the behavioral hypotheses to be tested. Sections 3.2 through 3.4 describe the experimental conditions. In the last section of the chapter we review the results of the experiment and demonstrate the validation of our hypotheses.

3.1. The Behavioral Hypotheses

There are three conditions under which our model of adaptative parsing is an appropriate method of interface design. We have already alluded to the first condition: self-bounded language use. Without asymptotic movement toward a fixed set of structures, no idiosyncratic style will develop. In addition, if the grammar grows without bound in relation to a finite task domain, the accompanying rise in ambiguity is likely to render the system's response time intolerable.

---

[9]For examples of other studies conducted under the hidden-operator paradigm, see [19], [30], [39], [57], and [58]. The experiments and results described in this chapter were first reported in [37].

The second necessary condition is significant across-user variance. This requirement is implicit in the idea of idiosyncratic style. If all users employ the same restrictions on form and reference then a methodology such as that of Kelley [30, 31, 32] or Good *et al.* [19] is preferable; through a generate-and-test cycle one builds a single, monolithic system incorporating the complete common subset. Without natural idiosyncracy of expression, the monolithic grammar would not suffer from the mismatch inefficiencies discussed in Chapter 1.[10]

The final condition requires limited adaptability in the user. A non-adaptive approach to interface design assumes that the cognitive burden of learning the interface's sublanguage is a minor one. If the user is able to adapt quickly and with little effort to a fixed system, adaptation on the part of the system is no longer a clear advantage. Although Watt [61] has argued convincingly against the efficacy of relying on user adaptation, it is nevertheless of interest to test the claim of limited user adaptability empirically.

These three conditions correspond to an assumption that certain behaviors are characteristic of frequent users. If we require the conditions, we must show that the commensurate behaviors exist. We can satisfy the first two conditions by demonstrating that the Regularity and Deviation Hypotheses are true. These hypotheses, introduced in Section 2.3, state that with frequent interaction a user will develop, and come to rely on, idiosyncratic language patterns:

H1. The Regularity Hypothesis: In a natural language interface used by the same individual over time, the user will tend to rely on those forms of expression that she remembers as having worked in the past. In other words, we assume that frequent use leads the user to a *stylized* and *self-bounded* grammar.

H2. The Deviation Hypothesis: Self-bounded behavior is more appropriately described as asymptotic to a fixed set of structures. When a user deviates from her canonical utterances, she will do so in small, idiosyncratic ways.

To satisfy the third condition we form:

H4. The Adaptability Hypothesis: a system that permits the natural expression of idiosyncratic language patterns through adaptation will result in better task performance than a system that forces the user to adapt to it.

The series of hidden-operator experiments discussed in the remainder of this chapter

---

[10]Some empirical evidence supporting significant across-user variance can be found in [18] and [40]. Those studies, however, examined individual variation only at the word level. The experiments described in this chapter explore the degree of variation across all types of grammatical constituents.

validate our three behavioral hypotheses and give ample evidence that with frequent use the required conditions are met.

3.2. The Experimental Condition

A calendar scheduling task was chosen for the experiment because of its fairly well-defined semantics and its normal requirement of frequent interactions over time. In a hidden-operator design, users were told that they would be typing the input to a *natural language learning interface* that would increase its knowledge of English while helping them keep an on-line calendar for a busy professor/entrepreneur. In each of nine sessions, the user was given ten to twelve pictorial representations of changes to be made to the calendar. The stimuli was pictorial in order to minimize its influence on the user's forms of expression. In general, users had little trouble interpreting the *subtasks*, two examples of which are shown in Figure 3-1. The picture on the left indicates that the ending time of John's June 26 speech research meeting should be changed to 11:30. The picture on the right indicates that Flight #616 on June 28 should be cancelled, and another flight scheduled, closer to six o'clock (performing the second subtask requires at least two utterances: one to look for a replacement flight in the airline database and one to make the change to the calendar). The user was asked to effect the change in each picture by typing her commands as if she were "speaking or writing to another person." Although no time limit was enforced, the instructions told users to proceed to the next subtask after three unsuccessful tries.

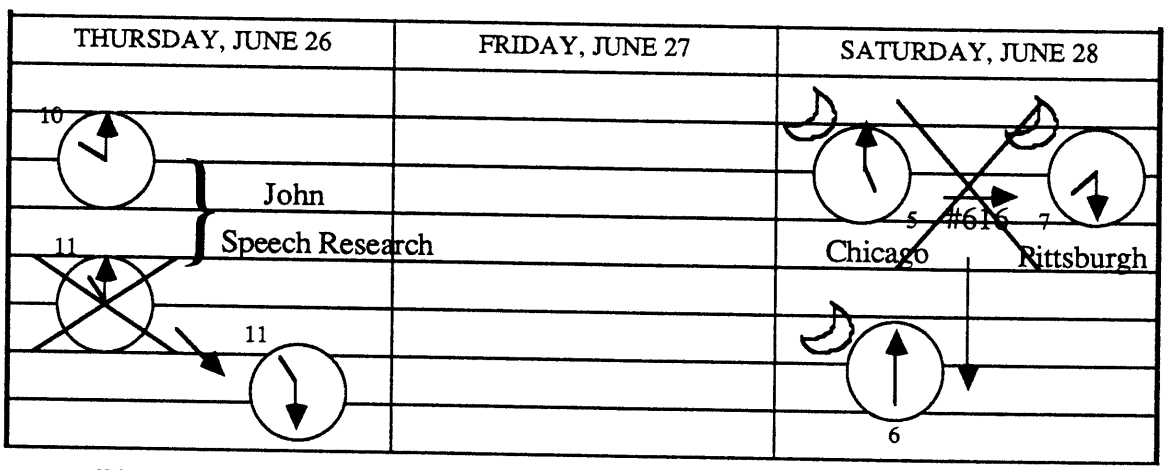

Figure 3-1: Sample pictorial stimuli for the hidden-operator experiments.

In responding to a user's utterances, the hidden operator had the user's current grammar and lexicon available. With that information, an utterance was judged either *parsable* (no deviations), *learnable* (at most two deviations), or *uninterpretable* (more than two deviations). The algorithm for making the determination and the actions taken by the hidden operator in response are presented in Figure 3-2.

1. IF the utterance can be understood using the current grammar and lexicon,
   THEN consider it *parsable* and tell the user that the action has been carried out.

2. IF there are unknown segments in the utterance,
   THEN try to resolve them as instances of extendable classes through interaction.

   2a. IF they are resolvable in this way,
   THEN add the new instances to the appropriate classes and goto 1.

3. IF the utterance can be understood by positing at most two deviations with respect to the current grammar,
   THEN consider it *learnable* and ask the user to verify the interpretation.

   3a. IF the user verifies the interpretation and the deviations are simple,
   THEN change the grammar and tell the user the action has been done.

   3b. IF the user verifies but the deviations are not simple,
   THEN tell the user the action has been done and change the grammar after the session is over.

   3c. IF the user does not verify but there is another interpretation
   THEN goto 3a.

   3d. IF the user does not verify and no other *learnable* interpretation exists,
   THEN consider the utterance *uninterpretable*, indicate to the user which portions were understood, and ask her to "Please try again."

4. IF the utterance is composed of constituents from which a legitimate database action can be inferred and within which no more than two deviations occur,
   THEN consider it *learnable* by "constructive inference," and verify as above.

5. IF most of the user's sentences were *parsable* during the session,
   THEN give her supplementary instructions at the beginning of the next session that encourage her to perform the task as quickly as possible.

Figure 3-2: Rules followed by the hidden operator in simulating the adaptive interface.

There are a number of observations to be made about the rules in the figure. First, note that Rule 2 adds new instances to extendable classes without considering the resolution as a deviation. The rationale is straightforward: as the grammar approaches its bound, the probability that an unknown segment corresponds to a new instance must increase. In the limit, all unknowns can be resolved in this way. Since new instances of known classes are exactly the sort of change that will continue to occur even after the grammar has stabilized, we want to make learning them relatively inexpensive and routine. In the algorithm above, learning new class instances is accomplished at Deviation-level 0.

Another point of interest is the limit of two deviations in a learnable utterance in Rule 3. The model presented in Chapter 2 has no inherent limit; search continues in a least-deviant-first manner until a verified interpretation is found. In practice, however, some limit is necessary to control the amount and quality of interaction with the user. In general, a sentence with more than two deviations produces a large number of poor explanations.[11]

The algorithm's definition of *learnable* (Rules 3 and 4) points to a difficulty in the experimental method. If the hidden operator's response time was to be reasonable, not all adaptations to the grammar could be done "on-line." Certainly simple deletions and substitutions could be accomplished quickly, but complex changes had to be delayed until the session was over. A change was considered complex if there were a number of ways it could be integrated into the grammar; between sessions, one new form was chosen. Sentences that were learnable by constructive inference (Rule 4) were almost always complex in this sense because the model does not specify a method of generalization. Although we will discuss this (and other) problems of underspecification in Chapter 9, we note here that the delayed adaptations for complex forms did not seem to effect materially those aspects of performance we were examining.

Finally, Rule 5 mentions "supplementary instructions" asking the user to perform the task as quickly as possible. While it has been observed that given a natural course of events users will come to employ terser, more economical language [38, 24, 31, 48], it was unclear whether nine sessions would be adequate time for the tendency to be manifested. The point of providing the additional instructions was to subject the model to more extreme linguistic behavior within the time available. Note, however, that the supplementary instructions were not given until the user's grammar had stabilized substantially (see, for example, Figure 3-3). By perturbing the stable situation, we were also able to examine the model's performance when behavior begins to vary (as discussed in Section 2.2).

---

[11]The number of explanations produced increases as a function of both the deviation-level and the relative ambiguity of the constituents hypothesized to be present. Explanations at higher deviation-levels tend to be poorer because the recovery actions are largely knowledge-free. If they encoded more knowledge, more deviation could be tolerated successfully. Unfortunately, knowledge is not only power—every piece of knowledge built into a system is also an assumption the system may be unable to relax. Chapter 11 discusses this issue further.

3.3. Control Conditions

The experimental condition described in the previous section (the "Adapt" condition) was designed to test our hypotheses about the development of self-bounded, idiosyncratic grammars. To evaluate the counterargument to adaptation that maintains that the user will naturally adapt to the system's limitations faster than the system can usefully adapt to the user's idiosyncrasies, we designed two variations on the previous experimental condition.

In the "No-Adapt" condition, the experiment was conducted as outlined above with the following exceptions:

- Users were told the system was a natural language interface (not a learning interface).
- The kernel grammar was never changed.
- No supplementary instructions were given.

Although the system remained more permissive of extragrammaticality than the average natural language interface (by allowing up to two deviations with respect to the kernel grammar), the boundaries of the grammar *were* rigidly fixed. Any improvement in performance would therefore be attributable to the user's adaptability.[12]

The "Adapt/Echo" condition was included in response to arguments by Slator *et al.* [57] that users want to learn, and will learn, a mnemonic command syntax and domain-specific vocabulary. When given the graphics command language equivalent to their natural language input, Slator's users were able to gradually integrate the terser forms into their utterances. It was unclear whether individuals would show the same propensity in interactions with an adaptive system. Users in the Adapt/Echo condition were given the same instructions as those in the Adapt condition except that they were told the system would display a paraphrase of their utterance in an "internal form" that they were free to incorporate into their commands or to ignore, as they saw fit.

---

[12]Because it was important to keep the criteria for parsability the same under all conditions, the kernel grammar for this condition was extended in one respect: use of articles was made optional. This is the most common ungrammatical construction displayed by interface users. If the grammar had not been extended in this way, most sentences would have involved more than two deviations and, without learning, the results from this condition would have been comparable almost exclusively to session one of the Adapt condition.

3.4. User Profiles

All of the users were female adults between twenty and sixty-five years of age. None had prior experience with natural language interfaces. Each was employed as a secretary or executive assistant in either a business or university environment. Although not every user maintained a calendar for her employer, each had kept a personal calendar for at least one year.

Sessions were run at approximately 24 hour intervals (except User 1 whose sessions were run twice a day with a five hour break). The number of sessions varied in the two control conditions in accordance with the availability of the users. Table 3-1 summarizes the relevant user information.

User	Years Other Calendar	Years Own Calendar	Employer	Condition	# Sessions
User 1	2.25	1	univ-psych	Adapt	9
User 2	0	1.33	company	Adapt	9
User 3	1.5	2	company	Adapt	9
User 4	2	4	company	Adapt	9
User 5	0	4	univ-compsci	Adapt/Echo	3
User 6	0	10	univ-compsci	No-Adapt	3
User 7	0	"always"	company	Adapt/Echo	5
User 8	4.5	40	univ-english	No-Adapt	5

Table 3-1: Summary of user information.

3.5. Results and Discussion

The results for users in the two adaptive conditions indicate a high degree of self-limiting behavior, converging towards a user-specific recurrent subset of English. There was virtually no overlap of derived forms at the sentential level across users; non-sentential similarities were limited primarily to missing articles and some common phrase marker substitutions. This profile of within-user consistency and across-user variability confirms the expectations in the Regularity and Deviation Hypotheses and satisfies the first two conditions necessary for an adaptive approach to interface design.

Figure 3-3 shows the approximate number of changes to the grammar over time for each of the four Adapt condition users. Figure 3-4 shows the same information for the Adapt/Echo condition. Both figures indicate, by user and session, the number of new constructions per *interpretable* sentence which is calculated as the total number of

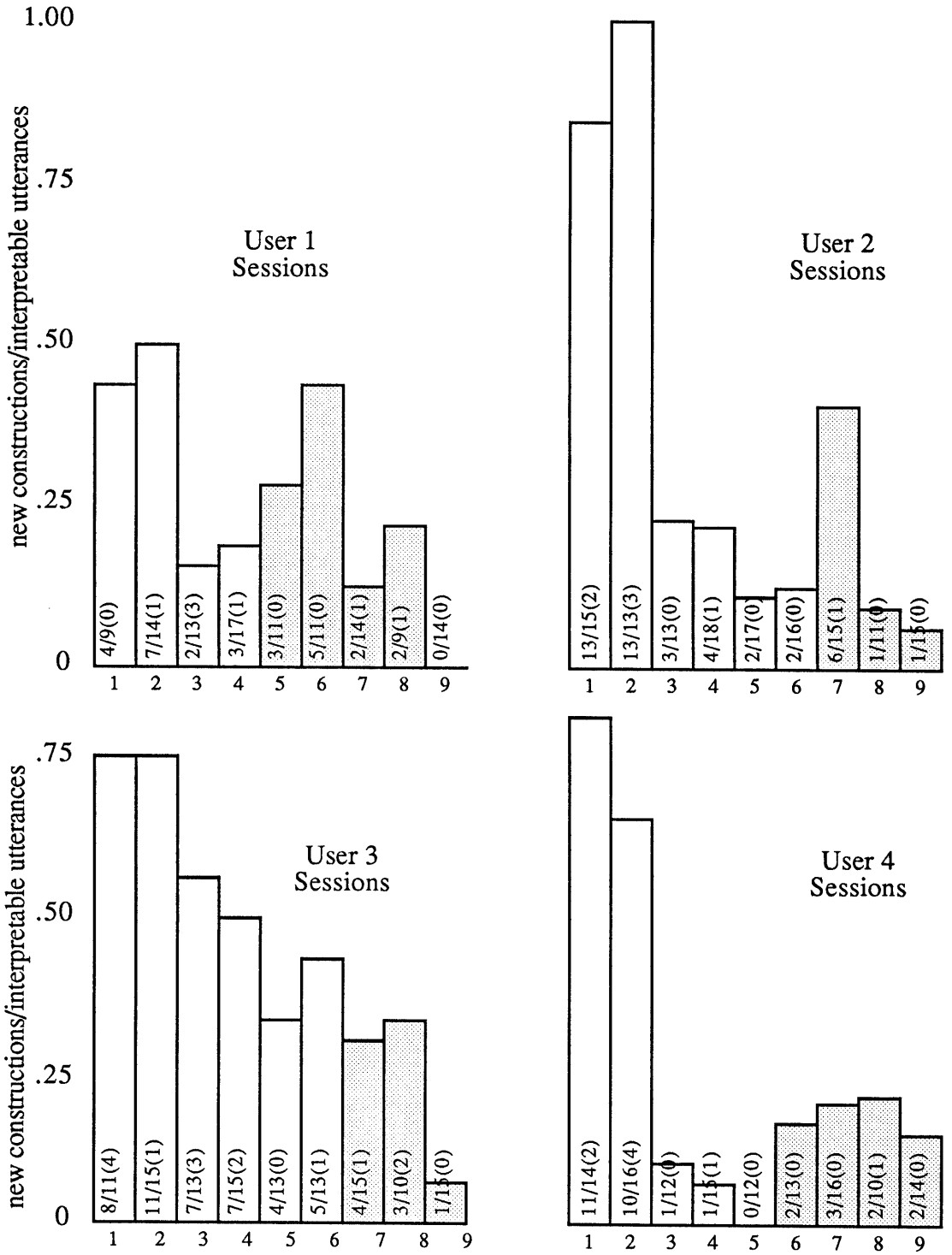

Figure 3-3: Self-limiting behavior in the Adapt condition as measured by the number of new constructions/the number of interpretable sentences (shaded sessions followed the supplementary instructions to work quickly, the number of unparsable utterances for each session is given in parentheses).

changes to the grammar divided by the total number of *parsable* or *learnable* sentences. The number of *uninterpretable* sentences in a session is given in parentheses next to the fraction. The computation of the total number of changes to the grammar did *not* include learning new instances of extendable classes because the tokens in the stimuli corresponding to unknown names were the same for all users. Shaded areas of Figure 3-3 indicate sessions following the introduction of the supplementary instructions to work quickly.

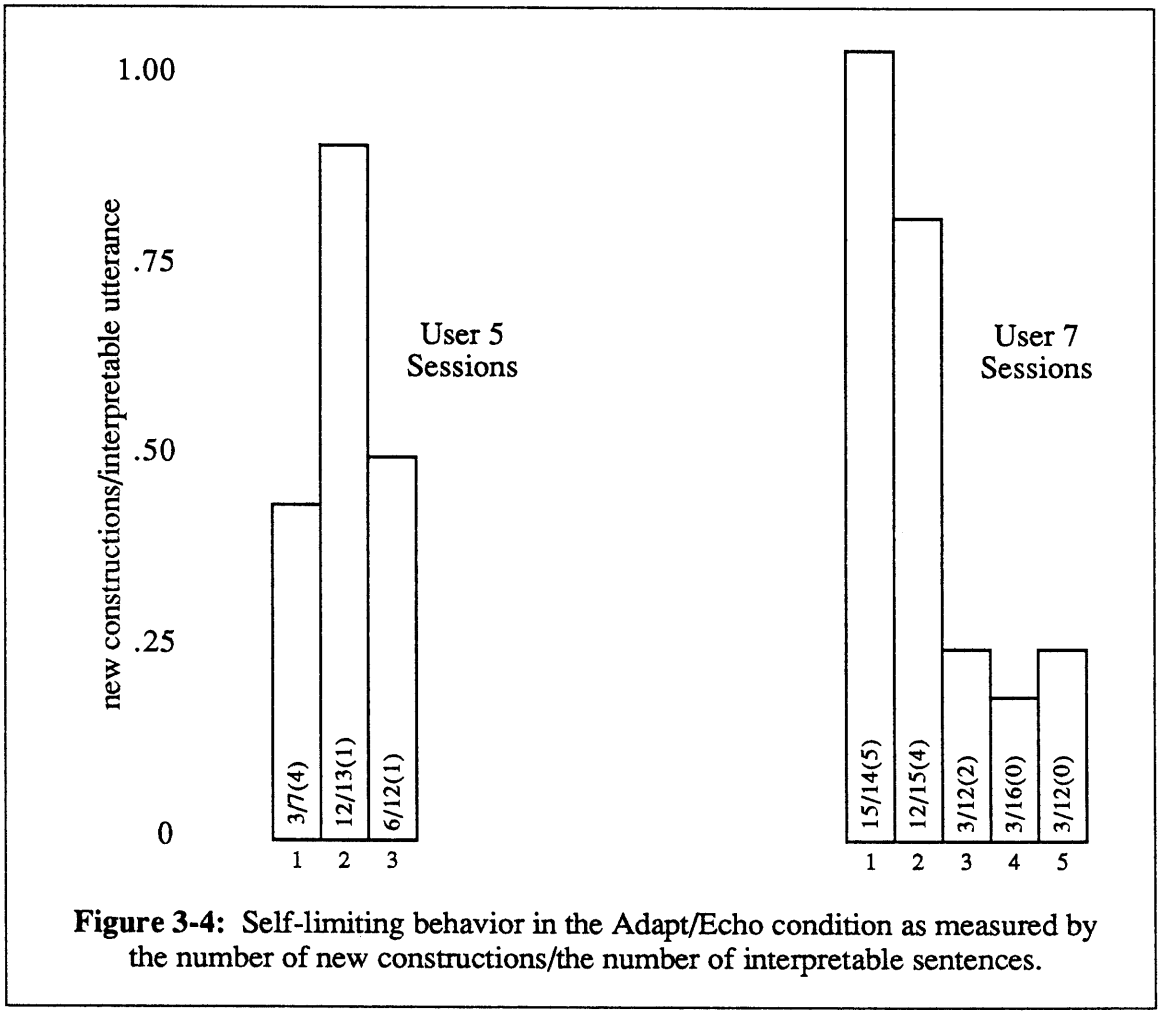

Figure 3-4: Self-limiting behavior in the Adapt/Echo condition as measured by the number of new constructions/the number of interpretable sentences.

Figures 3-3 and 3-4 show both the effect of the choice of kernel forms and the difference in the rates at which language stabilization occurs. Observe that User 1's grammar requires the least learning and stabilizes quickly. She began with terse utterances and a vocabulary that was quite close to the kernel chosen for the experiment. User 2's grammar showed less initial resemblance but she employed it consistently and thus stabilized quickly as well. User 3 had a tendency to be both polite and verbose, so her utterances were more likely to contain new forms and stabilization occurred more slowly. User 4 showed the most rapid stabilization because, although the new forms she

introduced in sessions one and two were not terse, they were quite "natural:" they were used consistently and without modification in subsequent sessions.

User 5's behavior is of limited interest as she became ill after the third session and her participation was discontinued. User 7's graph shows much the same trend as that found in the Adapt condition. Neither User 5 nor User 7 showed integration of the command language paraphrase, results at odds with Slator's general claims. Our experience did, however, conform well with a broader interpretation of Slator's claims and replicated the experiences of Good *et al.* [19] and Coboum [11]: there is a strong assumption on the part of users that whatever the system can generate it can also parse. Although neither user in the Adapt/Echo condition chose to integrate the command language provided, many of the users displayed idiosyncrasies that were clearly based on the system's output. User 7, for example, came to construct most instances of the schedule command (and only this command) using minor variations of the template: <start-time> - <end-time> <date> <meeting or meal> <other cases>, as in:

"12:00 - 1:30 June 11 lunch with Andy."

These utterances were learnable with respect to the grammar current at the time of use. Moreover, they closely conform to the template I used as hidden operator to seek confirmation of an interpretation:

Do you want:
12:00-1:30 lunch, Andy

A second set of experiments (described in Chapter 9) demonstrated that the tendency to assume that the system understands what it produces will persist despite explicit instructions to the contrary.

To illustrate what we mean by a self-limiting grammar, the evolution of User 2's referring phrase for denoting a time interval by its end points is shown in Figure 3-2. The five forms she uses to refer to an interval are given in the left portion of the figure. The actual use of a form in each session is shown as a fraction of the number of opportunities for its use. Initially, the grammar contains only the system form labelled SF. Using the general recovery actions, the experimenter equates the user forms (labelled with UF) as they appear. Note the transience of UF1 and UF2 as well as the eventual reliance on the relatively terse SF and the terser user derivative (UF3). The tendency towards abbreviated forms is symptomatic of movement toward a virtual, user-specific command language.

Figures 3-5 and 3-6 display the results for the No-Adapt condition. Recall that the purpose of this variation was to study the claim that people adapt well enough to obviate the need for system adaptation. User adaptation in this condition is measured by categorizing each utterance by the minimum number of deviations required to interpret it. If the user is adapting to the limitations of the system we should see a general increase in

Forms	Session								
	1	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*
SF: from <time> to <time>	-	2/6	-	5/9	4/4	2/3	-	6/6	1/3
UF1: from <time> till <time>	2/4	-	-	-	-	-	-	-	-
UF2a: beginning at <time> and ending at <time>	2/4	3/6	6/7	-	-	-	-	-	-
UF2b: beginning at <time> ending at <time>	-	1/6	1/7	-	-	-	-	-	-
UF3: <time> to <time>	-	-	-	4/9	-	1/3	1/1	-	2/3

Table 3-2: Self-Limiting behavior as shown by the evolution of two preferred forms for references to a time interval in User 2's grammar.

interpretable sentences (those containing zero, one or two deviations). Further, grammatical utterances (zero deviations) should come to dominate. The latter conjecture is based on the belief that slower response time and more work are negatively reinforcing. Since the interpretation of utterances with one or two deviations took longer and had to be confirmed by the user, forms in the kernel grammar should have been preferred.

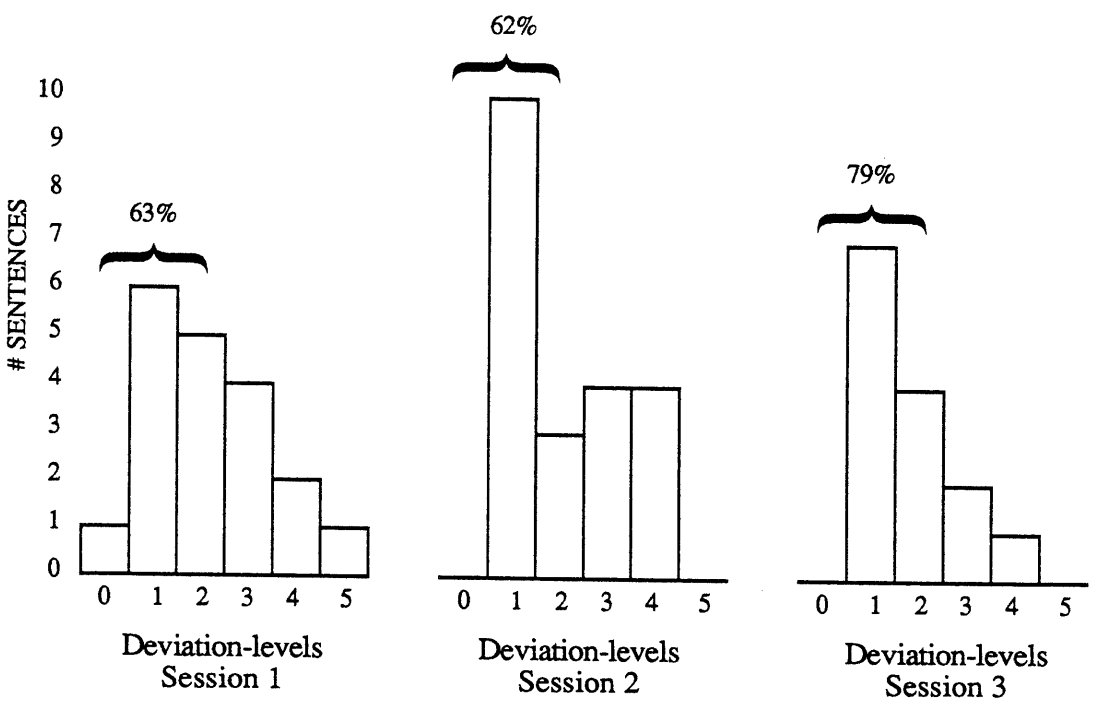

Figure 3-5: The number of utterances at each deviation-level in the No-Adapt Condition, User 6, with the percentage of interpretable utterances highlighted.

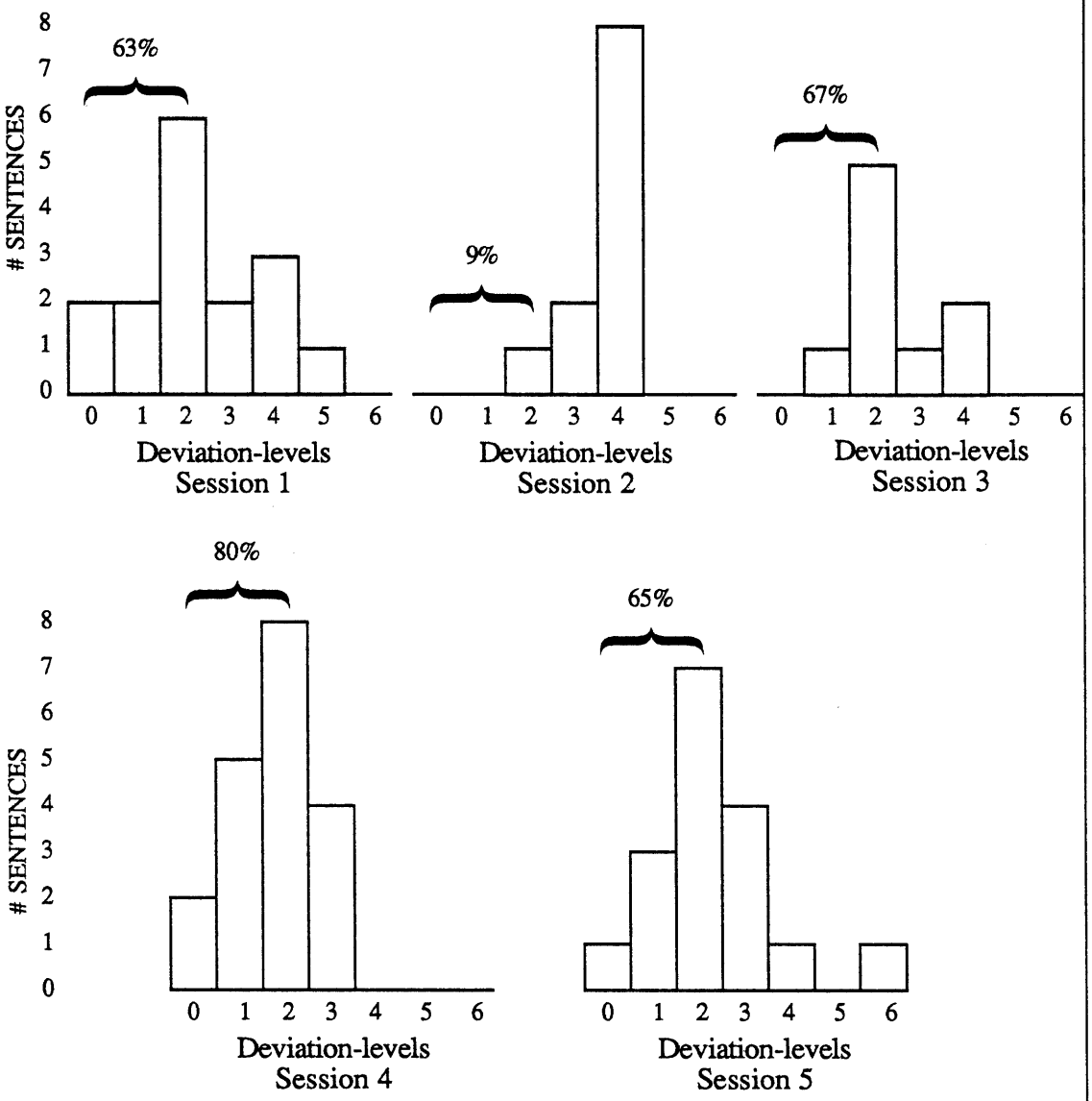

Figure 3-6: The number of utterances at each deviation-level in the No-Adapt Condition, User 8, with the percentage of interpretable utterances highlighted.

The graph for User 6 (3-5) shows a clear increase in interpretable sentences. Her behavior is interesting in two respects. First, it is reminiscent of User 1's behavior: User 6's grammar was fairly close to the kernel, especially with respect to vocabulary, and she tended to rely on terse forms. As a result, there were, in general, fewer loci for deviation. Like the users in the adaptive conditions, User 6 relied on those forms that had worked in the past. When an utterance did meet with failure, her next try was usually a minor variation that almost always succeeded. In short, User 6 performed successfully in the No-Adapt condition, but relatively little adaptation was required of her. A second point of

interest is that only one of her 54 sentences was without deviation, a fact that seems to disconfirm the conjecture that grammatical utterances would be sought. The conjectured pressure to seek grammatical forms relies implicitly on the user recognizing that response time could be faster, however. Since only one of her utterances was without deviation, it is possible that User 6 did not realize that her ungrammatical forms were causing the additional interactions. Instead she may have believed that the extra steps taken to confirm an interpretation were part of the system's normal function and not under her control.

Figure 3-6 displays the results of analyzing User 8's performance in the No-Adapt condition. There is no pattern of improvement in User 8's data; her success at adapting to the interface's subset of English was extremely limited. We will defer further discussion of User 8 until we have examined two additional measures of performance. Figure 3-3 shows the number of subtasks completed by each user in each session. Figure 3-4 shows the related metric of time taken per subtask. With respect to the former measure, users other than User 8 generally managed to effect the changes in each subtask they tried.[13] They rarely required more than three attempts per subtask in the initial sessions and averaged only slightly more than one attempt per subtask in later sessions.

A general decrease in task time is a predictable result of practice. In this experiment, however, the decrease in task time is also attributable to the users' self-limiting behaviors. Users in the adaptive conditions accomplish the task in less time as the grammar of the simulated system comes to reflect their own and more sentences are judged parsable rather than learnable. When the supplementary instructions are given (Adapt condition only) the users are implicitly encouraged to employ simpler, terser forms. As a result, task times increase as the new forms are learned. When these forms stabilize, response times fall to new lows because the shorter forms take less time to check against the grammar. User 6 shows the same general trend in time spent as those in the adaptive conditions both because her initial grammar conformed so closely to the kernel and because she is consistent in her use of what worked in the past.

User 8 shows neither consistent decline in the time taken each session nor in the number of uninterpretable sentences (see Figure 3-6). Both her work experience and the content of her utterances lead to the conclusion that little of her behavior can be attributed to task misunderstanding. Put simply, she was unable or refused to adapt.

Her linguistic style, like that of User 3, can be characterized as verbose. In response to a

---

[13]There were four sessions in which a user skipped all or part of one or more subtasks (indicated by a bracketed superscript in Table 3-3). In these cases no utterance relating to the subtask appears in the log file. Half a subtask may be skipped if, for example, a value was to be changed and the old value was removed but the new value was not added.

Experimental Condition	Session (Total Number of Subtasks Possible)								
	1 (10)	*2 (11)*	*3 (12)*	*4 (10)*	*5 (10)*	*6 (10)*	*7 (11)*	*8 (10)*	*9 (11)*
Adapt									
User 1	7[1]	11	12	10	10	8[2]	11	10	11
User 2	10	11	12	10	10	10	11	10	11
User 3	8	11	12	10	10	10	11	10	11
User 4	10	11	12	10	10	10	11	9.5[.5]	10.5[.5]
Adapt/Echo									
User 5	6.5	9	12						
User 7	10	11	12	10	10				
No-Adapt									
User 6	10	11	12						
User 8	7	1	4.5	5	6				

Table 3-3: The number of completed subtasks, arranged by user and session (''[#]'' indicates number of subtasks skipped).

Experimental Condition	Session								
	1	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*
Adapt									
User 1	5.5*	3.0	2.5	2.8	1.7+	2.4	1.6*	2.1*	1.0
User 2	5.0	3.5	2.0	3.5	2.2	2.1	2.2+	1.3	1.3
User 3	6.3	3.2	2.7	3.3	1.7	2.3	1.6+	2.0	1.0
User 4	3.4	3.3	1.7	2.1	1.1+	1.6	1.5	1.5	1.1
Adapt/Echo									
User 5	8.5	5.3	3.3						
User 7	4.7	3.8	2.5	3.3	1.5				
No-Adapt									
User 6	4.3	3.7	2.0						
User 8	6.3	31.0	7.1	5.0	5.3				

Table 3-4: The average number of minutes per subtask, arranged by user and session (''*'' indicates approximate value due to damage to log files ''+'' indicates instructions to work quickly given this session).

typical utterance, the system would tell her which segments it could parse and ask her to try again. She would then try typing exactly those segments just echoed, with no connecting text. When the terse form met with failure she would gravitate back to overly explanatory sentences.

User 8's performance in session two was so poor (in 31 minutes she produced one learnable sentence with two deviations), and her frustration so great, that she was given hints about how to use the system more effectively prior to beginning session three.[14] Although the help appeared to improve her performance in sessions three and four, in session five there is still a preponderance of unparsable forms. Contrast this with the behavior of users in the other conditions, most of whom had reduced their number of unparsable utterances to zero by session five. Finally, note User 8's low values for the number of tasks completed. The low values were due largely to the fact that she never managed to find a parsable form for an entire class of subtasks (those involving the **change** action). Although User 8 did rely on the few forms that had worked in the past, taken as a whole her performance must be seen as a strong counterargument to the notion that everyone finds it natural or easy to adapt to a system's linguistic limitations.

The purpose of the experiments described in this chapter was to establish certain behavioral characteristics of frequent users which, in turn, guarantee the conditions necessary for an adaptive interface to benefit user performance. The behavior of the users in the adaptive conditions demonstrates the self-limiting, idiosyncratic language use predicted by the Regularity and Deviation Hypotheses. As a result, we may assume that the conditions of within-user consistency and across-user variance will be met, making single-user, adaptive interfaces a practical and desirable alternative to a monolithic interface design. In the broader view, our empirical results demonstrate that self-bounded linguistic behavior is a natural by-product of frequent use. Phrased differently: we have shown that style can be viewed as a dependable source of constraint.

The experimental results validate the Adaptability Hypothesis as well. Although we cannot guarantee that a particular user will find adaptation to an interface's limitations difficult, User 8 serves as a dramatic example of the kind of limited user adaptability we must be prepared to encounter. Further, a comparison of User 6 and User 8 supports Watt's argument [61] that the ease with which a user adapts to a rigid interface depends significantly on a fortuitous correspondence between the user's natural language and the sublanguage provided by the interface designer. More importantly, the performance of the users in the adaptive conditions demonstrates that an initial lack of correspondence can be overcome by system adaptation.

---

[14]Specifically, she was told to try to type simple, but fully grammatical sentences and to think of the system as someone to whom she was writing instructions. She was the only user who was given this aid.

Having proven our conjectures about user behavior, we return to our single conjecture concerning system behavior in an adaptive environment: the Fixed Kernel Hypothesis stated in Section 2.3. Its basic claim is that the adaptation model of Section 2.2 in conjunction with a small kernel grammar creates an interface design responsive to the frequent user's needs. In order to validate the Fixed Kernel Hypothesis we now turn our attention to a particular implementation of the model we call CHAMP.

Chapter 4

System Architecture and Knowledge Representation for an Adaptive Parser

In Chapter 2 we considered adaptation as a general solution to the problem of natural language interface design. The model we presented was adequate to pinpoint certain implicit assumptions in our approach and to generate a number of testable hypotheses. At the same time, however, the model left many important components underspecified and many practical issues unresolved. In Section 3.2, for example, we noted that without a specific generalization method, multiple ways of integrating a derived form into the grammar were possible. As a result, the hidden operator simply chose an appropriate representation for the derived form without considering the limitations a particular generalization method would create. Similarly, the algorithm for categorizing an utterance (also presented in Section 3.2) required only that some minimally deviant correspondence between input and grammatical form be found; the algorithm did not take into account the computational complexity of finding it.

Because the simulation was being performed by a person who understood the utterances, the "system" usually behaved both correctly and efficiently when the model underspecified the action to be taken. A derived form, for example, was represented in the most useful way possible given the structure and contents of the grammar at that moment. Thus, the conjectures about user behavior were validated by experiments in which an "ideal" system was being simulated. To validate our remaining conjecture, the Fixed Kernel Hypothesis, we must address the creation of an actual adaptive parser and determine whether its behavior can be sufficiently close to the ideal along relevant dimensions. We have implemented the model as an adaptive interface named CHAMP.[15] Of course any implementation of a model forces the designer to make specific what was previously abstract. The particular set of choices embodied in CHAMP, as well as the implications of those choices and their behavior in a running system, are the subjects addressed in the remainder of this dissertation.

---

[15]The system is written in COMMON LISP and currently runs on an IBM RT. The acronym stands for CHAMeleonic Parser.

Figure 2-2 presented a simple schematic diagram of the components and processes of the adaptation model in Chapter 2. Figure 4-1 represents the same components and processes as they have been realized in CHAMP.

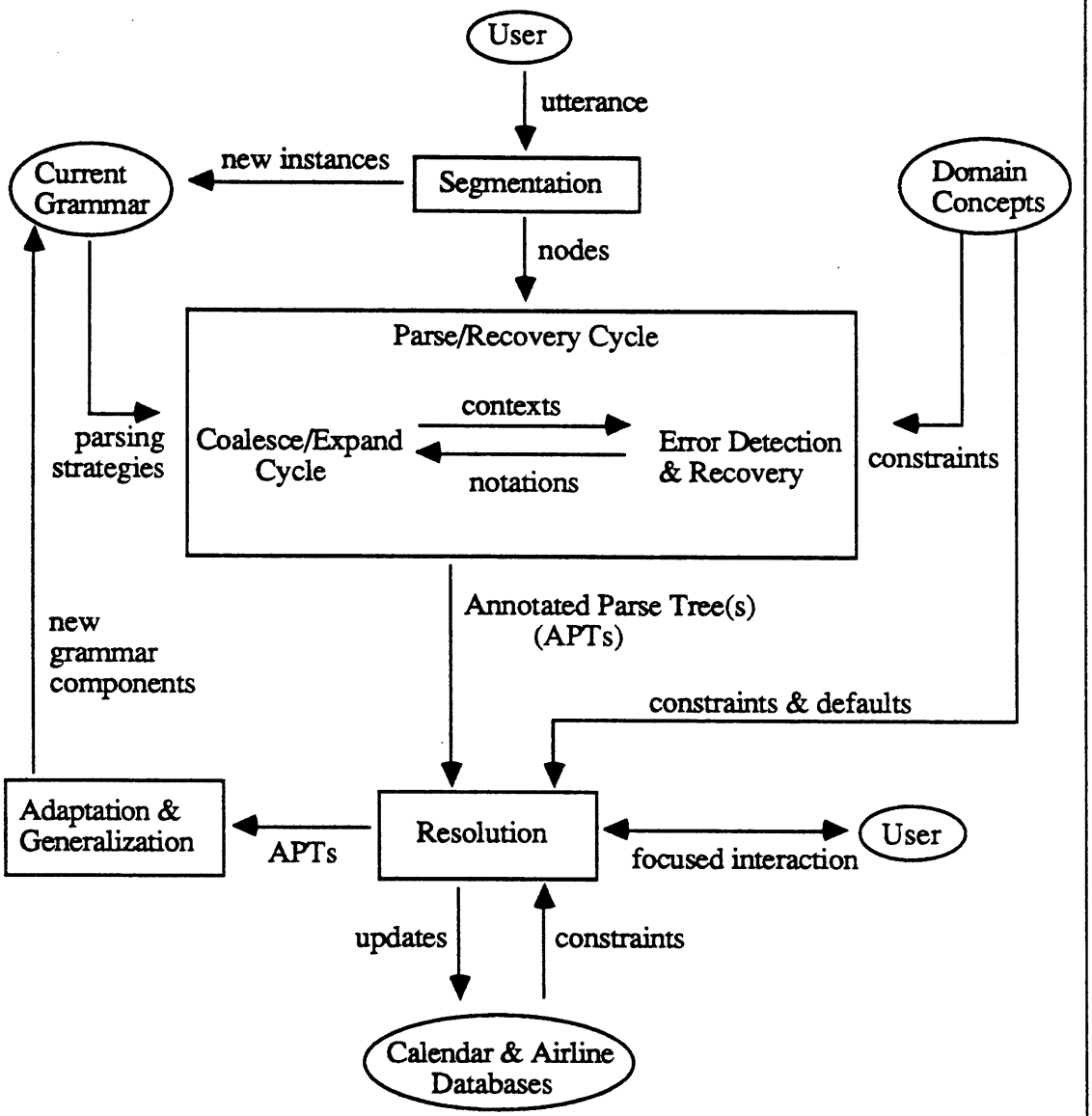

Figure 4-1: A schematic representation of knowledge and control in CHAMP (knowledge sources are enclosed in ovals, processes in rectangles.)

A comparison of Figures 2-2 and 4-1 shows that the original two-step understanding process has been decomposed into four distinct phases: Segmentation, the Parse/Recovery Cycle (which is made up of the Coalesce/Expand Cycle and Error Detection & Recovery), Resolution, and Adaptation & Generalization. Considered as a distinct unit (see Chapter 5), Segmentation and the Coalesce/Expand Cycle comprise a

bottom-up parser capable of learning new instances of known extendable classes. When we embed the Coalesce/Expand Cycle inside the Parse/Recovery Cycle, however, we create a least-deviant-first parser by integrating Error Detection & Recovery into the Expand phase of the bottom-up algorithm (see Chapter 6). Error Detection & Recovery has the dual responsibilities of catching violated expectations in a candidate grammatical constituent (called a *context*) and augmenting the parse tree with *recovery notations*.

When the Parse/Recovery Cycle produces more than one annotated parse tree (APT) we consider the set of APTs as competing explanations of the utterance. Resolution (Chapter 7) is responsible for selecting the intended meaning from this set and performing the user's requested action. To accomplish resolution CHAMP may require guidance from the user, or may need nothing more than the additional constraints provided by the entries in the databases.

Once a meaning has been assigned to a deviant utterance, the system is ready to adapt the grammar so that the previously unknown form can be recognized directly in the future as a way of referring to the semantic structure. Since it is unlikely that any particular utterance will appear repeatedly, word for word, memorization would be of little use. Instead, we need the changes to the grammar to reflect an appropriate degree of generalization. In addition, any successfully understood utterance (deviant or not) may contain information that can help eliminate prior generalizations that were incorrect. Once Adaptation & Generalization (Chapter 8) has been accomplished, the system is ready for the next interaction.

Although the decomposition of processing steps has changed, the ovals in the figure show that CHAMP contains the same four sources of knowledge present in the model: the User (at two points), Current Grammar, Domain Concepts, and Databases (specifically, the Calendar and Airline Databases from the experiments described in the previous chapter). The knowledge structures associated with database interaction are discussed in the context of Resolution, in Chapter 7. The representations chosen for the grammar and the domain concepts are discussed in the remaining sections of this chapter.

4.1. Representing Knowledge in CHAMP

Considered from the system's point of view, CHAMP performs two tasks. The first is to aid in the duties of an executive assistant to a professor/entrepreneur. The duties consist of helping to maintain a schedule of meetings and events, and helping to arrange airline reservations when events require travel. CHAMP's second and less visible task is to extend its grammar to understand more effectively the idiosyncratic language of a particular user. Because the tasks are distinct, it is useful to distinguish between two types of knowledge in CHAMP: the learning-related structures that comprise the

grammar, and the application-related structures that organize the domain concepts and databases. This division is not one of syntax versus semantics—both types of structures contain what is usually thought of as syntactic and semantic information. The important distinction is between what can and cannot change as a result of adaptation. As a consequence of the Fixed Domain Assumption (Section 2.3), only those structures that make up the grammar can be the target of learning.

To help the user schedule events, the number of general actions and object types the system must know about (the Domain Concepts in Figure 4-1) is fairly small (about fifty). On the other hand, the number of specific objects, such as particular individuals and locations, is unconstrained via the use of extendable classes. In keeping a calendar, CHAMP limits the available actions to viewing some portion of the current schedule or adding, deleting, or changing the value of an entry. Objects that can be referred to include meetings, seminars, specific times of day, time intervals, types of locations (businesses, buildings, rooms), participants, and topics, to name a few. To schedule travel events (which may be by car or plane) the four actions remain the same, but objects such as the airline schedule, flight numbers, cities, and arrival and departure times must be added.[16]

Although the number of types of actions and objects that can be referred to is small, the possible number of ways of referring to them is not. In other words, the size of the domain has little effect upon the degree of linguistic complexity CHAMP might encounter. Consider, for example, possible initial variation between different users expressing the same request:

User A: Schedule a 2:00 - 3:00 meeting with the Robotics Group.
User B: Add a Robotics Group meeting from 2 p.m. to 3 p.m.

User A: Show me the airline schedule leaving NY and arriving in Pittsburgh.
User B: Display flight information for flights from New York to Pittsburgh.

User A: Cancel my 3 o'clock appointment.
User B: Delete the meeting beginning at 3:00 p.m.

User A: Reschedule the meeting at 4 to 10 am.
User B: Change the starting time of the 4 p.m. meeting to 10 a.m.

User A: Change the 10 am meeting's location to John's office.
User B: Change the location of the meeting at 10 am to John's office.

[16]In order to demonstrate the generality of the system, CHAMP's knowledge structures were originally developed for the calendar task without trip scheduling—the "calendar domain." After a complete working version of the system had been built, structures for the travel concepts and travel grammar were written as if scheduling the professor/entrepreneur's trips was an independent task—the "travel domain." As a result, CHAMP can be run with the calendar domain, the travel domain, or both.

We know from the experiments described in Chapter 3 that even for a single user reference to the same events may change over time. User A's language, for example, might develop as follows:

Day 1: Schedule a 2:00 - 3:00 meeting with the Robotics Group.
Show me the airline schedule leaving NY and arriving in Pittsburgh.
Cancel my 3 o'clock appointment.
Reschedule the meeting at 4 to 10 am.
Change the 10 am meeting's location to John's office.

Day 14: Schedule 2-3 Robotics Group
Show flights from NY to Pittsburgh
Cancel @ 3
Reschedule @ 4 to @ 10 am
Relocate to John's office

Day 28: Sc Robotics Group 2-3
flights NY to Pgh
C 3
Res 4 to 10 am
Rel John's ofc

The implementation of the understanding and adaptation processes in CHAMP is independent of any particular domain. Thus to understand or to learn to understand these kinds of utterances, CHAMP must be given information about how the user wants to refer to the actions and objects meaningful to the task, as well as information about the actions and objects themselves. Because we assume that the domain remains fixed once loaded (except for extendable classes), only the forms of reference can be changed by adaptation. The structures that capture forms of reference (and new instances of known classes) comprise the system's grammar. Specifically, learning-related knowledge is organized by a lexicon and a Formclass Hierarchy, both described in the next section. Information about the actions and objects themselves, information that cannot be changed and is independent of any particular form of reference, is organized by the Concept Hierarchy, described in Section 4.3.

4.2. Learning-Related Knowledge: The Lexicon and Formclass Hierarchy

The best way to understand the organization of a grammar in CHAMP is by examining part of such a grammar. Although two separate kernel grammars have been implemented, we will use examples taken primarily from the calendar domain in the discussion that follows. The upper portions of the figures that introduce the next three subsections (Figures 4-2, 4-5, and 4-8) display a small portion of the calendar kernel as it is organized in CHAMP's caseframe representation [15]. The representation in the figures has been simplified slightly for expositional purposes; both kernels are presented in the exact

representation used by CHAMP in Appendix A. The reader is encouraged to examine Figures 4-2, 4-5, and 4-8 briefly before continuing.

The figures show that the grammar is created from the following five types of components:

- **Formclass**: the component that organizes all the syntactically distinct ways of referring to a concept that are recognized by the grammar.
- **Form**: the component that represents a particular way of referring to a concept.
- **Wordclass**: the component that organizes all the lexically distinct ways of referring to a concept that are recognized by the grammar.
- **Step**: the component that assigns a formclass or wordclass to a subsegment in the utterance.
- **Lexical definition**: the component that ties a particular word or phrase to one or more wordclasses.

Each of these types of component is discussed fully in the subsections that follow this overview. It is clear from the figures that the grammar is composed primarily of semantic categories, although there are a few classes (such as **m-intervalforms** which represents a time interval marked by a preposition) that reflect syntactic distinctions.

To establish a frame of reference, the lower portions of Figures 4-2, 4-5, and 4-8 show those aspects of the caseframe grammar that can be expressed using standard BNF notation. Although we will study both representations in detail, a high-level comparison reveals that a form is comparable to a non-terminal whose expansion is to be matched against the utterance. The steps taken by a form to recognize a segment of the utterance correspond to looking for the constituent wordclasses and formclasses in the expansion. A wordclass is like a non-terminal that expands to only terminals; recognizing a wordclass means matching actual words and phrases in the lexicon. A formclass, on the other hand, is like a non-terminal that expands to the names of forms; recognizing a constituent formclass means using a form in the constituent class to recognize a subsegment.

Do the two sets of structures—caseframes and BNF rules—recognize the same language? Not exactly. As we examine each of the five types of components in CHAMP's grammar in turn, we will find that five of the fields in CHAMP's caseframes have no BNF correlates. The *ig* field that may be associated with a formclass and the *def* field that may be associated with a lexical definition facilitate the recognition task without changing the language recognized. In contrast, the *concept* and *active* fields associated with a formclass and wordclass, and the *bindvar* field associated with a step allow CHAMP to apply context-sensitive constraints during parsing. As a result, CHAMP recognizes a more semantically consistent language than a context-free BNF

grammar will permit. Our comparison of the two representations will also uncover the significant reduction in grammar size afforded by a form's *unodes*, *mnodes*, and *rnode* fields.

4.2.1. The Formclass

actionforms	**addforms**	**m-intervalforms**	**u-intervalforms**
isa root	isa actionforms	isa root	isa root
members (addforms)	members (ACT1)	members (INT0)	members (INT1 INT2)
locations *root*	locations *root*	locations (702)	locations (206)
agenda-level 8	agenda-level 7	agenda-level 2	agenda-level 1
active t	active t	active t	active t
concept action	concept add	concept interval	concept interval
ig (ig-act res)			ig (ig-act exp)

<actionforms> ::= <addforms>
<addforms> ::= <act1>
<m-intervalforms> ::= <int0>
<u-intervalforms> ::= <int1> | <int2>

Figure 4-2: Examples of formclass definitions found in the calendar domain displayed in CHAMP's caseframe representation and those portions of the caseframes that can be expressed in standard BNF notation.

A formclass is a grammatical component that organizes all the syntactically distinct ways of referring to a concept that are recognized by the grammar. The upper portion of Figure 4-2 shows that a formclass may have associated with it seven pieces of information. The first, given by the *isa* field, is the parent formclass. In the portion of the kernel displayed, only **addforms** stands in a hierarchical relationship with another class (**actionforms**). The other formclasses shown—**actionforms**, **m-intervalforms**, and **u-intervalforms**—are roots of some of the subtrees that comprise the Formclass Hierarchy (discussed below). The second piece of information in a formclass definition is *isa*'s inverse: *members*. Thus, **addforms** *isa* **actionforms** just as the *members* of **actionforms** include **addforms**. This inverse relationship holds across different types of components in CHAMP: formclass and formclass, formclass and form (compare, for example, the *members* field of **addforms** with the *isa* field of ACT1 in Figure 4-5), and wordclass and lexical definition (see Figure 4-8). In the lower portion of Figure 4-2 the *isa* and *members* relations are captured in BNF.

Just as the *isa* and *members* fields tie formclasses to other formclasses and to forms, the *locations* field ties formclasses to steps. *Locations* is the inverse of the steps' *class* fields (see Figure 4-5); the value in a formclass's *locations* field is simply the union of all the step numbers that assign that formclass to a subsegment in the utterance. Thus, the set of *locations* associated with **m-intervalforms** contains only step 702 which, in turn, has **m-intervalforms** as its *class*. These relations are left implicit in the BNF notation. We

will see in Chapter 5 that the *locations* field holds information used to expand partial parse trees.

The next field, the*agenda-level*, ties a formclass into the partial ordering of all formclasses called the Formclass Hierarchy. The purpose of the hierarchy is to organize the search for constituents during parsing. Thus, the structure of the hierarchy is derived from the relative embeddedness of constituents in the grammar. Strictly speaking, CHAMP's grammar is a forest of trees defined by the *isa* fields of the formclasses and forms (formclasses without parent classes serve as the roots of the trees). Relative embeddedness is particularly easy to see in the BNF rules: since <addforms> expands to <act1> (Figure 4-2), and <act1>'s expansion contains <m-intervalforms> (Figure 4-5), <m-intervalforms> must be lower in the hierarchy than <addforms>. Figure 4-3 demonstrates that to recover the same information from the caseframe representation we must follow the relevant *isa* relations and replace the step numbers in a form's *strategy* with the *class* assigned by the step. Thus, step 702 in ACT1's strategy list indicates that ACT1 may contain as a constituent a member of **m-intervalforms**. Alternatively, we say that step 702 "seeks" an **m-intervalform**. Since ACT1 *isa* **addform** and ACT1 contains step 702, **m-intervalforms** is lower in the Formclass Hierarchy than **addforms**; in a bottom-up parser we must recognize marked intervals before we try to recognize references to actions. Similarly, the formclass **u-intervalforms** is lower than the formclass **m-intervalforms** because part of recognizing a marked interval involves recognizing an unmarked interval (step 206 of INT0).

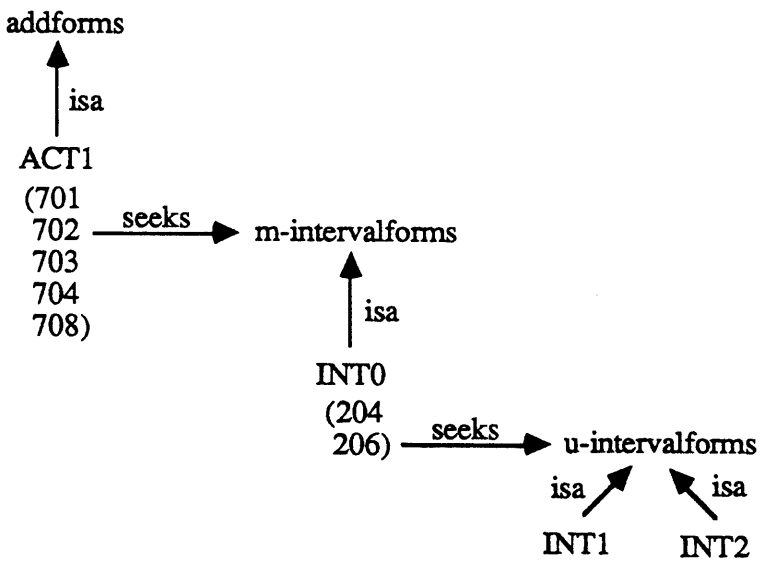

Figure 4-3: An example of the relationships among grammatical components that define the Formclass Hierarchy.

The final three fields in a formclass caseframe contribute to the application of context-sensitive constraints during parsing. The role of the *active* field is explained when we discuss segmentation in Section 5.1. The *concept* field is a connection between learning-related knowledge and application-related knowledge. Any information that might help recognize a reference to the **add** action or the **interval** object, but that cannot be changed by adaptation, is accessible through the token **add** or **interval**, respectively. For example, the **interval** concept contains the knowledge that the starting hour of an interval must be strictly less than the ending hour. This prevents CHAMP from interpreting "5 to 3" or "5 to 5" as a time interval even though the BNF rule for <u-intervalforms> permits such an interpretation. Note that both the classes **m-intervalforms** and **u-intervalforms** point to the same concept; what is true about intervals is true regardless of the presence or absence of a preposition.

The *ig* field (short for "instance generator") contains the name of a LISP function. The purpose of an ig is to create a canonical value for a segment in the formclass. **Ig-hour**, for example, tries to convert the segment of the utterance recognized as a **u-hourform** into a number between zero and 2400. In this way, each of "12:00 p.m.," "12 pm," and "noon" becomes the same canonical value, **1200**. If the information in the segment is incomplete, **ig-hour** produces as its canonical value a number with a question mark (such as **(1200 ?)** for "12:00"). Thus, a canonical value distills the important information in a constituent into a convenient form. Some igs (such as **ig-hour**) have an **exp** token associated with them in the formclass field; this means they are invoked during the Expand phase of the Coalesce/Expand Cycle. Alternatively, an ig may be invoked during the Resolution phase (via a **res** token) in order to help convert an annotated parse tree (APT) into the canonical form expected by the database functions. **Res** igs (such as **ig-act** in Figure 4-2) are discussed in Chapter 7.

Figure 4-4 demonstrates how a canonical value can facilitate the application of semantic constraints. The ig for **u-intervalforms**, for example, uses the canonical values previously generated by **ig-hour** for its **starthour** and **endhour** to represent the interval in terms of a twenty-four hour clock. Once this has been done, the semantic constraint that assures that the **starthour** precedes the **endhour** can be applied (the constraint application process is explained in Section 4.3). In Figure 4-4 the constraint will be satisfied because **ig-int** creates the interval **(1500 1630)**. Checking the constraint requires only a simple test because the relevant information is represented in a way that is independent of the actual referring phrases.

The main purpose of a formclass is to organize the information that applies to all its members. Although a formclass's *members* may be other formclasses (see **actionforms** in Figure 4-2), most of the time the *members* are forms, the grammatical component we examine next.

Utterance: "Schedule an AI seminar from 3 p.m. to 4:30 on June 5"

IG	Invoked During	Argument Bindings	Canonical Form Result
ig-hour	parsing	hr=3, minutes=0, dnbit=pm	1500
ig-hour	parsing	hr=4, minutes=30, dnbit=nil	(430 ?)
ig-int	parsing	starthour=1500, endhour=(430 ?)	(1500 1630)

Figure 4-4: The role of instance generators in providing canonical values for constraint application during parsing.

4.2.2. The Form and its Steps

ACT1
isa addforms
strategy
(701 702 703 704 708)
mode (704 708)
unodes ((701 702 703))
mnodes ((701 702))

701: class m-hourforms
702: class m-intervalforms
703: class m-dateforms
704: class addwd
708: class m-i-ggforms

INT0
isa m-intervalforms
strategy
(204 206)
mode (204 206)

204: class ivlmkr
bindvar marker
206: class u-intervalforms
bindvar (ivl divl)

INT1
isa u-intervalforms
strategy
(107 108 109)
mode (107 108 109)

107: class u-hourforms
bindvar starthour
108: class ehrmkr
bindvar emarker
109: class u-hourforms
bindvar endhour

INT2
isa u-intervalforms
strategy
(110)
mode (110)

110: class intervalwd
bindvar instance

<act1> ::= ([[<m-hourforms> | <m-intervalforms>] [<m-dateforms>]] |
[[<m-dateforms>] [<m-hourforms> | <m-intervalforms>]])
<addwd> <m-i-ggforms>
<int0> ::= <ivlmkr> <u-intervalforms>
<int1> ::= <u-hourforms> <ehrmkr> <u-hourforms>
<int2> ::= <intervalwd>

Figure 4-5: Examples of form and step definitions found in the calendar domain displayed in CHAMP's caseframe representation and those portions of the caseframes that can be expressed in standard BNF notation.

A form is a declarative structure that is interpreted by the system as an algorithm (recognizer) for detecting references to the formclass's concept. The upper portion of Figure 4-5 displays some of the kernel forms that are members of the formclasses in Figure 4-2. The definitions of the *strategy* steps that make up the algorithm are given in the middle of the figure. Steps are implemented as distinct structures so that they may be shared among strategies. We will see in the next chapter how sharing steps helps

eliminate redundant search during a parse. The lower portion of Figure 4-5 shows those portions of the caseframes that can be expressed in BNF.

To understand the correspondence between a form and its BNF representation we must first understand the role of the strategy step. The purpose of a step is to assign a *class* to a constituent that may span a number of contiguous segments in the utterance. The process of assigning a class to a set of contiguous segments is similar to that of substituting a non-terminal for its expansion in a context-free parser. Thus, in Figure 4-5, it is not until each of steps 107, 108, and 109 is satisfied by a portion of the input that an instance of **u-intervalforms** has been found by INT1. Similarly, finding <u-hourforms>, <ehrmkr>, and another <u-hourforms> tells us we have found <int1>. At this point, however, the differences in the information available from each of the representations changes the nature of the understanding process. As we saw in Figure 4-4, in CHAMP the *concept* **interval** associated with the class **u-intervalforms** imposes semantic constraints on candidate intervals. The tokens in a step's *bindvar* field are used in the definition of those semantic constraints. In this way the predicate associated with **interval** can enforce the relation **starthour < endhour**. Of course, no processing of this kind occurs in a context-free parser.

Expanding our view from the strategy step to the form itself, let us first compare INT0 and <int0>. INT0's *strategy* list contains two steps: 204 followed by 206. The step's definitions indicate that step 204 is satisfied by finding a member of the wordclass **ivlmkr** in the utterance, while step 206 seeks a member of the formclass **u-intervalforms**. Since both steps are in the *rnode* field of the INT0 form, they are both required if we are to be certain we have found a marked interval. In other words, we have found an **m-intervalform** using the strategy in INT0 if we can find an **ivlmkr** followed by a **u-intervalform** in the sentence. In BNF we capture the same idea by writing:

<m-intervalforms> ::= <int0> and <int0> ::= <ivlmkr> <u-intervalforms>.

In general we say that a form succeeds if all the steps in the *strategy* list are satisfied by contiguous portions of the utterance occurring in the given step-order. Exceptions to this rule are expressed by the annotation nodes: the *rnode* stands for "required," the *unode* for "unordered," and the *mnode* for "mutually-exclusive." More precisely, the *rnode* field overrides the need for all steps to be satisfied; a strategy may still succeed as long as those steps on the *rnode* list are matched. Similarly, the *unodes* field overrides the ordering of the strategy list; steps in a unode's sublist may be satisfied by any permutation of corresponding contiguous segments in the utterance. The *mnodes* field demands that steps in its sublists be considered mutually-exclusive—only one of the set may appear.[17] As an example of the interpretation of annotation nodes within a parsing

[17]There is one more type of annotation node which is not shown in Figure 4-5: the *snode*. It is discussed in the context of the Coalesce/Expand Cycle, in Section 5.2.

strategy, let us consider ACT1. Figure 4-6 displays the variation in utterances recognized by the form. We can summarize the algorithm represented by ACT1 as follows:

1. If there are contiguous references to the marked forms of a date, an hour, or a time interval, they are part of the reference to the concept **add**. Any order is permitted among the introductory adverbial phrases, but only one of **m-hourforms** or **m-intervalforms** may be present.
2. Regardless of the presence or absence of a date, hour, or interval, an **addwd** must be present. If any of the introductory adverbial phrases are present, they must precede the **addwd**.
3. A group-gathering marked by an indefinite article (**m-i-ggforms**) must be found following the **addwd**.

Form	Strategy Steps	Annotations
ACT1	701 m-hourforms	mutually-exclusive with 702, unordered with 703
	702 m-intervalforms	mutually-exclusive with 701, unordered with 703
	703 m-dateforms	unordered with 702 and 701
	704 addwd	required
	708 m-i-ggforms	required

Examples of Directly Recognizable Utterances:

- "On June 4 at 5 p.m. schedule lunch with John."
- "From noon to 1:30 on June 7 schedule a speech research meeting."
- "On June 12 add a seminar in Room 5409 from 3 to 4."
  (the marked interval is picked up by m-i-ggforms in step 708)

Examples of Utterances Requiring Recovery:

- "Schedule on June 4 a meeting with Alice."
  (steps 703 and 704 out of order)
- "On June 7 at 5 p.m. [add] a natural language interfaces seminar."
  (required step 704 omitted)
- "At 6 from 6 to 7 p.m. schedule dinner with dad."
  (mutually-exclusive steps 701 and 702 both present.)

Figure 4-6: The **addform** ACT1 considered as a parsing strategy: examples of non-deviant utterances it recognizes directly, and deviant utterances it fails to recognize without error recovery.

At first glance it seems trivial to capture the "required," "unordered," and "mutually-exclusive" relations in BNF. Square brackets indicate optionality and therefore, indirectly, those elements that are required. The Kleene star permits both optionality and permutation by matching zero or more elements from a set in any order. The vertical bar represents mutual-exclusion among elements that may fill the same position in the production. Unfortunately, the ways in which the "unordered" and "mutually-

exclusive'' relations are manifested in natural language makes a trivial translation from *unodes* and *mnodes* to star and bar impossible.

Consider the single *unode* associated with ACT1; it says that the hour, date, and interval may occur in any order. Implicit, however, is that only one of each constituent may be present. Thus, translating the *unode* using the Kleene star is inappropriate because the star allows, by its definition, multiple dates, hours, and intervals to be matched in the input. To avoid an overly general production but still capture the meaning of the *unode* in ACT1, we must explicitly include in the BNF all the permissible orderings of the unordered steps in the caseframe representation. The *mnode* relation between the hour and interval means that the true number of constituents in the unordered set is two. In general, if there are n unordered constituents then there will be $n!$ terms in the disjunct representing the permissible orderings. Figure 4-7 shows that the appropriate translation from ACT1 to <act1> results in a top-level disjunct with two terms.

In the case of ACT1 and <act1> the mutual-exclusivity relation between **m-hourforms** and **m-intervalforms** could be expressed in BNF using the vertical bar because the two non-terminals play the same role in the same position in the production. Expressed differently, there are no constraints between **m-hourforms** and the other non-terminals that are not identical for **m-intervalforms**. In English, however, there are common mutual-exclusion relations that do not have this property. Subject-verb agreement is an example where the mutual-exclusion relation holds over non-terminals playing different roles. To capture the idea that ''single/plural subject requires single/plural verb'' in BNF, we cannot write

<sentence> ::= (<singlesubj> | <pluralsubj>) (<singleverb> | <pluralverb>).

Instead, we must enumerate the pairings:

<sentence> ::= (<singlesubj> <singleverb>) | (<pluralsubj> <pluralverb>).

Expressing the mutual-exclusion between prenominal and postnominal instances of the same modifying case is another requirement for understanding English. This time the discontiguity of non-terminals playing the same role causes the enumeration. MEETING1 and <meeting1> in Figure 4-7 demonstrate the problem. The definition of MEETING1 says that if the location of a meeting is included, it may be given prenominally (''AISys meeting''), or postnominally (''meeting at AISys''), but not both. The expansion of <meeting1> expresses the same idea by enumerating the possible positions for the modifying case relative to the head noun. If we ignore the question of case order for a moment, then the number of disjuncts required to capture n modifying cases that may occur either prenominally or postnominally is given by:

$$\sum_{k=0}^{n} \binom{n}{k}\binom{n}{n-k}$$

```
ACT1
 strategy (m-hourforms m-intervalforms m-dateforms addwd m-i-ggforms)
 mode (addwd m-i-ggforms)
 unodes ((m-hourforms m-intervalforms m-dateforms))
 mnodes ((m-hourforms m-intervalforms))

<act1> ::=
 [[<m-hourforms> | <m-intervalforms>] [<m-dateforms>]] <addwd> <m-i-ggforms> |
 [[<m-dateforms>] [<m-hourforms> | <m-intervalforms>]] <addwd> <m-i-ggforms>

MEETING1
 strategy (u-locationforms meetingwd m-locationforms)
 mode (meetingwd)
 mnodes ((u-locationforms m-locationforms))

<meeting1> ::=
 [<u-locationforms>] <meetingwd> | <meetingwd> [<m-locationforms>]

FULLMEETING
 strategy (u-hourforms u-intervalforms u-dateforms u-locationforms u-subjectforms
          u-participantforms meetingwd m-hourforms m-intervalforms m-dateforms
          m-locationforms m-subjectforms m-participantforms)
 mode (meetingwd)
 unodes ((u-hourforms u-intervalforms u-dateforms u-locationforms
          u-subjectforms u-participantforms)
         (m-hourforms m-intervalforms m-dateforms m-locationforms
          m-subjectforms m-participantforms))
 mnodes ((u-hourforms m-hourforms u-intervalforms m-intervalforms)
          (u-dateforms m-dateforms) (u-locationforms m-locationforms)
          (u-subjectforms m-subjectforms) (u-participantforms m-participantforms))

<fullmeeting> ::= 3840 terms
```

Figure 4-7: A comparison of CHAMP's representation to BNF notation in capturing the complex ordering and mutual-exclusivity information common in English.

In other words, all *n* modifiers may appear to the left of the head noun, or any one of the modifiers may appear on the left with the remaining *n-1* on the right, and so forth.

The reality, of course, is that we cannot ignore case ordering. It should not be surprising, however, that the size of a grammar in BNF grows quickly when discontiguity and mutual-exclusivity interact with freedom of ordering. FULLMEETING in Figure 4-7 shows the actual definition for a meeting in CHAMP—each of the six modifying cases may occur prenominally or postnominally (but not both) in any order. The general formula for determining the number of BNF disjuncts required under these circumstances is:

$$\sum_{k=0}^{n} \frac{n!}{(n-k)!} \frac{n!}{k!}$$

which is at least *n(n!)* (because *(n-k)! >= 1* for all *k*). As a result, the BNF equivalent of FULLMEETING requires 3840 disjuncts (*n = 5* because the four time modifiers are all mutually-exclusive). The shorthand afforded by the annotation nodes in CHAMP's caseframe representation clearly leads to a more compact grammar.

A formclass represents all the ways of referring to a concept that are recognized by the grammar while each form represents a particular way of referring to the concept through its strategy steps and annotation nodes. The *class* field of a strategy step may assign either a formclass or a wordclass to a subsegment of the utterance. Having discussed formclasses above, we turn our attention now to the wordclass.

4.2.3. The Wordclass and the Lexical Definition

addwd	**ivlmkr**	**ehrmkr**	**intervalwd**
members (add schedule)	members (from)	members (to)	members (lunch)
locations (704)	locations (204)	locations (108)	locations (110)
active t	active t	active t	active t
concept add	concept none	concept none	concept none
add	**from**	**to**	**lunch**
isa addwd	isa sourcemkr	isa ehrmkr	isa mealwd
schedule	isa ivlmkr	isa targetmkr	(isa intervalwd
isa addwd			def (1200 1300))

<addwd> ::= add | schedule
<ivlmkr> ::= from
<ehrmkr> ::= to
<intervalwd> ::= lunch

Figure 4-8: Examples of wordclass and lexical definitions found in the calendar domain displayed in CHAMP's caseframe representation and those portions of the caseframes that can be expressed in standard BNF notation.

As explained in Section 4.2.1, the Formclass Hierarchy is recursively defined through the *isa* fields of forms and formclasses and through the *class* fields of the step definitions. At the bottom of the hierarchy are the *wordclasses*, some of which are shown in the top portion of Figure 4-8. The fields in a wordclass definition are a subset of those in a formclass definition and carry the same general meaning. The *concept* field contains a pointer into application-related knowledge if there are semantic constraints that are signalled by the presence of member of a wordclass in the utterance. If, for example, the word "add" is present in the utterance and only the kernel definition for "add" is in the lexicon, then knowledge in the **add** concept will turn off the *active* fields for

deleteforms, **deletewd**, **changeforms**, **changewd**, and so forth. In other words, if the only way in which "add" can be understood is as a referent to the **add** action, then we may eliminate from the search space all paths that cannot contribute to that interpretation. The *members* of a wordclass are the actual words and phrases in the lexicon. As the lower portion of Figure 4-8 demonstrates, wordclasses in CHAMP correspond to non-terminals that expand to only terminals in a BNF representation.

The middle portion of Figure 4-8 shows that lexical definitions provide entry into the Formclass Hierarchy through the wordclass in the *isa* field. CHAMP allows both word and phrase entries in the lexicon. The system also permits multiple definitions for each entry (see, for example, "from"). In addition to the *isa* field, an entry may contain a *def* field that associates a canonical value with the usage of a word. When "lunch" is used as an **intervalwd**, for example, the *def* of "lunch" is **(1200 1300)**—the canonical value for the interval noon to 1 p.m. Instance generators, which have the task of producing canonical representations that aid in constraint application and database search, also have the responsibility of recovering the value in a *def* field. Figure 4-9 demonstrates how the mechanism works. In the example, the second definition of "lunch," which reflects usage of the word as a modifier indicating time, leads to a successful parse. INT2 seeks the wordclass for this definition (**intervalwd**) in its sole step (110). Thus the presence of the word "lunch" is adequate to satisfy INT2 and invoke the instance generator for the class **u-intervalforms**. The special token **instance** in the *bindvar* field of a step directs an ig to recover the canonical value from the lexical definition rather than to compute it. In this way the canonical form of the default lunch interval, **(1200 1300)**, is bound to **ivl** and **divl** in step 206.

4.3. Application-Related Knowledge: The Concept Hierarchy

By basing forms and formclasses on semantic categories, CHAMP's learning-related structures contain some language-independent knowledge about the domain. A formclass ties together different ways of expressing the same concept. In turn, the fixed number of formclasses limits the user to a fixed number of meaningful actions and objects; we organize these domain concepts into a Concept Hierarchy. It is not surprising that the structure of the Concept Hierarchy largely parallels that of the Formclass Hierarchy—the former organizes information about meaningful referents, the latter organizes information about meaningful references. The two hierarchies can be thought of as densely connected within but sparsely connected between. We have already seen that connections between the hierarchies are limited to the wordclass's and formclass's *concept* and *active* fields and the strategy step's *bindvar* field.

If all we knew about domain concepts were their names and categories, it would be unnecessary to build a separate structure to organize them. Fortunately, we can say more

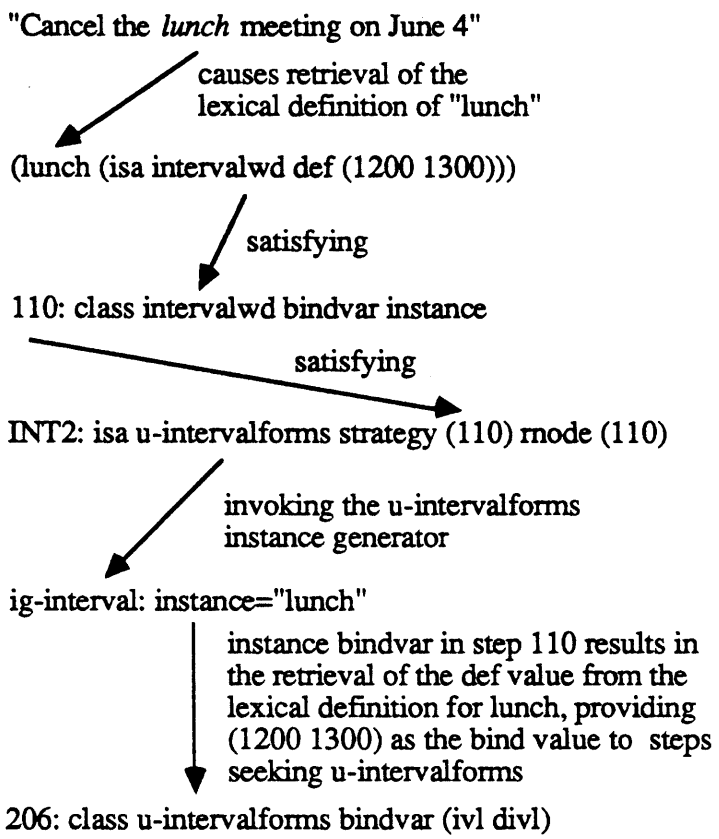

Figure 4-9: Providing canonical values through the interaction of **instance** *bindvars* and lexical *def* fields.

than just what constitutes a meaningful referent. Specifically, what we mean by a "domain concept" is precisely that collection of invariant information that defines an action or object within an application. In this section we will examine how the information associated with concepts and *conceptclasses* can help to constrain search and facilitate user interaction. Figure 4-10 displays both kinds of knowledge for a portion of the calendar Concept Hierarchy. Like the representation of the grammar, the representation of domain concepts has been simplified slightly in our examples for expositional purposes—the exact representation can be seen in Appendix A. In CHAMP, a Concept Hierarchy must be hand-coded in terms of the primitive fields (*isa*, *seg*, *bind*, *exp*, *rec*, *default*, and *extend*) separately for each domain. To the extent that different domains share concepts, the application knowledge coded for one may be used in the other. CHAMP shares action concepts, time concepts, and some location concepts between its two domains. As in the previous section, most of our examples will be taken from the calendar domain.

Search control is aided by the four types of constraint information found in the fields labelled *seg*, *bind*, *exp*, and *rec*. CHAMP takes an integrated approach to controlling

```
add
isa action
seg ((pointers addwd gowd)
     (opponents deletewd showwd...)
     (turnoff deleteforms showforms...))
rec ((no-del obj))

meal
isa object
default ((location (* *)))

lunch
isa meal
default ((starthr 1200))

subject
isa root
rec ((no-del subject)
     (no-trans (marker-subject)))
extend (subject subjname)

interval
isa root
exp ((deref starthour endhour)
      (lambda (ds de)
       (or (< (hour ds) (hour de))...))
rec ((no-del interval starthour endhour)
     (no-trans (marker interval)
               (starthour endhour)
               (emarker endhour)))

hr
isa root
bind (<= 1 !value 12)

minutes
isa root
bind (<= 0 !value 59)
```

Figure 4-10: Examples of application-related knowledge from the calendar domain (an "*" value is a wildcard).

search [54], applying each type of constraint as early and as often in the understanding process as possible. Thus, the four kinds of constraint information correspond to the four points in processing at which constraining knowledge can be brought to bear.[18]

At *seg*mentation-time we may use the presence of a word or phrase that is unique to a wordclass or domain to eliminate large portions of the search space. Thus, if the utterance contains the word "add" and our grammar contains no definition for "add" other than the kernel definition (*isa* **addwd**), we know that we may ignore those portions of the grammar that are incompatible with an **add** action. We eliminate the incompatible paths in the search space by turning off the *active* fields of the wordclasses found in the *opponents* subfield and the formclasses in the *turnoff* subfield. The uses of the *seg* and *active* fields are discussed further in Section 5.1.

[18]Actually, there are five points in processing at which we apply constraining knowledge. We have postponed any discussion of the constraints imposed by the database until Chapter 7, when we examine the resolution process.

At *bind*-time we apply predicate constraints to the values we want associated with a *bindvar*. In Figure 4-10, for example, we see that a strategy step whose *bindvar* list contains the token **hr** can be satisfied only by a number between one and twelve. Bind-time constraints are also discussed further in Section 5.1.

*Exp*and-time constraints are applied during the Expand phase of the Coalesce/Expand Cycle (Section 5.2). At that time we consider the limiting effects of intercase constraints. In Figure 4-10 the concept **interval** carries a constraint of this type on the cases **starthour** and **endhour**. Each case is recognized by a step seeking a **u-hourform** (see the definition of INT1 in Figure 4-5). The expand-time predicate accesses the canonical values bound to **starthour** and **endhour** and prevents further search down any path in which the **starthour** is later than the **endhour**. As an example, consider the sentence

"Change the meeting from 5 p.m. to 3 p.m."

In this case, the interval constraint will prevent the segment "5 p.m. to 3 p.m." from being parsed as an interval (and, consequently, prevent "from 5 p.m. to 3 p.m. from being parsed as a marked interval). The concept definition for a source-target pair, on the other hand, demands only that the classes of the source and target be compatible. Thus, our sample sentence will be correctly understood as a command to change the starting time of a particular meeting.

*Rec*overy-time constraints occur, naturally enough, during error recovery (Chapter 6). These constraints block error corrections otherwise permitted by the model. Let us consider the recovery constraints associated with the conceptclass **subject**. A **subject** is referred to by any word or phrase that acts as the specific topic of a meeting or seminar (for example, the phrase "Non-monotonic Logics" in "Schedule an AI seminar about Non-monotonic Logics"). Figure 4-11 summarizes what CHAMP knows about the concept. As shown, SUBJ0, a member of **m-subjectforms**, is made up of a marker and a member of **u-subjectforms**. The recovery constraint in the concept definition **(no-del subject)** prevents the recovery mechanism from deleting step 214 whose *bindvar* list contains the token **subject**. In other words, the constraint prevents us from considering as a reference to a **subject** any segment that does not actually contain a subject. Similarly, **(no-trans (marker subject))** prevents recovery from transposing steps 213 and 214; what makes a marker a marker is that it introduces what it marks. Although they do not apply to **subject**, **no-sub** constraints may also be specified. The concept **date**, for example, contains the recovery constraint: **(no-sub day year)**. This entry prevents substitution of non-numbers in links with the *bindvar* symbol **day** or **year**. The fourth type of error recovery, insertion, has no corresponding constraint—insertions are always permitted.

Since recovery constraints are clearly tied to the form of a reference, it is reasonable to ask why they are contained in the Concept Hierarchy. In part, the reason is that these

From Kernel Grammar

m-subjectforms
isa root
members (SUBJ0)
locations (318)
agenda-level 2
active t
concept subject

SUBJ0
isa m-subjectforms
mode (213 214)
strategy
213: class sbjmkr bindvar marker
214: class u-subjectforms bindvar subject

From Concept Hierarchy

subject
rec ((no-del subject)
(no-trans (marker subject)))
extend (subject subjname)

Figure 4-11: A portion of CHAMP's knowledge about subjects.

constraints help insure that a new form of reference is meaningful. More important, however, is that recovery constraints are like other pieces of application-related knowledge—they are part of the fixed assumptions about the domain. Since they are not truly language-independent, recovery constraints may affect the language the system can learn. Why, then, do we add them? In general we include a constraint because we believe it will eliminate unnecessary search. Recovery constraints generally eliminate search down paths that correspond to nonsensical explanations of deviant utterances. By adding the recovery notation **(no-del subject)**, we choose not to entertain the notion that the existence of a preposition that may introduce a subject is, by itself, adequate to hypothesize that the subject has been deleted. Similarly, **(no-trans (marker subject))** means we will not accept that the word following a recognized subject may be a legitimate substitution for a missing marker (because we expect markers to appear before what they mark). Although it cannot be guaranteed that the search eliminated by these constraints is unnecessary, when used carefully the possibility of eliminating the user's intended meaning is slight, and the savings in search increases at each deviation-level.

In addition to constraining search, information organized in the Concept Hierarchy may facilitate user interaction. Knowledge of this type is located in the *default* and *extend* fields. If explicit default values are provided for a concept, CHAMP may use them to compensate for user omissions. Consider the sentence

"Schedule lunch with John on June 4."

which indicates neither where the participants will eat nor at what time. Under these

circumstances, CHAMP will use the reference to lunch to provide a default time and location. The former value is given explicitly in the definition of the concept **lunch**, while the latter is inherited from **meal** through lunch's *isa* field (see Figure 4-10).[19] In general, default reasoning by the system prevents interactions perceived as unnecessary by the user. We will have a great deal more to say about default reasoning and inference when we discuss resolution in Chapter 7.

The *extend* field supplies class extendability information. The how's and why's of learning new instances of known classes are discussed in full in Section 5.1.3. Here we recall only that the conceptclasses that are to be considered extendable are fixed. Figure 4-10 shows that **subject** is one such class. As a result, when CHAMP encounters an unknown segment in a context that permits a **subjectform** constituent, it will, with the user's help, try to resolve the segment as a new subject. In this way, the system learns about new topics, people, and places as the user's experience dictates. The price paid for relaxing the Fixed Domain Assumption in this limited sense is additional (but focused) user interactions. The benefits include reduced search, fewer future interactions, and the ability to learn even when presented with an ultimately unparsable sentence.

One of the main challenges in building an adaptive interface is designing knowledge structures that are (1) general enough to support grammar growth in unanticipated directions, and (2) specific enough to constrain search to reasonable limits. The decomposition of a grammar based on semantic categories into the components of a Formclass Hierarchy gives CHAMP the flexibility to capture the user's idiosyncratic language by adaptation at the appropriate level of representation. The combination of formclasses, forms, wordclasses, and lexical entries is robust enough to learn new sentential, phrasal or lexical references, with the latter including both synonyms and new instances of extendable classes. At the same time, shared strategy steps and the invariant constraints, defaults and extendability information in the Concept Hierarchy help control the search for an effective meaning. Thus the knowledge structures discussed in this chapter appear to meet the design challenge imposed by (1) and (2), above. In the next chapter we will begin our examination of how these knowledge structures are used in the understanding and adaptation processes.

---

[19]The inherited location (* *) is a way of saying that the location of a meal may legitimately remain unspecified.

Chapter 5
Understanding Non-Deviant Utterances

The knowledge representations introduced in the previous chapter guide the parser's search for a meaning structure. In CHAMP, the meaning of an utterance is represented by an *annotated parse tree* (APT). A grammatical sentence—one with no deviations—produces an APT with no recovery notations (or more than one unannotated tree if the sentence is ambiguous). We call such a tree an *explanation* of the utterance. Note that explanations are produced at Deviation-level 0 by definition. If the sentence requires error recovery, an APT with notations is produced at Deviation-level 1 or higher; we call an APT of this type a hypothetical explanation, or *hypothesis*. This chapter discusses the bottom-up parsing process by which APTs and their constituent parse nodes are constructed (for an example of an APT see Figure 5-25 at the end of Section 5.2). To simplify the exposition, we consider here only Deviation-level 0 search, creating explanations. In the next chapter we extend our discussion to creating hypotheses.

Figure 5-1 presents an overview of the parsing process for non-deviant utterances; the figure extracts from Figure 4-1 those aspects of CHAMP required for parsing without error recovery. The first step in parsing is the segmentation of the utterance into the words and phrases found in the system's lexicon. As shown in the figure, the resolution of unknown segments as new instances of extendable classes is also accomplished at this time.

When segmentation has been completed, CHAMP has access to the leaf nodes in the parse tree for the utterance. CHAMP constructs the remainder of the APT in a bottom-up manner, building larger and larger subtrees (constituents) via the interactions of the Coalesce/Expand Cycle (shown in Figure 5-1) and a data structure called the *Agenda* (not shown in the figure). The purpose of the Agenda is to impose on the search process the partial ordering of constituents given by the Formclass Hierarchy. Each level of the Agenda corresponds to a level in the Hierarchy. Thus, the number of Agenda-levels is domain-dependent rather than implementation-dependent, reflecting the degree of embedding inherent in a particular grammar. CHAMP's basic parsing algorithm progresses bottom-up, level-by-level through the Agenda coalescing previously constructed subtrees into new constituents, then expanding the new constituents to higher

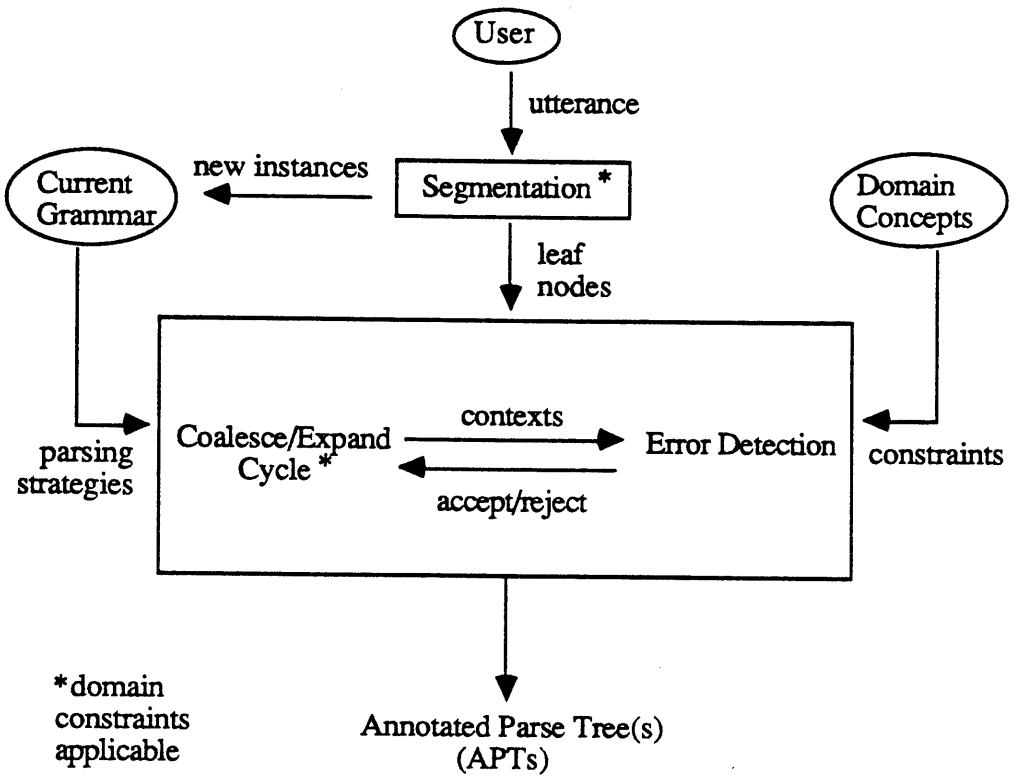

Figure 5-1: The components for parsing at Deviation-level 0 extracted from Figure 4-1.

levels of the Agenda.[20]

If we consider only non-deviant utterances, parsing in CHAMP is analogous to bottom-up parsing in a context-free grammar. Strategy lists perform the same role as productions. A form's strategy list dictates which constituents may coalesce at that form's *agenda-level* in the same way that the right-hand side of a production dictates which of the existing terminals and non-terminals should be matched. Expanding the coalesced set of constituents to a higher Agenda-level is analogous to replacing the set of matched terminals and non-terminals with the non-terminal on the left-hand side of the production. Figure 5-1 indicates (by asterisk) one significant difference between the two parsing methods, however. During both segmentation and the Coalesce/Expand Cycle, CHAMP may apply semantic search constraints that are external to the grammar.

In the remainder of this chapter we examine in detail how each phase of parsing is implemented in CHAMP by following a non-deviant utterance

"Cancel the 3 p.m. speech research meeting on June 16"

[20]The process is similar in many ways to semantic chart parsing [17].

through the parsing process. The output from CHAMP used in the examples below reflects the work done at Deviation-level 0 with both the calendar and travel domains loaded. Where appropriate, the descriptions of CHAMP's algorithms have been simplified to specify only Deviation-level 0 behavior; the full algorithms are given in Chapter 6.

5.1. Segmentation

The purpose of segmentation is to create the leaf nodes in an APT. The process is divided into five parts: converting the input string to tokens, partitioning the tokens into meaningful segments, trying to resolve unknown segments as instances of extendable classes, applying segmentation-time constraints, and seeding the Agenda. CHAMP's algorithm for performing segmentation is introduced in Figure 5-2 and explained as we examine each subprocess in turn.

5.1.1. Creating Tokens

Since CHAMP expects a string as input, the first step in the segmentation process converts the input string to tokens recognizable in the grammar. Three kinds of changes to the utterance occur during this stage: capitalization is removed (''John Smith'' creates the tokens (john smith)), spaces are inserted around punctuation marks and numbers (''June 14th'' becomes (june 14 th)), and punctuation marks are converted to an internal form (''3:00-4:00'' becomes (3 %colon 0 %hyphen 4 %colon 0)). The conversion of our sample input produces the tokens: (cancel the 3 p %period m %period speech research meeting on june 16).

5.1.2. Partitioning Tokens into Segments

Once the string has been converted to tokens, the tokens must be partitioned into the discrete words and phrases in the lexicon. CHAMP does this by creating *pnodes*. A *pnode* assigns a class to a subsegment of the utterance using three values: the starting and ending positions of the phrase in the input and the name of a class. In our sample sentence, for example, the token **cancel** causes the creation of the pnode: (0 0 deletewd).

During the experiments described in Chapter 3, if any segmentation of the input could be found that made the utterance parsable, that segmentation was chosen. The need to implement segmentation in CHAMP, however, revealed the computational complexity of achieving this ideal behavior. There were essentially three issues to be addressed:

1. If both a phrase and a word within the phrase have definitions, do we explore both? In other words, do we explore only the maximal defined subsegment (for example, ''University of Chicago''), or all defined subsegments (''University,'' ''of'', and ''Chicago'') as well?

SEGMENT (string)
Convert the string to tokens (1)
FOR each token in the utterance, DO
  IF the token is a number
  THEN create a pnode with class **number** (2)
  IF a token has no definition (3)
  THEN apply spelling correction
    IF the user accepts a spelling correction
    THEN replace the token with the correct word
    ELSE create a pnode with class **unknown**
  FOR EACH lexical definition of the token as part of a phrase, DO (4)
    IF the whole phrase is present and maximal in length
    THEN create a pnode for the definition's class spanning the whole phrase (4a)
    ELSE IF some word in the phrase is present in the utterance and there is no definition for that word by itself (4b)
      THEN create a pnode for **partof** the class spanning the subphrase
    IF the token was not captured by a completed phrase (5)
    THEN FOR EACH definition of the token that is not a phrase definition, DO
      create a pnode for the token and definition's class
IF contiguous pnodes have class **unknown** or are **partof** different phrases (6)
THEN combine the pnodes into one of class **unknown**
IF a pnode represents **partof** a phrase that is NOT in an extendable class (7)
THEN make the pnode **unknown**
ELSE try resolve it via interaction as an abbreviation of the phrase and
    IF it can't be resolved as an abbreviation
    THEN make it **unknown**
IF a pnode is **unknown** (8)
THEN try to resolve it as a new instance of an extendable class
Apply segmentation-time constraints (9)
FOR EACH pnode with an active wordclass, DO (10)
  FOR EACH active step that seeks the pnode's class, DO
    IF the step has a bind-time constraint AND the pnode satisfies it
    THEN create a pc for that step and pnode and place it on the Agenda

Figure 5-2: The behavior of CHAMP's SEGMENT Algorithm at Deviation-level 0.

2. If an unknown segment spans more than one token, do we assume a single grammatical function for the phrase, or do we explore multiple functions? The problem is demonstrated by contrasting the sentence "Schedule a seminar with John speaking about AI" when neither **John** nor **speaking** are known and "Schedule a seminar with John Smith about AI" when neither

John nor **Smith** are known. In the latter case the unknowns form a single functional unit while in the former case they do not.

3. How do we treat the appearance in the utterance of portions of a phrase in the lexicon: as unknown segments, or as indications of abbreviation? If we know "Columbia University" is a school, do we assume "Columbia" refers to the same place? Do we make the same assumption for "University"? Are the rules different if we know "show me" but the utterance contains only "me"?

CHAMP's segmentation algorithm resolves these questions in an interrelated fashion. Complete phrases of maximal length are preserved over any shorter, embedded complete phrases (step (4a) of Figure 5-2), over incomplete phrases (step (4b)), and over the definitions of individual words within a phrase (step (5)). Thus, even if each of "university of chicago," "university," "of," and "chicago," appears in the lexicon, only "university of chicago" will be assigned a pnode during segmentation.

Addressing the question raised by multi-word unknown segments, the algorithm incorporates:

- **The Single Segment Assumption**: contiguous unknown tokens are considered to perform a single function in the utterance.

This assumption avoids the combinatoric cost of exploring multiple functions within a segment at the price of rendering sentences that violate the assumption unparsable.[21] Thus, SEGMENT behaves correctly when given "Schedule a seminar with John Smith about AI" by considering "John Smith" as a single segment; the algorithm behaves incorrectly, however, when given "Schedule a seminar with John speaking about AI" by considering "John speaking" as a single segment as well.

The third question concerned the status of incomplete phrases. The SEGMENT algorithm says that if only a portion of a phrase is present in the utterance, its status depends upon what else is in the sentence and what is in the lexicon. If all of the words within the incomplete phrase can be defined independently of the phrasal definition, the phrasal definition is ignored (steps (4b) and (5)). Even if a subsegment of the utterance can be explained only as a partial phrase, the subsegment may still lose its assignment to

---

[21]Each of the learning programs discussed in Section 2.5 (prior research) makes the same assumption in a more restrictive way by guaranteeing *a priori* that the input contains exactly one unknown segment performing exactly one function. That guarantee enables the systems' designers to avoid most of the questions we have posed as integral to the complex issue of segmentation. In designing CHAMP to handle real utterances that contain multiple unknowns requiring different kinds of resolution, we were forced to face the issue directly. Our solution was the algorithm presented and the Single Segment Assumption. Unfortunately, the evaluation of our experimental data in Chapter 9 clearly demonstrates that the assumption is sometimes violated in real user input. Equally unfortunate, however, is the performance degradation that would accompany the relaxation of the assumption. We postpone further discussion to Chapter 11.

the phrase's class if it is surrounded by unknown segments or other incomplete phrases (step 6). This merging of contiguous unknowns and partial phrases enforces the Single Segment Assumption.

The final status of a subsegment that remains identified as a partial phrase after step (6) depends upon whether or not the class associated with that phrase is extendable. If the class is extendable the chance to establish an abbreviation is offered to the user (for an example, see Figure 5-5, page 71)). If the abbreviation is refused, or if the phrase's class is not extendable, the pnode is assigned to the class **unknown**. Thus, either of "Columbia" or "University" would signal a possible abbreviation (because **schoolname** is an extendable class) but "me" would be considered unknown (because **showwd** is not extendable). In the next section we examine how unknown segments are treated as candidate new instances of extendable classes (not as candidate abbreviations for a known phrase).

Figure 5-3 shows the effect of the first five steps of SEGMENT on the conversion of the tokens in our sample sentence into pnodes. Each token leads the system to kernel lexical definitions which, in turn, dictate the partitioning of tokens into segments. The pnode marks the bounds of the segment within the utterance and assigns to the segment the grammatical role of the definition that created it. Notice the different ways in which **the** and **speech** are treated. Both tokens have phrasal and non-phrasal definitions. In the sample sentence, however, the phrasal definition for **speech** can be completed while the phrasal definitions for **the** cannot. Thus, step (4) identifies **speech research** as the name of a subject (by looking up the completed phrase in the lexicon) and the alternative meaning of **speech** (as a projectname) is ignored in step (5). In contrast, **the** falls through step (4) because its phrases are incomplete, allowing step (5) to create the pnode for a definite article. As a final note to Figure 5-3 observe that it is possible to produce multiple interpretations for a segment; in our example, **on** creates two pnodes because it may mark two semantically distinct kinds of dates.

5.1.3. Resolving Unknown Segments as New Instances of Extendable Classes

Since our sample sentence does not contain unknown segments or incomplete phrases, it is unaffected by steps (6), (7), and (8) of the algorithm in Figure 5-2. Let us suppose, however, that the words "speech" and "research" have no definition in the kernel lexicon. A strict interpretation of the Fixed Domain Assumption (Section 2.3) requires that an unknown segment correspond to a new way of referring, not to a new referent. While the Fixed Domain Assumption is a powerful mechanism for controlling search, it seems, at the same time, to be an unreasonable limitation for an adaptive interface. Following Kaplan [29], we would prefer a system in which the types of objects that could be referred to could not change, but the particular instances of those types could. To

Tokens	Kernel Lexical Definitions	Segments	Pnodes
cancel	deletewd	cancel	(0 0 deletewd)
the	(part1 (the airport))		
	(part1 (the office))		
	defmkr	the	(1 1 defmkr)
3	<no entries>	3	(2 2 number)
p	(1 (p %period m %period))	(p %period	(3 6 nightwd)
%period	(2 (p %period m %period))	m %period)	
	(4 (p %period m %period))		
	(2 (a %period m %period))		
	(4 (a %period m %period))		
	eosmkr		
m	(3 (p %period m %period))		
	(3 (a %period m %period))		
%period	(2 (p %period m %period))		
	(4 (p %period m %period))		
	(2 (a %period m %period))		
	(4 (a %period m %period))		
	eosmkr		
speech	(1 (speech research))	speech research	(7 8 subjname)
	projectname		
research	(2 (speech research))		
meeting	meetingwd	meeting	(9 9 meetingwd)
on	(2 (arriving on))		
	(2 (departing on))		
	datemkr	on	(10 10 datemkr)
	departuredatemkr	on	(10 10 departuredatemkr)
june	monthwd	june	(11 11 monthwd)
16	<no entries>	16	(12 12 number)

Figure 5-3: The pnodes created during segmentation of the tokens from "Cancel the 3 p.m. speech research meeting on June 16."

accomplish this limited relaxation of the Fixed Domain Assumption in CHAMP, we designate certain conceptclasses to be extendable (see Sections 2.3 and 4.3). In particular, the conceptclasses for participants, locations, subjects, general topics, hours, time intervals, and dates are extendable.[22] Thus, "speech research," if unknown, would be a candidate new instance for these classes.

Assuming we wish the introduction of new instances to be possible, why not treat the

[22]The inclusion of hours, time intervals, and dates in the list may seem odd since the number of hours in a day and days in a year are fixed. We make hours extendable so that words can come to designate particular hours (for example, "tea time" or "lunch time"). Similarly, the user may wish to nominalize particular intervals ("happy hour"). We make dates extendable in order to be able to designate particular days (for example, John's birthday).

occurrence as a new type of deviation? The justification for distinguishing between deviations and new instances is simple: we expect the former to become increasingly infrequent but not the latter. The model predicts (and our experiments confirm) that as the adapted grammar comes to more closely reflect the user's actual grammar, the number of deviant utterances approaches zero. On the other hand, we cannot assume the same monotonic decrease in the number of new people met, places traveled or topics discussed. Over time, then, we expect that unknown phrases will, with increasing likelihood, correspond to new instances rather than new ways of referring. As a consequence, we want to explore that correspondence early in the understanding process—before embarking on the comparatively lengthy search for a deviant explanation. Thus, the reason for detecting new instances at Deviation-level 0 is to guarantee a more responsive interface in the long run.[23]

There are three reasons why we try to resolve unknown phrases specifically at segmentation-time rather than during parsing at Deviation-level 0. First, assigning a segment to a particular wordclass enables us to eliminate search during the parse whenever the class triggers the application of segmentation-time constraints (as discussed in the next section). Since most of the extendable categories in CHAMP index such constraints, it is in our best interest to identify them *before* parsing begins. The second and third reasons for resolving unknowns during segmentation are interrelated; early resolution enables us to detect one kind of unparsable sentence quickly and to increase the system's knowledge despite the fact that the sentence is unparsable. Observe that if, after segmentation, the number of still unresolved segments is greater than the maximum number of deviations permitted by the system (two in CHAMP), the parse must ultimately fail. Thus, the user can be asked to rephrase her request immediately rather than after a potentially lengthy search that cannot succeed. At the same time, the new names and places learned during segmentation are available without further interaction when the rephrased request is entered.

CHAMP's method for detecting new instances requires user interaction. In general there is a trade-off between user effort and user satisfaction—the more the system requires of the user the less useful the system is perceived to be. This is especially true if the required interaction appears unnecessary or confusing to the user. Since confusion can come from misinterpreting a user's response, CHAMP maintains tight control over the interaction by asking only yes/no or multiple choice questions. The course of the interaction is dictated by the knowledge in the *extend* fields of the conceptclasses (see Section 4.3) and by the user's responses. Figure 5-4 shows a typical interaction that might result from our sample sentence given the assumption that both "speech" and

---

[23]The same reasoning explains why spelling correction is implemented during segmentation rather than as part of deviation detection and recovery (see step (3) of the SEGMENT algorithm).

"research" are unknown. Note that since the tokens are both contiguous and unknown they are considered as a single segment.

```
next> cancel the 3 p.m. speech research meeting on June 16

Trying to understand (SPEECH RESEARCH). Does it refer to a/an:
(0) DATE, (1) TIME-INTERVAL, (2) SINGLE-TIME, (3) PARTICIPANT,
(4) LOCATION, (5) SUBJECT, (6) GENERAL-TOPIC,
(7) none of these? [give number]: 5
Ok, adding new definition.
```

Figure 5-4: Resolving "speech research" as a new instance of **subjname**.

When the pnode created for "speech research" has its class defined as **unknown**, CHAMP searches the Concept Hierarchy for extendable classes in the current domain. Part of the information found in the *extend* field for a concept is the token to present to the user (for example, **time-interval**). The remaining information tells the system how to behave if that token is chosen. In the case of the concept **subject**, the *extend* field contains only **(subject subjname)**. The first value is the token shown to the user, the second value is the wordclass placed in the *isa* field of the lexical definition created in response to her choice. In this way, the choice of SUBJECT results in the addition of a definition of "speech research" as a member of **subjname** to the lexicon.

The interaction proceeds slightly differently if the pnode has been identified by SEGMENT as **partof** a phrase. Let us assume that "speech research" has been successfully added to the dictionary as a result of the interaction in Figure 5-4. Now suppose that the user asks CHAMP to: "Schedule a speech meeting on June 20 at noon." Although "speech" by itself has no definition in the lexicon, it is present as part of the defined phrase "speech research." The interaction in Figure 5-5 shows the kind of extremely focused question posed by the system when abbreviation is correctly suspected.

```
next> Schedule a speech meeting on June 20 at noon.

Is SPEECH short for (SPEECH RESEARCH)? [y or n] y
Ok, adding new abbreviation.
```

Figure 5-5: Introducing "speech" as an abbreviation for "speech research."

In contrast, the interaction in Figure 5-6 shows how the system's behavior changes when an abbreviation is incorrectly suspected—the user's rejection causes the segment to be treated as an unknown. As an unknown, the word "speech" must be offered to the

user as a candidate instance of the full set of extendable categories. In the figure, the user chooses to assign the word to the category **participant**. Unlike the simple value in the *extend* field of the **subject** concept, the value in the *extend* field of the **participants** concept contains a tree structure: **(participant ((student studentname) (professor profname)...(project projectname))**. This hierarchical arrangement of increasingly specific categories may continue to an arbitrary depth and translates into a focused clarificational dialogue during interaction. In Figure 5-6 the dialogue ends after a single clarification, when the user identifies "speech" as the name of a project. The example demonstrates how the additional definition of "speech" as a **projectname** in Figure 5-3 could be introduced into the lexicon.

```
next> Schedule a speech meeting on June 20 at noon.

Is SPEECH short for (SPEECH RESEARCH)? [y or n] n

Trying to understand SPEECH. Does it refer to a/an:
(0) DATE, (1) TIME-INTERVAL, (2) SINGLE-TIME, (3) PARTICIPANT,
(4) LOCATION, (5) SUBJECT, (6) GENERAL-TOPIC,
(7) none of these? [give number]: 3

Is SPEECH the name of a:
(0) STUDENT, (1) PROFESSOR, (2) FAMILY-MEMBER, (3) GROUP,
(4) BUSINESS, (5) PROJECT,
(6) none of these? [give number]: 5
Ok, adding new definition.
```

Figure 5-6: Introducing "speech" as a member of the wordclass **projectname**.

5.1.4. Applying Segmentation-time Constraints

Once we have assigned segments to wordclasses we may have access to a great deal of information constraining the possible meanings of the utterance. The relevant information is available through the *seg* fields of the concepts pointed to by the wordclasses. As shown in Figure 5-7, our sample sentence contains three segments that provide the constraints called for in step (9) of the SEGMENT algorithm.

The algorithm that applies segmentation-time constraints is straightforward. Any pnode with a class that appears in the *pointers* list of a concept's *seg* field acts as evidence for that concept. Evidence is considered conclusive if all the pnodes covering that segment act as pointers to consistent concepts and no pnode exists uniquely covering some other segment and having a class in the concept's *opponents* field. In other words, evidence is conclusive if every explanation of some segment refers to a set of consistent concepts and no other segment can be explained only by an inconsistent concept. If a concept is voted

Tokens	Wordclass	Concept
cancel	deletewd	delete
the	indefmkr	
3	number	
p %period m %period	nightwd	
speech research	subjname	groupgathering
meeting	meetingwd	groupgathering
on	datemkr	
	departuredatemkr	
june	monthwd	
16	number	

Concept Definitions:

delete
isa action
seg (pointers deletewd)
(opponents addwd gowd changewd indefmkr calendarwd...)
(turnoff addforms taddforms showforms tshowforms m-calsegforms...)

groupgathering
isa object
seg (pointers meetingwd classwd seminarwd mealwd buildingname subjname...)
(opponents tripwd flightwd poundsymb departuredatemkr...)
(turnoff mrtripforms tripforms utripforms m-dephrforms...)

Figure 5-7: Segmentation-time constraints provided by segments in "Cancel the 3 p.m. speech research meeting on June 16."

for conclusively the wordclasses in its *opponents* field and the formclasses in its *turnoff* field have their *active* fields set to false—if we know that some segment must be explained as a reference to a particular concept then we need never generate interpretations inconsistent with that reference. We shall see in the next section (and Section 5.2) that only active classes may contribute constituents to an annotated parse tree.

Let us consider the effect of segmentation-time constraints on our example. Since the token **cancel** is a **deletewd** it indexes the segmentation-time constraints for the concept **delete**. As there is no alternative referent for **cancel** and no opposing reference among the other pnodes, all the wordclasses and formclasses that are inconsistent with a delete action are made inactive for the duration of the parse. The classes turned off are those unique to the other actions recognized by the system: add, change, and show.

Both **speech research** and **meeting** index the **groupgathering** concept through **subjname** and **meetingwd**, respectively. Neither segment has an alternative definition but there does appear to be an opponent pnode: one definition for **on** is as a **departuredatemkr**, a wordclass in **groupgathering**'s *opponents* field. The evidence for **groupgathering** is considered conclusive, however, because **on** has an alternative definition (as a **datemkr**) that is not inconsistent with the **groupgathering** concept. Thus, the wordclasses and formclasses pertaining to flights and trips are also made inactive.

Note that the application of segmentation-time constraints depends on both the values in the *pointers*, *opponents*, and *turnoff* fields (which are fixed) and the lexical definitions indexed by the tokens in an utterance (which may vary as the lexicon grows). As a result, a word may provide constraint at one point in the evolution of a grammar but lose that constraining property if it gains a definition that points to an opposing concept. If enough lexical ambiguity enters the grammar it is possible for an utterance to introduce no constraints from the Concept Hierarchy during segmentation.

5.1.5. Seeding the Agenda

The final step of CHAMP's segmentation algorithm places the possible leaf constituents for an APT onto the Agenda. The pnodes we have constructed contain only some of the information required. To fully specify a constituent parse node we must turn each pnode that contains an active class into one or more *pclones* (*pcs*, for short). A *pc* is the representation of a grammatical constituent within an annotated parse tree. A *pc* is specified by the values in five fields:

Pc Field	Description
step	associates a segment of the utterance interpreted as a member of a grammatical class with a particular strategy step that seeks that class.
pnode	assigns a segment to a grammatical class. The class in the pnode is always the same as the class sought by the *step*. More than one pc may share the same pnode (thus, the "clone" in "pclone").
subs	pcs representing the roots of the subtrees of the current pc. Equivalently, the pcs representing embedded constituents. Leaf nodes have no *subs*.
strats	the names of the forms with *strategy* lists that contain all the steps of the pcs in the *subs* field. In other words, the forms by which the subconstituents may coalesce to create a member of the class. Leaf nodes also have no *strats*.
dlevel	records notations during error recovery and keeps track of the amount of deviance in the subtree rooted at the current pc.

Figure 5-8 shows a constituent from the APT for our example sentence (the full APT

appears at the end of the chapter). The figure introduces the notation used to display pcs: the *step* field is given first, followed by the *pnode*, *strats*, *subs*, and *dlevel* fields. In future figures, we suppress fields with nil or zero values.

pc25: 316 (10 12 m-dateforms) (DATE0) (8 22) 0

pc field	value	comments
step	316	captures marked date as part of a **groupgathering**
pnode	(10 12 m-dateforms)	assigns "on June 16" to **m-dateforms**
subs	(8 22)	pc8 is the **datemkr**, pc22 is the **u-dateform**
strats	(DATE0)	contains the steps in pc8 and pc22 in its strategy list
dlevel	0	"on June 16" is a non-deviant marked date

Figure 5-8: The pc representing "on June 16" as a marked date in the APT for "Cancel the 3 p.m. speech research meeting on June 16."

Figure 5-9 demonstrates how the steps contained in a wordclass's *locations* field are used to convert a pnode to one or more pcs. The figure illustrates step (10) of the SEGMENT algorithm applied to the token **16** in our example sentence. The pnode for the token **16** contains the class **number** which, in turn, can be found at eight different step locations in the grammar.

Figure 5-9 shows that not every pc that can be created for a pnode is placed on the Agenda; both segmentation-time and bind-time constraints can affect the process of conversion. As an example of the former, consider step 353 which seeks a member of the class **number** to act as a flight number. Recall that the presence of the token **meeting** previously turned off all references to flights. In particular, the formclass **flightforms** was made inactive. Since step 353 occurs only in FL2's strategy list and FL2 is a member of the inactive formclass **flightforms**, step 353 is also inactive and no pc is created for its meaning of **number**.

As an example of bind-time constraints, consider step 4 which seeks a number to act as an hour designator. Since **16** does not meet the requirement of falling between one and twelve, no pc is created for this interpretation either. Thus, although a number may play eight distinct roles in the grammar, context-sensitive constraints eliminate all but three interpretations for the token **16** in our sample sentence: **16** as a day (step 102), as a class-level (step 12), and as minutes (step 6). To place these interpretations on the Agenda, we trace back through the step's forms to find their formclass' *agenda-level* fields (a step may be shared by forms in more than one formclass but every formclass sharing a step must have the same *agenda-level*). As an example, pc*x* contains step 6 which is found in form HR2, a **u-hourforms**; the *agenda-level* field for **u-hourforms** places pc*x* at Agenda-level 0.

Pnode: (12 12 number)
Token at position (12 12) in the utterance: **16**
Step locations for the wordclass **number**: (4 6 11 12 14 102 105 353)

step	forms	active	bindvar	bind-time constraint
4	HR1, HR2	T	hour	1 <= hour <= 12
6	HR2	T	minutes	0 <= minutes <= 59
11	LOC6	T	roomnum	roomnum >= 4000
12	CLNUM1	T	class-level	15 <= class-level <= 18
14	CLNUM1	T	class-id	700 <= class-id <= 800
102	DATE1	T	day	1 <= day <= 31
104	DATE1	T	year	year >= 1980
353	FL2	F	flightnum	(member *flightlist*)

Conversion produces:
pcx: 6 (12 12 number)
pcy: 12 (12 12 number)
pcz: 102 (12 12 number)

Figure 5-9: Cloning a pnode.

In total, the *location* fields of the wordclasses indexed by the segments for our sample utterance suggest twenty-four interpretations of those segments. Segmentation-time and bind-time constraints prune the twenty-four to twelve. Figure 5-10 shows the complete set of pcs corresponding to potential leaf nodes in the APT for our sample sentence, arranged by Agenda-level. If we compare this figure to Figure 5-3 we find that only one pnode, (7 7 departuredatemkr), created no pcs (because its wordclass was made inactive during the application of segmentation-time constraints).

The relationship between a pnode and its pcs supports the design decision to make strategy steps into distinct structures that can be shared. A pnode assigns a segment to a class; one pnode must be created for every class to which the segment could belong. A pc assigns a pnode to a step; one pc must be created for every step seeking the pnode's class. If each strategy requiring a constituent of a particular class is given a unique step seeking that class, then the number of pcs created for every pnode will be equal to this number of unique steps. When a member of the class is present in the utterance, each step will be satisfied and will, in turn, create a distinct search path. If, on the other hand, steps are implemented as structures that can be shared across strategies, then the number of pcs created for every pnode can be reduced to the number of unique ways in which a class is used in the grammar. Since every pc represents a unique path through the grammar,

Agenda-level	pc: step (pnode)	Comments
7:	pc0: 705 (0 0 deletewd)	
6:		
5:		
4:	pc1: 401 (1 1 defmkr)	
3:	pc7: 309 (9 9 meetingwd)	
2:	pc8: 201 (10 10 datemkr)	
1:	pc4: 102 (2 2 number)	**3** interpreted as day
	pc6: 129 (7 8 subjname)	
	pc9: 101 (11 11 monthwd)	
	pc12: 102 (12 12 number)	**16** interpreted as day
0:	pc2: 4 (2 2 number)	**3** interpreted as hour
	pc3: 6 (2 2 number)	**3** interpreted as minutes
	pc5: 16 (3 6 nightwd)	
	pc10: 12 (12 12 number)	**16** interpreted as class-level
	pc11: 6 (12 12 number)	**16** interpreted as minutes

Figure 5-10: Possible APT leaf nodes created during segmentation of "Cancel the 3 p.m. speech research meeting on June 16".

redundant search is eliminated. As long as two or more forms occupy the same level of the Formclass Hierarchy, there need be only one copy of any steps they have in common. In this way only one pc need be created per distinct usage at each level, rather than one pc per virtual location (this is especially important as the grammar expands through adaptation and the number of virtual locations increases).

5.2. The Coalesce/Expand Cycle

When segmentation has been completed, the Agenda from Figure 5-10 becomes available to the Coalesce/Expand Cycle. The bottom-up parsing algorithm, shown in Figure 5-11, is quite simple. Explicit in the algorithm is the implementation choice to consider only maximally coalescable sequences; if a constituent can be JOINed to others, it is. This decision makes CHAMP's search heuristic rather than exhaustive (thus, we refer to the design choice as the *Maximal Subsequence Heuristic*). In general the decision is a good one—paths containing EXPANDed non-maximal subsegments will usually fail because segments that should have JOINed will be left dangling, unable to JOIN with higher level constituents. Although it depends on the particular utterance and adapted grammar, experiments with CHAMP have demonstrated that an exhaustive search produces about three times as many nodes as a heuristic search based on maximal

subsequences.[24]

The Coalesce/Expand Cycle

```
FROM Agenda-level 0 TO the highest Agenda level, DO                          (1)
  FOR EACH pair of nodes, x and y, DO
    IF x and y are COALESCABLE
    THEN JOIN them AND merge the new pc back into the current Agenda-level   (1a)
    ELSE IF x is maximal for its class at this Agenda-level
        THEN EXPAND x to higher Agenda-levels                                (1b)
```

Figure 5-11: CHAMP's bottom-up parsing algorithm: behavior of the Coalesce/Expand Cycle at Deviation-level 0.

If we use a heuristic search we must ask what effect the heuristic has on the language recognized by the parser. With maximal subsequences, two distinct problems arise. The first problem occurs whenever two constituents that can JOIN may also be used independently higher in the grammar. As an example, consider what happens to the sentence

"Change the time of the meeting from 3 to 4"

in Figure 5-12. Recall that parsing proceeds bottom-up. At Agenda-level 1, the individual constituents for "3," "to," and "4" JOIN as an unmarked time interval. At Agenda-level 2, the unmarked interval becomes marked. At Agenda-level 3 the pc for "meeting" can coalesce with the marked interval; JOINed, the two subtrees EXPAND to an instance of the class of **meetingforms**. By the time the Coalesce/Expand Cycle finishes Agenda-level 3, however, two different interpretations have been given to the segment "from 3 to 4": one as a marked interval and the other as the source and target of a source-target pair. At Agenda-level 4 the **meetingform** picks up its marker. Independently, the source and target JOIN to become a source-target pair. When processing reaches Agenda-level 5, the **slotform** cannot JOIN with the required source-target pair because the segment "from 3 to 4" is already embedded in the marked **meetingform**. Since the source-target pair of a **slotform** carries a **no-delete** recovery constraint, the parse ultimately fails.

---

[24]The estimate of a three-fold increase is based on a version of the system with exhaustive search. The system was run on the sentences of four users from the experiments described in Chapter 3. The estimate is an average; the actual values depend on the stage of development of the grammar, the particular tokens in the utterance, and the level of deviation required by the parse. Under one set of circumstances, the node count was six times higher under exhaustive search. Although the increase is given as a constant, the impact of that constant on an IBM RT running COMMON LISP is pronounced once memory limitations are approached. In fact, because of the almost continuous need to garbage collect and swap to the disk as more nodes are produced and retained and memory limits are reached, one sentence required more than seventeen hours using the exhaustive search but only five minutes with the heuristic. In two of the four grammars tested, parsing a two-deviation sentence using exhaustive search usually took more than an hour after a certain point in the development of each idiosyncratic grammar. Even Deviation-level 1 sentences were, in general, too slow for the interface to be evaluated on-line.

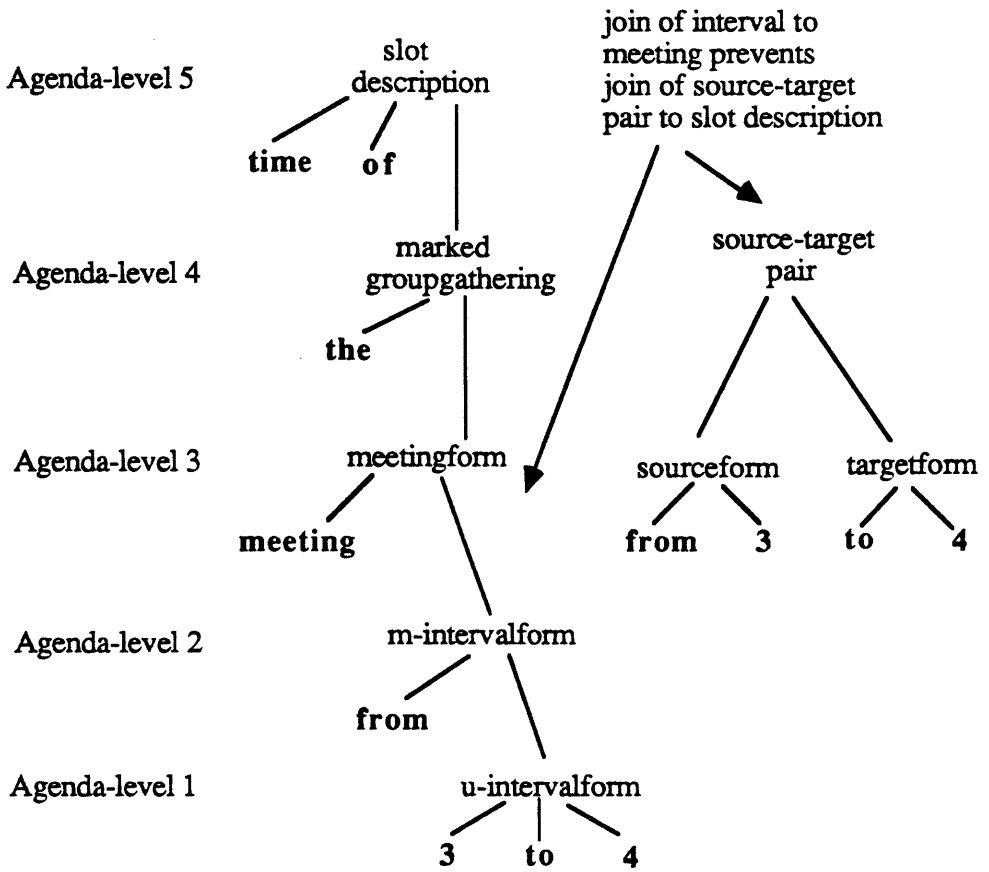

Figure 5-12: One problem with maximal coalesced sequences: JOINing nodes needed independently by higher level constituents in the grammar.

To overcome this problem a fourth type of annotation node for forms was introduced. The *snode* explicitly instructs the parser to independently EXPAND a portion of a maximal subsegment. The kernel **meetingform** includes on its *snode* list the step that seeks a member of **m-intervalforms**. Thus, when the system sees that "meeting" and "from 3 to 4" can coalesce, it EXPANDs both the JOINed node (as shown) and a separate node covering "meeting" alone. Both **meetingform** nodes become marked **groupgatherings** at Agenda-level 4. At Agenda-level 5, however, a pc representing "the meeting" is now available to JOIN with pcs representing "time," "of" and, most important, "from 3 to 4" interpreted as a source-target pair. The subtree produced by that JOIN also produces a successful parse.[25]

The second problem with maximal subsequences is an inherent interaction between

[25]The best way to implement the snode solution would have been through a dynamic analysis of interdependencies in the grammar. In CHAMP, however, the snode lists are hand-coded.

using the heuristic and using a bottom-up parsing algorithm. Figure 5-13 illustrates how a maximal subsequence may prevent the expansion of a correct interpretation and cause the parse to fail. After segmenting "Schedule a June 4 10 p.m. meeting with John," the token **4** will be represented by a number of pcs. One of those pcs correctly interprets **4** as the day portion of an unmarked date. A different pc uses **4** to satisfy the hour step of HR2. HR2 seeks an hour, a colon, minutes, and a **daywd** or **nightwd**. Although the colon is missing, the other steps can be filled if we use the interpretation of **10** as minutes. In this way the segment "4 10 p.m." is JOINed as a candidate **u-hourforms**. Even though this pc fails to expand at Deviation-level 0 for the kernel grammar (because the colon is required and missing), it prevents expansion of the embedded segment "10 p.m." as a **u-hourform**.[26] The example in Figure 5-13 seems contrived because it is extremely difficult to find instances of the problem. In fact, of the 657 utterances from the six users in the adaptive conditions of the experiments described in Chapter 3, not one was rejected by CHAMP because of the Maximal Subsequence Heuristic. The gain in speed seems worth the slight possibility of eliminating the correct parse.

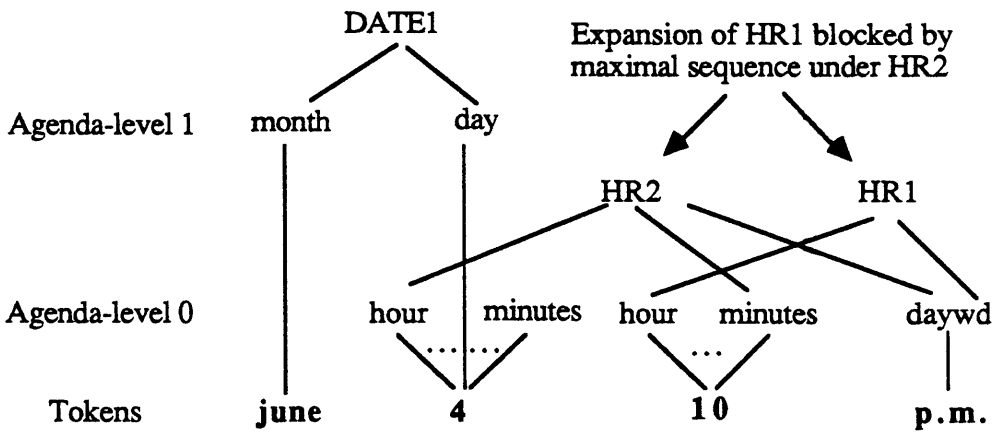

Figure 5-13: Another problem with maximal coalesced sequences: JOINing together a wrong interpretation that masks the correct interpretation.

The Coalesce/Expand Cycle relies primarily on the predicate COALESCABLE and the two functions JOIN and EXPAND. The five criteria under which two nodes are considered COALESCABLE at Deviation-level 0 are shown in Figure 5-14.

The JOIN function is shown in Figure 5-15. JOINing two COALESCABLE nodes preserves some of the work done by the predicate in the pc fields *subs* and *strats*. Recall

[26]In this particular example, the problem would be solved if we removed from the Agenda those JOINed nodes that fail to EXPAND. Unfortunately the solution does not work in general; the problem would still exist if, for example, the user's grammar included a derived form in which the colon was no longer required.

COALESCABLE (pc1 pc2)

Pc1 and pc2 represent contiguous segments of the utterance AND (1)
Both nodes are *resolved* (neither pc's class is **unknown**) AND (2)
There is at least one active form whose strategy list contains the steps in both nodes (pc1 may be a JOINed node containing multiple steps) AND (3)
No step is represented twice AND (4)
No mutually-exclusive steps (sharing an mnode) are present. (5)

Figure 5-14: The behavior of CHAMP's COALESCABLE predicate at Deviation-level 0.

that *subs* is used to record co-constituent nodes while *strats* is used to record the forms that parse the *subs*.[27]

JOIN (pc1 pc2)

IF pc1 is already a JOIN node (1)
THEN add pc2 to pc1's *subs* field
    let pc1's *strats* field include only the active strategies parsing all the *subs*
ELSE create a new JOIN node (2)
    let its *subs* field contain pc1 and pc2 AND
    let its *strats* field contain the active strategies that parse both subnodes

Figure 5-15: The behavior of CHAMP's JOIN algorithm at Deviation-level 0.

Recall from steps (1a) and (1b) of the Coalesce/Expand Cycle that JOINed nodes are merged back into the current Agenda-level until they represent maximally coalesced sequences; then they are EXPANDed. Figure 5-16 demonstrates the effect of the Coalesce phase on Agenda-level 0 for our sample sentence, "Cancel the 3 p.m. speech research meeting on June 16." Only two node pairs are COALESCABLE. Pc2 (which interprets "3" as an hour) and pc5 (representing "p.m.") can coalesce by way of strategies HR1 and HR2. Pc3 (which interprets "3" as minutes) can also coalesce with pc5, but only by way of HR2. Pc10 and pc12 are not COALESCABLE because they are discontiguous with the other nodes at the Agenda-level. Note that although the step field of a JOIN node is empty, the node is considered by step (3) of the COALESCABLE predicate to represent the steps of the subnodes.

[27]Because the field is named *strats* we will generally refer to the values that fill the field as "strategies" rather than by their proper name, "forms." Thus the reader may consider "strategy" to be synonymous with "form." In order to avoid confusion, we will use the full phrase "strategy list" when we wish to refer only to that portion of a form contained in the *strategy* field.

Agenda-level 0	Pcs Created by JOIN
pc2: 4 (2 2 number)	
pc3: 6 (2 2 number)	pc13: nil (2 6 join) (HR1 HR2) (2 5)
pc5: 16 (3 6 nightwd)	pc14: nil (2 6 join) (HR2) (3 5)
pc10: 12 (12 12 number)	
pc12: 6 (12 12 number)	

Figure 5-16: The JOIN nodes produced by coalescing Agenda-level 0 for "Cancel the 3 p.m. speech research meeting on June 16."

JOINing nodes together creates virtual grammatical constituents. EXPAND's responsibility is to create the actual higher level pcs and place them on the Agenda. The algorithm accomplishing this is presented in Figure 5-17. The process of EXPANDing a set of co-constituents is essentially the same as that of converting a pnode to a set of pcs. Just as we created one pc for each step seeking the pnode's class, so we want to create one pc for each step seeking each formclass represented by a maximal set of co-constituents.

```
EXPAND (pc)
IF the pc is a JOINed node                                          (1)
THEN let the strategies = the strats field
       let the subnodes = the subs field
ELSE let the strategies = the active locations of the pc's step
       let the subnodes = the pc itself
FOR EACH strategy, DO                                               (1a)
  IF the subnodes violate no expectations for that strategy
  THEN consider that strategy successful
Partition the successful strategies into classes by their isa fields (2)
FOR EACH class, DO                                                  (3)
  IF the subnodes satisfy all expand-time constraints for this class
  THEN create a pnode for this class that spans the subnodes
         IF there is an expand-time ig for this class
         THEN invoke it AND cache the canonical value for this pnode
  FOR EACH active step that seeks the pnode's class, DO             (4)
    IF the step has a bind-time constraint AND the pnode satisfies it
    THEN create a pc for the step, pnode, subnodes, and strategies and
           place it on the appropriate level of the Agenda
```

Figure 5-17: CHAMP's EXPAND algorithm

To create each higher level pc we must fill its five fields: *step*, *pnode*, *subs*, *strats*, and *dlevel*. Two of the values we need are immediately available: the *dlevel* of a pc is always zero at Deviation-level 0 and the *subs* field is simply the *subs* in a JOIN node or the pc itself if it is maximal (step (1) in Figure 5-17).

The value to fill the *strats* field of the higher level pcs also seems to be immediately available until we consider that a set of subconstituents may violate a strategy's expectations. COALESCABLE only guarantees that a set of constituents can be explained by a set of *strats*, not that they can be explained without deviation. Step (1a) of the EXPAND algorithm performs the error detection. As discussed in Chapter 2, a violated expectation corresponds to an insertion, deletion, transposition or substitution with respect to the context established by a form. In Figure 5-16, for example, the attempt to EXPAND pc14 will fail because of a violated expectation in HR2. HR2 expects an hour, colon, minutes and day or night marker. Since only the minutes and night marker are bound, the time is incompletely specified. Two deletions would be required to explain "3 p.m." using HR2, causing a failure along this path at Deviation-level 0.

It is possible that the strategies that remain after error detection fall into more than one formclass. Since a pnode assigns a unique class to a segment, the next step in expansion (step (2)) partitions the strategies into distinct formclasses. Step (3) demonstrates that we create a pnode for a formclass only if the coalesced subconstituents satisfy the class's expand-time constraints. Expand-time constraints were introduced in Section 4.3; they correspond to intercase dependencies. Although the constituents in our sample sentence do not invoke any expand-time constraints, a source-target pair, for example, must satisfy the constraint that the value bound to the **source** is compatible with the value bound to the **target**. Thus, the constraint accepts interpretations that change a meeting from one location to another or from one time interval to another, but rejects interpretations that try to change a location to a time interval.

Step (3) also shows the point at which **exp**-type instance generators are invoked. The canonical value produced by the generator is computed once for a set of subnodes and a class; by tying the canonical value to the pnode, all the pcs that share that pnode also share the value.

Pcs that share a pnode are nevertheless distinguished by the values in their *step* fields. The steps seeking the formclass in each pnode are available through the formclass's *locations* field. Note that the fourth step in EXPAND is the same as the last step of SEGMENT (Figure 5-2); bind-time constraints are applied during expansion as well.

We now have all the information needed to create a set of higher level pcs from the constituent passed to EXPAND. One pc is created for each step seeking each formclass represented by one or more strategies that parsed the maximal subsegment. Figure 5-18 shows the result of EXPANDing pc13.

Maximal subsegment	Pcs created by expansion
pc13: nil (2 6 join) (HR1 HR2) (2 5)	pc15: 113 (2 6 u-hourforms) (HR1) (2 5)
	pc16: 107 (2 6 u-hourforms) (HR1) (2 5)
	pc17: 109 (2 6 u-hourforms) (HR1) (2 5)
	pc18: 302 (2 6 u-hourforms) (HR1) (2 5)

Figure 5-18: The result of applying EXPAND to pc13.

The versions of SEGMENT, COALESCABLE, JOIN, and EXPAND described in this chapter define the behavior of a bottom-up parser that uses context-sensitive constraints to reduce search. In the next chapter we examine how to modify and generalize these algorithms to produce a least-deviant-first parser whose output acts as a source of new grammatical components. Before extending the system's capabilities, however, we conclude our description of the basic bottom-up parser by following our sample sentence through the remainder of the Coalesce/Expand Cycle at Deviation-level 0.

5.3. A Detailed Example: The Complete Coalesce/Expand Cycle for "Cancel the 3 p.m. speech research meeting on June 16."

After the Coalesce phase, Agenda-level 0 contains seven nodes (the original 5 plus the two JOIN nodes, pc13 and pc14). The fate of each node during expansion is shown in Figure 5-19. Although only one node (pc13) can be EXPANDed, it produces four new pcs: one pc for each location of **u-hourforms** that is still active in the grammar (four locations seeking **u-hourforms** were made inactive by segmentation-time constraints). Since **u-hourforms** has an associated expand-time instance generator, the canonical representation of "3 p.m.," **1500**, is shared by pcs 15 through 18.

Agenda-level 0	New pcs or reason for non-expansion
pc2: 4 (2 2 number)	not maximal (sub of pc13)
pc3: 6 (2 2 number)	not maximal (sub of pc14)
pc5: 16 (3 6 nightwd)	not maximal (sub of pc13 and pc14)
pc10: 12 (12 12 number)	requires 2 deletions in CLNUM1
pc12: 6 (12 12 number)	requires 2 deletions in HR2
pc13: nil (2 6 join) (HR1 HR2) (2 5)	pc15: 113 (2 3 u-hourforms) (HR1) (2 5)
	pc16: 107 (2 6 u-hourforms) (HR1) (2 5)
	pc17: 109 (2 6 u-hourforms) (HR1) (2 5)
	pc18: 302 (2 6 u-hourforms) (HR1) (2 5)
pc14: nil (2 6 join) (HR2) (3 5)	requires 2 deletions in HR2

Figure 5-19: Results from EXPANDing Agenda-level 0 for "Cancel the 3 p.m. speech research meeting on June 16."

The last step in EXPAND places the new pcs onto the Agenda; pc15, pc16, and pc17

are placed at Agenda-level 1, and pc18 is placed on Agenda-level 3, in accordance with the position of their step's strategies in the Formclass Hierarchy. Once expansion is over at Agenda-level 0, the Coalesce/Expand Cycle returns to the Coalesce phase to process the next level of the Agenda.

Figure 5-20 shows the results from processing Agenda-level 1. "June" and the interpretation of "16" as a day first JOIN (pc19, in the left-hand column of the last row) then EXPAND to create the unmarked dates sought by step 203 and step 303. In addition, "speech research" (pc6) becomes a member of the class **u-subjectforms**. Note that pc6 did not have to JOIN in order to be EXPANDed because IFSUBJ (short for Instance Form of SUBJect) requires only a **subjname** for success. The remaining pcs at this level all have interpretations only as subconstituents: pc4 represents part of an unmarked date (in particular, the day), pc15 represents the hour portion of a marked hour, pc16 represents the starting hour in an unmarked interval, and pc17 tries to use the same segment in the utterance to fill the ending hour. Without their co-constituents, these pcs cannot EXPAND at Deviation-level 0.

Agenda-level 1	New pcs or reason for non-expansion
pc4: 102 (2 2 number)	requires a deletion in DATE1
pc6: 129 (7 8 subjname)	pc20: 214 (7 8 u-subjectforms) (IFSUBJ) (6)
	pc21: 307 (7 8 u-subjectforms) (IFSUBJ) (6)
pc9: 101 (11 11 monthwd)	not maximal (sub of pc19)
pc12: 102 (12 12 number)	not maximal (sub of pc19)
pc15: 113 (2 6 u-hourforms) (HR1) (2 5)	requires a deletion in HR0
pc16: 107 (2 6 u-hourforms) (HR1) (2 5)	requires 2 deletions in INT1
pc17: 109 (2 6 u-hourforms) (HR1) (2 5)	requires 2 deletions in INT1
pc19: nil (11 12 join) (DATE1) (9 12)	pc22: 203 (11 12 u-dateforms) (DATE1) (9 12)
	pc23: 303 (11 12 u-dateforms) (DATE1) (9 12)

Figure 5-20: The Coalesce/Expand Cycle applied to Agenda-level 1 for "Cancel the 3 p.m. speech research meeting on June 16."

Figure 5-21 shows the effect of the Coalesce/Expand Cycle on Agenda-level 2 where the unmarked date produced at Agenda-level 1 picks up its marker (pc24). The unmarked **subjectform** produced at Agenda-level 1 has two interpretations in the kernel grammar: one as part of a postnominal marked subject (pc20 in Figure 5-20) and the other as a prenominal modifier (pc21 in Figure 5-20). Agenda-level 2 sees the demise of the marked interpretation because there is no marker to support it.

Activity at Agenda-level 3 (Figure 5-22) is interesting for three reasons. First, we see another example of the *snode* annotation's effects. Two pcs are produced during the JOINing of "3 p.m.," "speech research," "meeting," and "on June 16" because step 316 of pc25 is on GG1's snode list: pc30 advances the segment with the date, while pc31 advances the segment without it.

Agenda-level 2	New pcs or reason for non-expansion
pc8: 201 (10 10 dtmkr)	not maximal (sub of pc24)
pc20: 214 (7 8 u-subjectforms) (IFSUBJ) (6)	requires a deletion in SUBJ0
pc22: 203 (11 12 u-dateforms) (DATE1) (9 12)	not maximal (sub of pc24)
pc24: nil (10 12 join) (DATE0) (8 22)	pc25: 316 (10 12 m-dateforms) (DATE0) (8 22)
	pc26: 703 (10 12 m-dateforms) (DATE0) (8 22)

Figure 5-21: The Coalesce/Expand Cycle applied to Agenda-level 2 for "Cancel the 3 p.m. speech research meeting on June 16."

Agenda-level 3	New pcs or reason for non-expansion
pc7: 309 (9 9 meetingwd)	not maximal (sub of pc28)
pc18: 302 (2 6 u-hourforms) (HR1) (2 5)	not maximal (sub of pc28 and pc29)
pc21: 307 (7 8 u-subjectforms) (IFSUBJ) (6)	not maximal (sub of pc28 and pc29)
pc23: 303 (11 12 u-dateforms) (DATE1) (9 12)	requires 1 deletion in GG1-GG4
pc25: 316 (10 12 m-dateforms) (DATE0) (8 22)	not maximal (sub of pc29)
pc27: nil (2 8 join) (GG4 GG1) (18 21)	not maximal (masked by pc28)
pc28: nil (2 9 join) (GG1) (18 21 7)	pc31: 403 (2 9 meetingforms) (GG1) (18 21 7)
pc29: nil (2 12 join) (GG1) (18 21 7 25)	pc30: 403 (2 12 meetingforms) (GG1) (18 21 7 25)

Figure 5-22: The Coalesce/Expand Cycle applied to Agenda-level 3 for "Cancel the 3 p.m. speech research meeting on June 16."

As a second point of interest, Agenda-level 3 gives us the opportunity to introduce the concept of a *critical difference* between forms:

- **Critical Difference**: a form, X, contains a critical difference with respect to another form, Y, if X and Y contain the same steps with different ordering constraints, or if X and Y differ by at least one step.

The JOIN node produced by coalescing "3 p.m." and "speech research" is pc27 which contains GG4 and GG1 in its *strats* field. In creating pc27 we do not know if the combined segment is a reference to a meeting (GG1) or a meal (GG4) because steps 302 and 307 are shared by the two forms. By coalescing "meeting" onto "3 p.m. speech research," step 309 reduces the set of possible strategies to GG1 (see pc31). Thus, we say that step 309 is a *critical difference* between GG1 and GG4 because the satisfaction of step 309 by a segment in the utterance is enough to eliminate GG4 from further consideration. Clearly, critical differences between forms are an important factor in controlling search. We will have more to say about critical differences in the next chapter.

Our final observation about Agenda-level 3 also involves pc27 and the segment "3 p.m. speech research." Note that it is not EXPANDed because it is masked by the maximal segment in pc29 (pc28 is EXPANDed despite being non-maximal because it was created

by an *snode* annotation). It is masked by pc29 because pc29 and pc27 share a strategy (GG1) in the same formclass (**meetingforms**)—interpreting "3 p.m. speech research" as part of a **meetingform** via GG1 enables the system to create a larger segment than interpreting the same portion of the utterance as a **mealform** via GG4. By not EXPANDing the alternate interpretation of "3 p.m. speech research," the potential reference to a **mealform** is lost. In an exhaustive search we would have had to preserve that interpretation because it is distinct from the interpretation of the segment as a **meetingform**. Although pc27 would have failed to EXPAND at Deviation-level 0 (because the required head noun is missing), it would have EXPANDed had the parse proceeded to Deviation-level 1. Once created, that fragment would progress up the Agenda creating spurious interpretations at each level—interpretations that fail globally for this utterance. The Maximal Subsequence Heuristic prevents the unnecessary work; regardless of the deviation-level, a smaller segment will always be masked by a larger one if they share a strategy.

Agenda-level 4	New pcs or reason for non-expansion
pc1: 401 (1 1 defmkr)	not maximal (sub of pc32 and pc33)
pc30: 403 (2 12 meetingforms) (GG1) (18 21 7 25)	not maximal (sub of pc33)
pc31: 403 (2 9 meetingforms) (GG1) (18 21 7)	not maximal (sub of pc 32)
pc32: nil (1 9 join) (DGG1) (1 31)	pc34: 709 (1 9 m-d-ggforms) (DGG1) (1 31)
pc33: nil (1 12 join) (DGG1) (1 30)	pc35: 709 (1 12 m-d-ggforms) (DGG1) (1 30)

Figure 5-23: The Coalesce/Expand Cycle applied to Agenda-level 4 for "Cancel the 3 p.m. speech research meeting on June 16."

At Agenda-level 4 (Figure 5-23), each of the **meetingforms** produced at Agenda-level 3 picks up its definite article. Pc34 and pc35 reflect the use of the noun phrase as the subject of the sentence. If segmentation-time constraints had not turned off all the constituents in the grammar associated exclusively with **changeforms**, the marked group-gatherings would also be EXPANDed to permit embedding of the noun phrase in a slot description ("location of the 3 p.m. speech research meeting") at Agenda-level 5. The **slotform** would then have a chance to pick up its marker at Agenda-level 6. In our example, however, Agenda-levels 5 and 6 are empty, and the Coalesce/Expand cycle continues processing at Agenda-level 7 (Figure 5-24).

Agenda-level 7 shows the ultimate fate of the extra pc produced by the *snode* annotation during expansion of Agenda-level 3 (pc31). JOINed with pc1 at Agenda-level 4, pc31 reaches Agenda-level 7 as part of pc34. Pc34 represents a marked **meetingform** that does not contain the marked date "on June 16." Pc26, contiguous with pc34, appears to make the marked date available. However, since pc26 actually captures a segment of the utterance after the verb and ACT2 expects an introductory adverbial phrase, pc34 and pc26 may coalesce but may not EXPAND due to the violated ordering expectation. Of

Agenda-level 7	New pcs or reason for non-expansion
pc0: 705 (0 0 deletewd)	not maximal (sub of pc39)
pc26: 703 (10 12 m-dateforms) (DATE0) (8 22)	not maximal (sub of pc38)
pc34: 709 (1 9 m-d-ggforms) (DGG1) (1 31)	not maximal (sub of pc36)
pc35: 709 (1 12 m-d-ggforms) (DGG1) (1 30)	not maximal (sub of pc37)
pc36: nil (0 9 join) (ACT2) (0 34)	not maximal (masked by pc37)
pc37: nil (0 12 join) (ACT2) (0 35)	pc39: *ROOT* (0 12 deleteforms) (ACT2) (0 35)
pc38: nil (0 12 join) (ACT2) (0 34 26)	requires 1 transposition in ACT2

Figure 5-24: The Coalesce/Expand Cycle applied to Agenda-level 7 for "Cancel the 3 p.m. speech research meeting on June 16."

course, pc0 and pc35 do not have this problem because the date has already been explained as a postnominal modifier; they coalesce and EXPAND to produce pc39.

When all of the words in an utterance have been explained, one or more APTs with the special link *ROOT* will be placed by EXPAND at Agenda-level 8. The APT for our sample sentence, rooted at pc39, is given in Figure 5-25 in both CHAMP's internal form and in a more conventional tree form. In the internal form we do not list the value of the *subs* field but indicate the subtree relationship by indentation. The tree structure in the figure demonstrates that in addition to capturing the meaning of the utterance, the APT provides a virtual trace of the parse through the *step* and *strats* fields. By following these fields we can replay the successful path through the search space: apply steps 705 and 709 from ACT2, followed by steps 401 and 403 from DGG1, followed by step 302 from GG1, steps 4 and 16 from HR1, and so on.

Producing a *ROOT* for an APT signals the end of parsing but not the end of the understanding process. The explanation (or explanations if the sentence is ambiguous with respect to the current grammar) produced at Deviation-level 0 must still be checked for consistency against the state of the application databases. This is done during Resolution (see Chapter 7). If the explanation makes sense in the current state of the world, then the user's request is carried out without further interaction. As the final stage of understanding, an APT (even one produced at Deviation-level 0) is examined by the adaptation functions to see if it contains information for removing overgeneralizations in the adapted kernel (see Chapter 8).

As an adapted kernel grows to include a user's preferred forms of expression, the majority of utterances the system encounters will be directly recognizable by the process described in this chapter. It is imperative, therefore, that an adaptive interface understand non-deviant input efficiently. CHAMP accomplishes this goal by controlling the number of paths followed at each stage of the bottom-up parse. Controlling search takes three forms during segmentation: the Single Segment Assumption limits the number of

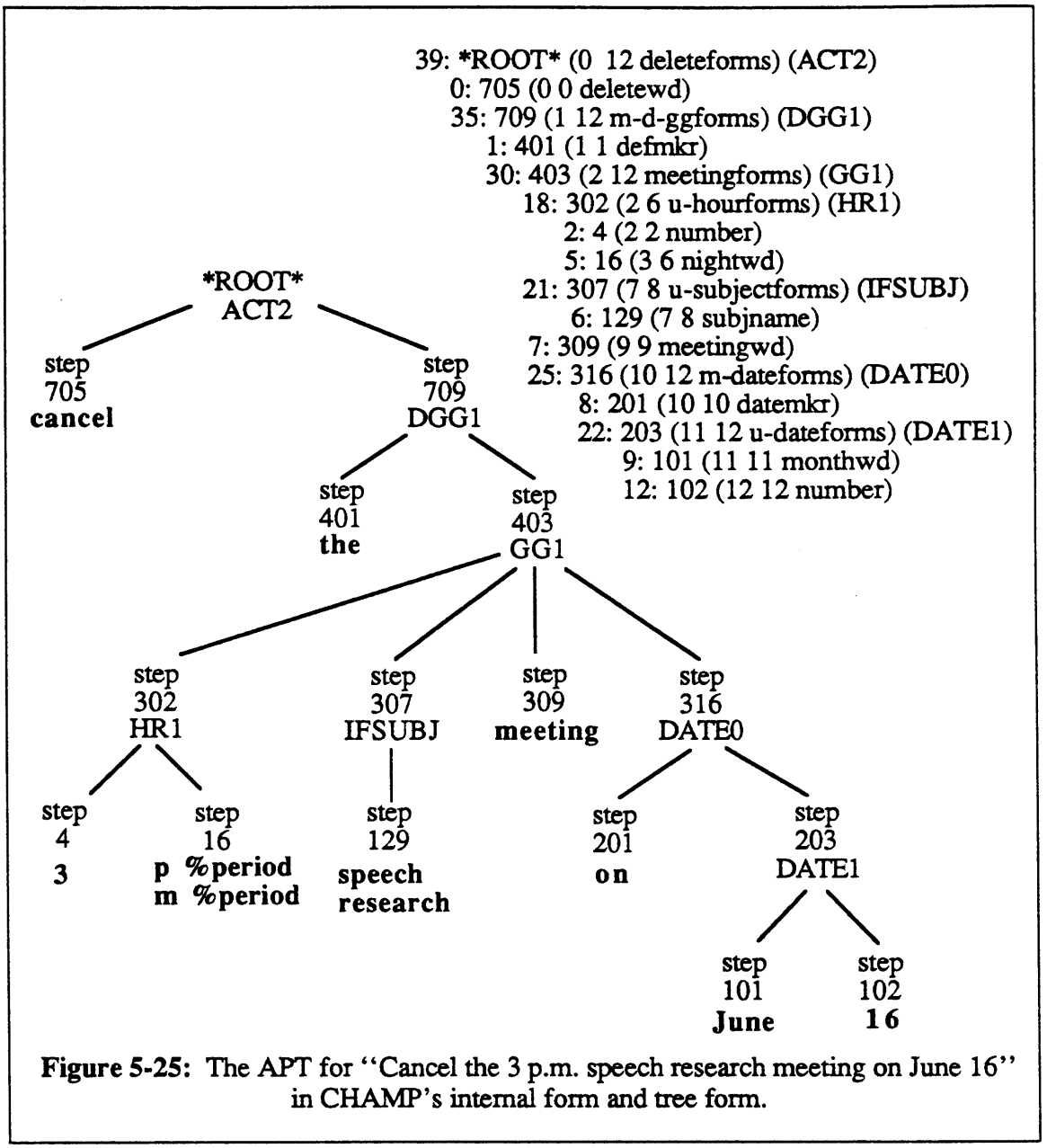

Figure 5-25: The APT for "Cancel the 3 p.m. speech research meeting on June 16" in CHAMP's internal form and tree form.

interpretations given to unknown segments, segmentation-time constraints may turn off whole areas of the search space, and bind-time constraints may eliminate the interpretation of a segment as a member of a particular formclass. During the Coalesce phase of the Coalesce/Expand Cycle, search is controlled by gathering only maximal sequences of co-constituents in active classes, as well as by the effects of incorporating critical differences into the kernel strategies. During the Expand phase both bind-time and expand-time constraints may be applied to candidate constituents in active classes. More importantly, during expansion at Deviation-level 0, a path is terminated if a violated expectation is discovered.

A different aspect of performing well in the long run demands that categories like names and topics be extendable. Even after the user's grammar has stabilized she is likely to continue to meet new people and discuss new subjects. Given an adapted kernel that captures the stabilized grammar, it will be the case that an unknown segment is a new instance of an extendable class most of the time. Recognizing this fact, CHAMP attempts to resolve unknown segments as new instances *before* proceeding with a lengthy search through the space of deviation hypotheses.

Of course unknown segments cannot always be resolved as new instances of known classes and user utterances cannot always be explained within the current grammar. In the next chapter we will see how the framework we have constructed for understanding non-deviant utterances can be extended to perform error recovery as well.

Chapter 6
Detecting and Recovering from Deviation

What are the changes that will transform the basic bottom-up parsing algorithm described in Chapter 5 into one that tolerates deviant input? By comparing Figures 5-1 and 6-1, we see that the key lies in extending the error detection mechanism to include error recovery. We know that error detection is the discovery of violated expectations in the context provided by a form and a set of co-constituents (see Figure 5-17). When we were considering only Deviation-level 0, error detection was a yes-or-no proposition; either a set of subnodes satisfied a strategy or they did not. If the answer was "yes" then we considered the strategy successful and EXPANDed the co-constituents as an instance of the recognized formclass. If the answer was "no" the search path represented by the context was simply terminated—at Deviation-level 0 no violated expectations are tolerated. It follows that to understand deviant utterances within this framework we must be willing to relax a form's expectations in a principled way without losing sight of what constitutes a good explanation. Following the model presented in Chapter 2, we have implemented the general recovery actions as our principled method of relaxing expectations. As a result, we define the best explanation of a deviant utterance as the one that requires the fewest recovery actions. Finding the best explanation is accomplished by a least-deviant-first search using the current grammar and successive applications of the recovery actions. Thus, if a deviant utterance can be explained by performing only one recovery action we say that it succeeds at Deviation-level 1, or, alternatively, that the utterance requires one deviation. An utterance requiring two deviations succeeds at Deviation-level 2 and *has no less deviant explanation within the current grammar.*

Figure 6-1 also reminds us that CHAMP's basic control structure is the Parse/Recovery Cycle which is made up of the Coalesce/Expand Cycle and the Error Detection & Recovery mechanism. Figure 6-2 shows that the Parse/Recovery Cycle is implemented in CHAMP's READLOOP through successive invocations of the Coalesce/Expand Cycle at increasing deviation-levels. Each time through the cycle, paths that failed at the previous deviation-level may be extended by a recovery action, reflecting increasing deviance in the partial APTs constructed. When describing our model of deviation and recovery in Section 2.2 we defined "Deviation-level" to be "a value that indicates the number of deviations permitted at a given point in the search for a meaning structure." Rephrasing the definition in terms more specific to the implementation, we redefine

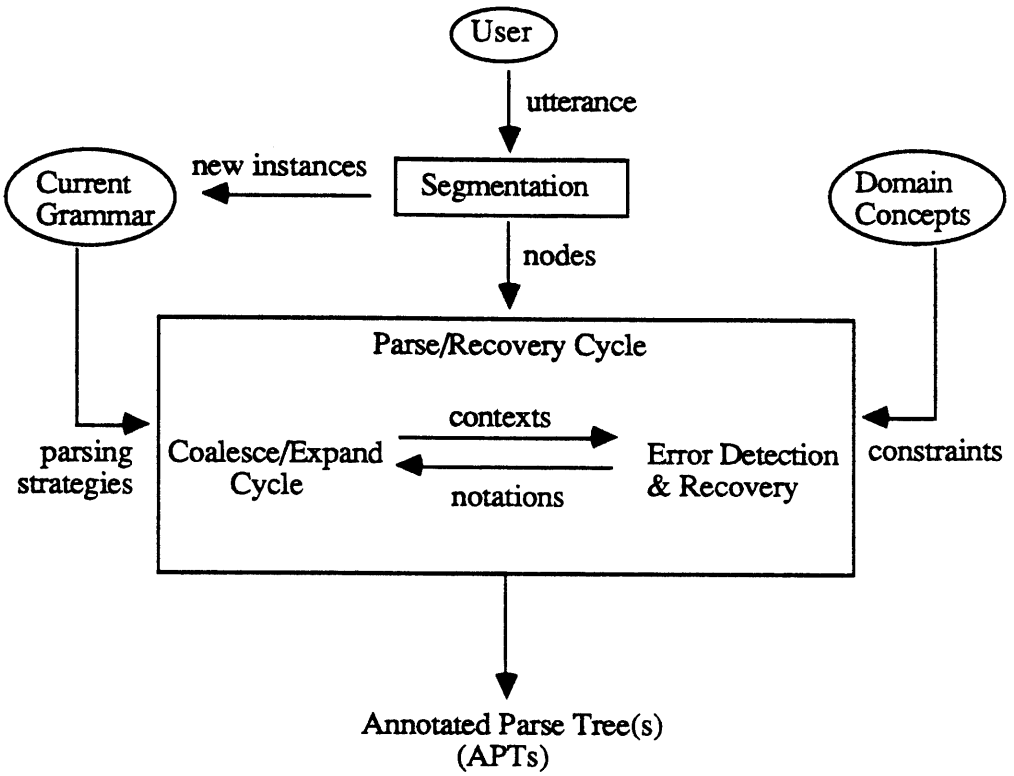

Figure 6-1: Making the transition from bottom-up to least-deviant-first parsing by adding error recovery (detail of Figure 4-1).

- **Deviation-level**: the total number of recovery actions permitted along a search path during the current Parse/Recovery cycle. Equivalently, the total amount of deviance that may accumulate in any partial APT during the current cycle.

The algorithm in Figure 6-2 also introduces

- **Maximum Deviation-level**: the largest number of recovery actions permitted along every search path before the parse is considered to have failed. In CHAMP this number is fixed at two, partly to keep the system's response-time reasonable, and partly to control the quality of the explanations produced.[28]

In the next section we examine how the parsing process must be changed to accommodate deviant input. Specifically, we reexamine most of the algorithms presented in the previous chapter, describing how to generalize them to function at any deviation-

[28]Note, however, that the restriction to two deviations in an utterance is a limitation of the implementation, *not* the model. In addition, the implementation accepts the Maximum Deviation-level as an argument—with one exception CHAMP is actually designed to work for an arbitrary maximum deviation limit. The exception is in the recovery mechanism's decomposition of cases (see Section 6.5).

READLOOP ()
UNTIL the end of the session, DO
  Set the Deviation-level to 0 (1)
  Get the next utterance (2)
  SEGMENT the utterance (3)
  IF the number of unknown segments > the Maximum Deviation-level (4)
  THEN give the user helpful feedback but consider the utterance unparsable
  ELSE UNTIL the Deviation-level > the Maximum OR a *ROOT* appears, DO (5)
    Run the Coalesce/Expand Cycle
    Increment the Deviation-level

Figure 6-2: CHAMP's READLOOP without adaptation
(The Parse/Recovery Cycle is Step (5))

level. Once we understand the parsing phase of the Parse/Recovery Cycle, we will turn our attention to the related processes of error detection (Section 6.3) and error recovery (Section 6.5).

6.1. Parsing as Least-deviant-first Search

If our goal is to be certain that we have found the best explanation for an utterance, then we must not accumulate more deviance along a search path than is permitted by the value of the deviation-level during the current cycle. To accomplish our goal is to generalize the parsing process described in Chapter 5 to create a least-deviant-first search. We begin by making a single alteration to CHAMP's SEGMENT algorithm, previously introduced in Figure 5-2. Figure 6-3 gives the actual algorithm used for segmentation. The only difference between the algorithms occurs in step (10). Throughout this chapter differences between old and new versions of an algorithm are marked by boldface step numbers.

The new version of step (10) creates pcs with a *dlevel* of one for unknown segments. We call such a pc *unresolved* because the segment it represents has yet to be assigned a meaningful wordclass and step. Recall that the *dlevel* field of a pc keeps track of the amount of deviance accumulated in the subtree rooted at that node. Since an unresolved pc must be explained via a substitution or insertion recovery, an unresolved pc has a *dlevel* of one by definition.

The generalized Coalesce/Expand Cycle (Figure 6-4) differs from its predecessor by including the partitioning steps that used to be in EXPAND (see Figure 5-17). Recall that the purpose of the partitioning is to separate by class the higher level grammatical constituents to which a set of pcs may be EXPANDed. Figure 6-5 demonstrates: the

SEGMENT (string)
Convert the string to tokens (1)
FOR each token in the utterance, DO
  IF the token is a number
  THEN create a pnode with class **number** (2)
  IF a token has no definition (3)
  THEN apply spelling correction
    IF the user accepts a spelling correction
    THEN replace the token with the correct word
    ELSE create a pnode with class **unknown**
  FOR EACH lexical definition of the token as part of a phrase, DO (4)
    IF the whole phrase is present and maximal in length
    THEN create a pnode for the definition's class spanning the whole phrase (4a)
    ELSE IF some word in the phrase is present in the utterance and there is no
      definition for that word by itself (4b)
      THEN create a pnode for **partof** the class spanning the subphrase
    IF the token was not captured by a completed phrase (5)
    THEN FOR EACH definition of the token that is not a phrase definition, DO
      create a pnode for the token and definition's class
IF contiguous pnodes have class **unknown** or are **partof** different phrases\(6)
THEN combine the pnodes into one of class **unknown**
IF a pnode represents **partof** a phrase that is NOT in an extendable class (7)
THEN make the pnode **unknown**
ELSE try resolve it via interaction as an abbreviation of the phrase and
    IF it can't be resolved as an abbreviation
    THEN make it **unknown**
IF a pnode is **unknown** (8)
THEN try to resolve it as a new instance of an extendable class
Apply segmentation-time constraints (9)
FOR EACH pnode with an active wordclass, DO **(10)**
  IF the pnode is **unknown**
  THEN create an unresolved pc with step=0 and dlevel=1 and
    place it on Agenda-level 0
  ELSE FOR EACH active step that seeks the pnode's class, DO
    IF the step has a bind-time constraint AND the pnode satisfies it
    THEN create a pc for that step and pnode and place it on the Agenda

Figure 6-3: CHAMP's SEGMENT Algorithm
(Figure 5-2 generalized to enforce a least-deviant-first search).

JOINed node (pcz) contains co-constituents (pcx and pcy) that are sought by GG1, GG2,

The Coalese/Expand Cycle

```
FROM Agenda-level 0 TO the highest Agenda level, DO                    (1)
  FOR EACH pair of nodes, pc1 and pc2, DO
    IF pc1 and pc2 are COALESCABLE
    THEN JOIN them and merge the new pc back into the current Agenda-level  (1a)
    ELSE CASE                                                          (1b)
          pc1 is unresolved:
            move pc1 to the next Agenda-level
          pc1 is a JOINed node:
            let strategies = the strats field
            let subnodes = the subs field
          pc1 is neither JOINed nor unresolved:
            let strategies = the active locations of the pc's step
            let subnodes = the pc itself
          Partition the strategies by formclass
          FOR EACH partition, DO
            EXPAND the context to higher Agenda-levels
```

Figure 6-4: CHAMP's Coalesce/Expand Cycle
(Figure 5-11 generalized to enforce a least-deviant-first search).

GG3, and GG4. Since each strategy explains a higher level constituent belonging to a different class, each strategy forms a distinct partition that, in turn, defines a different

- **Context**: a candidate grammatical constituent. A context is represented by a list of subnodes and a list of strategies. The subnodes are co-constituents that define members of the higher candidate class, the strategies are the forms that recognize members of the higher class by those co-constituents. Thus, all the strategies within a context recognize the same candidate class.

Each context, therefore, corresponds to a different path in the search space by explaining a particular segment as an instance of a particular class. A path that succeeds locally at Deviation-level *i* may still fail to produce a global parse—explaining an utterance may require extending a path that fails at Deviation-level *i* but succeeds at a higher one. At the end of this section we will see that partitioning candidate constituents into contexts *before* EXPANDing them helps CHAMP pinpoint exactly which search paths should be extended each time through the Parse/Recovery Cycle.

To insure a least-deviant-first search, the components of the Coalesce/Expand Cycle must also be generalized. Specifically, each of COALESCABLE, JOIN and EXPAND must be altered to insure that the amount of deviance along a search path does not exceed the current deviation-level. The *dlevel* field of a pc keeps track of the amount of deviance accumulated in the subtree rooted at that node. Similarly, we define the *dlevel* of a context to be the sum of the *dlevels* of its subnodes.

Utterance: "Change the 3 pm June 11 [meeting] to 4 pm."

During The Coalesce Phase:
pc*x* created from **3 + pm**
pc*y* created from **June + 11**
pc*z*: nil (2 5 join) (GG1 GG2 GG3 GG4) (pc*x* pc*y*)

Before EXPAND:
Four contexts created from pc*z*:
1: [(pc*x* pc*y*) (GG1)] explains "3 pm June 11" as part of a **meetingform**
2: [(pc*x* pc*y*) (GG2)] explains "3 pm June 11" as part of a **seminarform**
3: [(pc*x* pc*y*) (GG3)] explains "3 pm June 11" as part of a **classform**
4: [(pc*x* pc*y*) (GG4)] explains "3 pm June 11" as part of a **mealform**

Figure 6-5: Distinguishing search paths by creating contexts during the Coalesce/Expand Cycle

To provide for the effect of a subtree's *dlevel* on search, CHAMP's COALESCABLE predicate is changed from the algorithm in Figure 5-14 to the one in Figure 6-6. Step (2) of the original algorithm prevented coalescing an unresolved pc—representing an unknown or incomplete phrase—with one or more resolved pcs. Step (2) of Figure 6-6 generalizes this idea; it prevents the coalescing of any two pcs whose combined deviance is greater than that permitted in the current cycle.

COALESCABLE (pc1 pc2)

Pc1 and pc2 represent contiguous segments of the utterance AND	(1)
The sum of the *dlevels* of the nodes <= the current deviation-level AND	(2)
There is at least one active form containing the steps of both nodes AND	(3)
No link is represented twice AND	(4)
No mutually-exclusive links (sharing an mnode) are present.	(5)

Figure 6-6: CHAMP's COALESCABLE Predicate (Figure 5-14 generalized to all deviation-levels).

To see the effect of the true COALESCABLE predicate on processing, consider the sentence

"Change the 3 pm seminar June 4 to rm 7620"

which contains two deviations with respect to the kernel: a deletion of the required **datemkr** in the postnominal date, and a substitution of the token **rm** for a recognizable token of the class **roomwd**. Figure 6-7 shows COALESCABLE's uniform effect at different deviation-levels. The figure arranges the results of processing by increasing deviation-level. The leftmost column at each deviation-level shows some of the maximal

sets of co-constituent subtrees. The middle column shows whether or not the subtrees can JOIN. The rightmost column explains what happens to the candidate context during expansion. The last row of Deviation-level 0 shows, for example, that a previously constructed, non-deviant unmarked hour (u-hour@0) may JOIN with the token **seminar** and EXPAND via GG2 to a subtree representing a non-deviant **seminarform**.

Utterance: "Change the 3 pm seminar [on] June 4 to *rm* 7620"

Coalesce/Expand Cycle at Deviation-level 0:

Subtrees	COALESCABLE	EXPAND
3 + pm	yes	u-hour@0, HR2
June + 4	yes	u-date@0, DATE1
rm@1 + 7620	no	
u-date@0	maximal	fails (missing marker)
u-hour@0 + seminar	yes	seminar@0, GG2

Coalesce/Expand Cycle at Deviation-level 1:

Subtrees	COALESCABLE	EXPAND
u-date@0	maximal	m-date@1, DATE0
rm@1 + 7620	yes	room@1, LOC6
to + room@1	yes	stpair@1, STP1
seminar@0 + m-date@1	yes	seminar@1, GG2
the + seminar@1	yes	m-d-gg@1, DGG1
change + m-d-gg@1+stpair@1	no	

Coalesce/Expand Cycle at Deviation-level 2:

Subtrees	COALESCABLE	EXPAND
change + m-d-gg@1+stpair@1	yes	change@2, ACT3
change@2	maximal	*ROOT*

Figure 6-7: The incremental progress of a least-deviant-first search as created by the uniform effect of COALESCABLE's step (2) across Deviation-levels.

Also at Deviation-level 0 we see that although the pcs for **June** and **4** can coalesce and EXPAND via DATE1 to produce an unmarked date (u-date@0), the subtree representing that unmarked date cannot EXPAND because its position in the utterance demands that it be marked by a preposition. The subtrees for **rm** and **7620** cannot even coalesce at Deviation-level 0 because **rm** is represented by an unresolved pc and, therefore, has a *dlevel* of one (rm@1).

At Deviation-level 1, each of the previously failed paths is advanced: rm@1 and **7620**

JOIN and then successfully EXPAND using a substitution in LOC6. The resulting subtree is represented by room@1. Meanwhile the subtree for "June 7" as an unmarked date (u-date@0) becomes a marked date by compensating for the deletion in DATE0. Of course, each of the new subtrees has a *dlevel* of one because each new subtree contains an explained violated expectation. As long as the subtrees containing the deviant marked date and the deviant room do not try to coalesce, higher level constituents can be constructed at the current deviation-level (for example, seminar@1). To parse the sentence, however, the two subtrees must eventually be JOINed—COALESCABLE's step (2) blocks the action until the deviation-level is incremented because no less deviant explanation could be found. When the extra deviation point becomes available, the subtrees coalesce and a *ROOT* is produced.

To see how COALESCABLE's step (2) may interact with adaptation to make the understanding process more efficient, Figure 6-8 shows the algorithm's effect on search under a derived grammar. To the kernel we add a single derived form, DATE0', that recognizes marked dates in which the marker has been omitted. Such a form would have been created, for instance, after CHAMP confirmed its interpretation of the utterance in Figure 6-7 (the definition of "rm" as a **roomwd** would have been added to the lexicon as well, but we will ignore that to simplify the comparison).

The utterance in Figure 6-8 is structurally identical to the sentence in Figure 6-7. CHAMP uses DATE0' to find the explanation of the new sentence at Deviation-level 1 rather than Deviation-level 2. Although the subtree for "rm 7620" still has a *dlevel* of one, the subtree for "June 9" as a marked date has a *dlevel* of zero, via the derived form. Thus when step (2) of the COALESCABLE predicate is reached at Deviation-level 1, the two subtrees are permitted to JOIN. As a result, a *ROOT* node appears on the Agenda, and no further invocations of the Parse/Recovery Cycle need occur. COALESCABLE's judgment that the previously ungrammatical segment has a non-deviant interpretation is made possible by the existence of DATE0' which was constructed in response to the prior interaction.

Figures 6-7, 6-8, and 6-9 demonstrate that when two subtrees JOIN their *dlevels* are combined. Thus the implementation, like the model, contains only a simple notion of deviance: every recovery action compensates for the same "amount" of deviation and the amount of deviance along a path is additive. In Chapter 11 we discuss briefly the advantages of a more complex approach to measuring deviation.

COALESCABLE and JOIN comprise the Coalesce phase of the Coalesce/Expand Cycle, leaving the task of generalizing EXPAND to work at non-zero deviation-levels. Although the function in Figure 6-10 looks radically different from the one in Figure 5-17, there are only two important changes. The most important, of course, is that EXPAND now takes a context as an argument rather than a pc—the Coalesce/Expand

Utterance: "Change the 10 am class June 9 to *rm* 7620"
Derived Form: (DATE0'
isa m-dateforms
mode (203)
strategy (203: class u-dateforms))

Coalesce/Expand Cycle at Deviation-level 0:

Subtrees	COALESCABLE	EXPAND
10 + am	yes	u-hour@0, HR2
June + 9	yes	u-date@0, DATE1
rm@1 + 7620	no	
u-date@0	maximal	m-date@0, DATE0'
u-hour@0 + class + m-date@0	yes	class@0, GG3
the + class@0	yes	m-d-gg@0, DGG1

Coalesce/Expand Cycle at Deviation-level 1:

Subtrees	COALESCABLE	EXPAND
rm@1 + 7620	yes	room@1, LOC6
to + room@1	yes	stpair@1, STP1
change + m-d-gg@0 + stpair@1	yes	change@1, ACT3
change@1	maximal	*ROOT*

Figure 6-8: The uniform effect of COALESCABLE's step (2) over time (least-deviant-first search in a derived grammar).

JOIN (pc1 pc2)
IF pc1 is already a JOIN node (1)
THEN add pc2 to pc1's *subs* field
let pc1's *strats* field include only the active strategies parsing all the *subs*
add pc2's *dlevel* to pc1's *dlevel* **(1a)**
ELSE create a new JOIN node (2)
let its *subs* field contain pc1 and pc2 AND
let its *strats* field contain the active strategies that parse both subnodes
let its *dlevel* be the sum of the *dlevels* of the two nodes **(2a)**

Figure 6-9: CHAMP's JOIN Algorithm
(Figure 5-15 generalized to all deviation-levels).

algorithm does the actual expansion to and partitioning of the parent classes that used to be done in EXPAND. The second difference is that EXPAND now contains an explicit

gateway to the error detection and recovery mechanism in the function CLOSEOFF (step (1)). Notice that by creating unresolved pcs with a *dlevel* of one we guarantee that during error detection and recovery the number of unresolved pcs in a context cannot be greater than the number of deviations currently permitted along a path. The processing in CLOSEOFF and the range of values it can return are the topic of Section 6.5. Here we note only that through CLOSEOFF a context may be modified by the removal of unresolved subnodes, the elimination of strategies, or the addition of recovery notations. When control returns to EXPAND, the changes to the context are incorporated into the new pcs that are added to the Agenda. We look more closely at EXPAND's step (3) when we examine CLOSEOFF in detail (see Figure 6-23).

```
EXPAND (context)
CLOSEOFF the context                                                    (1)
IF CLOSEOFF returned failure                                            (2)
THEN RETURN
ELSE cache "true" for the context
Use the value returned by CLOSEOFF to modify the context and            (3)
  IF the subnodes satisfy all expand-time constraints for the context's class
  THEN create a pnode for the class that spans the subnodes
      IF there is an expand-time ig for the class
      THEN invoke it AND cache the canonical value for the pnode
      FOR EACH active step that seeks the pnode's class, DO             (4)
        IF the step has a bind-time constraint AND the pnode satisfies it
        THEN create a pc for the step, pnode, subnodes, and strategies in the
            modified context and place it on the appropriate level of the Agenda
```

Figure 6-10: CHAMP's EXPAND Algorithm
(Figure 5-17 generalized to all deviation-levels).

6.2. The Cache

CHAMP redoes very little work in increasingly deviant Coalesce/Expand Cycles because the system caches values for JOIN and EXPAND. The only node pairs offered to COALESCABLE each time through the Agenda are those that have not previously JOINed. In addition, every context that successfully EXPANDed in a prior cycle has a cached value of "true" (see step (2) in Figure 6-10). When that value is found, EXPAND need not be reinvoked for the context because all the EXPANDed nodes were placed on the Agenda during the previous cycle. Thus, when Deviation-level *i* is run, only previously unCOALESCABLE node pairs and paths that failed to EXPAND successfully at Deviation-level *i-1* actually produce new work. This is demonstrated by Figure 6-11 for the utterance

"Schedule meeting at 3 pm June 7"

which requires two deviation points in the kernel grammar: one for a deleted article and one for a deleted preposition. The figure shows only a small portion of the search space explored during parsing; a larger portion of the space will be examined in Figures 6-13 through 6-15 in the next section.

Each row of Figure 6-11 corresponds to a candidate constituent offered to EXPAND. The rows are divided into groups by increasing *dlevel* for the context. If a candidate constituent in the first column can be CLOSEdOFF without requiring more recovery actions that the current deviation-level allows, then the root node for the EXPANDed subtree is shown in the second column along with any recovery actions applied. If trying to CLOSEOFF the context uncovers an unacceptable degree of deviation, however, then the second column explains the error. Reading down the figure, the first context shows that **3** and **pm** coalesce at Deviation-level 0 to produce an unmarked hour via HR1. **At** then coalesces with the unmarked hour (at a higher Agenda-level, of course) to produce a marked hour. In the third context, **June** and **7** produce an unmarked date via DATE1, but the unmarked date cannot EXPAND to a marked date without its preposition. Since GG1 can take an unmarked date in the prenominal position, u-date@0 can coalesce with **meeting** and m-hour@0 but cannot EXPAND because of a transposition.

The middle section of Figure 6-11 demonstrates that the work done at Deviation-level 1 stems from the two paths that failed to EXPAND at Deviation-level 0. When the path containing u-date@0 is retried with one deviation permitted, the unmarked date becomes m-date@1 with the help of a deletion recovery action. In turn, the marked date contributes to the maximal set of subnodes at Deviation-level 1 that EXPANDs to meeting_1@1. Another path that failed to EXPAND at Deviation-level 0 interprets "meeting at 3 pm June 7" as a maximal subsegment containing an unmarked date. This path can also succeed at Deviation-level 1; in meeting_2@1, the deviation point is used to explain the transposition.[29] Both meeting_1 and meeting_2 fail to EXPAND to marked **groupgatherings** because the utterance is missing the article and no deviation points are left to compensate for the deletion.

Since no *ROOT* appears on the Agenda during the Parse/Recovery cycle for Deviation-level 1, the deviation-level is incremented to two. Again the only work done stems from paths that failed: each of meeting_1@1 and meeting_2@1 is allowed to accrue the additional deviation point required to omit the article. The grammar allows two possibilities in the expansion of each **meetingform**—it may be either the definite or the

[29]Actually, the kernel grammar contains a **no-trans** recovery constraint that prevents steps seeking prenominal modifiers from being transposed with respect to the step seeking the head noun. A similar constraint applies to postnominal modifiers. For demonstrative purposes we assume in this example that the **no-trans** constraint was not placed in the grammar.

Utterance: "Schedule [a] meeting at 3 pm [on] June 7"

Deviation-level 0:

Context	Result of EXPAND
[(3 pm) (HR1)]	u-hour@0
[(at u-hour@0) (HR0)]	m-hour@0
[(June 7) (DATE1)]	u-date@0
[(u-date@0] (DATE0)]	fails due to missing marker
[(meeting m-hour@0 u-date@0) (GG1)]	fails by transpose of u-date

Deviation-level 1:

Context	Result of EXPAND
[(u-date@0) (DATE0)]	m-date@1, DATE0 + deletion
[(meeting m-hour@0 m-date@1) (GG1)]	$meeting_1$@1, GG1
[(meeting m-hour@0 u-date@0) (GG1)]	$meeting_2$@1, GG1 + transpose
[($meeting_1$@1) (IGG1)]	fails by missing indefinite article
[($meeting_1$@1) (DGG1)]	fails by missing definite article
[($meeting_2$@1) (IGG1)]	fails by missing indefinite article
[($meeting_2$@1) (DGG1)]	fails by missing definite article

Deviation-level 2:

Context	Result of EXPAND
[($meeting_1$@1) (IGG1)]	m-i-gg_1@2, IGG1 + deletion
[($meeting_1$@1) (DGG1)]	m-d-gg_1@2, DGG1 + deletion
[($meeting_2$@1) (IGG1)]	m-i-gg_2@2, IGG1 + deletion
[($meeting_2$@1) (DGG1)]	m-d-gg_2@2, DGG1 + deletion
[(schedule m-i-gg_1@2) (ACT1)]	add_1@2
[(schedule m-i-gg_2@2) (ACT1)]	add_2@2
[(add_1@2) (*ROOT*)]	succeeds
[(add_2@2) (*ROOT*)]	succeeds

Figure 6-11: Extending the search space by EXPANDing only contexts that failed at the previous deviation-level.

indefinite article that has been deleted. Because the **add** action expects an object marked by an indefinite article, only the **m-i-ggforms** may coalesce with the pc for "schedule," leading to two *ROOT* nodes each with a *dlevel* of two. The first *ROOT* uses its two deviation points to explain the sentence with a deletion of the date marker and a deletion of the definite article. The second *ROOT* uses its two deviation points to explain the sentence with a deletion of the article and a transposition of the unmarked date.

Turning to the cache before recomputing the value of EXPAND has an important consequence: if a context was successfully CLOSEdOFF at a deviation-level less than *i* then no more deviance is allowed to accrue to that path. In other words, as the space of deviance hypotheses grows it does not grow uniformly. To see why this is a desirable consequence, consider that if a path EXPANDs at Deviation-level *i-1* but no complete parse can be found at that deviation-level then the utterance must require additional recovery actions to be understood. Using the additional deviation points on a path we have already explained cannot make the global parse successful; the partitioning of strategies into contexts that was done in the Coalesce/Expand algorithm guarantees that the additional recovery actions will change neither the size of the segment accounted for nor the formclass to which the segment will be EXPANDed. If neither the size of the segment nor its assigned class changes then no new paths have been created. Instead, the higher level constituents produced by EXPAND for the context will simply be more deviant versions of the paths produced at the lower deviation-level. Since no new paths have been created, the global parse must still fail. Thus, for the global parse to succeed the additional deviation points must have been needed by some other context—a context that failed to EXPAND in a prior cycle. These are precisely the contexts for which the cached value of EXPAND is not "true."

At Deviation-level *i*, the generalized parsing functions find sets of co-constituents that have together accrued no more than *i* deviation points. Before the contexts are converted from virtual to actual higher grammatical constituents, CHAMP's cache prevents redundant and provably useless work by allowing the system to weed out those contexts for which less deviant explanations already exist. The remaining contexts must be shown to require no more than a total of *i* deviation points before they can be EXPANDed. Determining the degree of deviance represented by a context is the responsibility of Error Detection, the portion of the system to which we now turn our attention.

6.3. Error Detection

Throughout the Parse/Recovery Cycle our strongest search constraint is the detection of violated expectations in the interpretation of a segment as a particular grammatical constituent. Even as we expand the search space to encompass larger and larger sets of deviance hypotheses, the boundary of the space during any given cycle remains rigidly defined by the errors we can explain at the current deviation-level. We have already seen that the violation of a Concept Hierarchy constraint or a condition in the COALESCABLE predicate is adequate to terminate a path at any deviation-level—these errors are *non-recoverable*. Because we want to detect non-recoverable errors as soon as possible, the responsibility for their detection is spread throughout the system. In contrast, the responsibility for detecting *recoverable* errors in CHAMP rests within the Error Detection & Recovery mechanism shown in Figure 6-1, and primarily within the function ISCOMPLETE (Figure 6-12).

```
ISCOMPLETE (context)
Separate the unresolved nodes (urs) from the other subnodes in the context      (1)
FOR EACH strategy in the context, DO                                            (2)
  Find the missing, required steps (mls)                                        (2a)
  Find the minimum number of transposed steps (tls)                             (2b)
  IF there are tls AND they violate a no-trans constraint OR                    (2c)
    there are mls AND no urs AND the mls violate a no-del constraint
  THEN go on to the next strategy
  Compute the amount of deviance for the subcontext where                       (2d)
    aod = the dlevel of the context +
          the number of tls +
          max (0, (the number of mls - the number of urs))
  Assign the subcontext to a partition according to its aod
IF the smallest aod <= the Deviation-level                                      (3)
THEN RETURN (stratname mls tls) FOR EACH form in the partition                  (3a)
ELSE RETURN (Retry the lowest aod)                                              (3b)
```

Figure 6-12: CHAMP's ISCOMPLETE predicate for general error detection.

Recall that the *dlevel* of a context is defined to be the sum of the *dlevels* of its subnodes. Expressed in another way, a context is a candidate grammatical constituent that may have already accrued some amount of deviance during the explanation of its sub-constituents. ISCOMPLETE determines whether a particular context can be explained without accumulating a total deviance greater than that allowed by the current deviation-level. Note that the function does not explain any errors; it merely determines whether (and how much of) the context should be passed to Error Recovery. Specifically, ISCOMPLETE tries to pinpoint which *subcontexts* contain the fewest additional violated expectations. A *subcontext* is defined as a pairing of the subnodes in the context with a particular strategy in the context. For example, if both the **u-hourforms** HR1 and HR2 are strategies in a context for pc*x* and pc*y* then each of [(pc*x* pc*y*) (HR1)] and [(pc*x* pc*y*) (HR2)] is a subcontext. By finding the minimally deviant explanation for each context, we guarantee that the least-deviant global explanation will be produced first.

From the model in Chapter 2 we know that there are four types of recoverable errors: insertion, deletion, substitution and transposition. In ISCOMPLETE these error types correspond to the values computed for particular variables for a given subcontext. Insertions and substitutions are signalled by the presence of unresolved nodes (**urs**, step (1)), transpositions by transposed steps (**tls**, step (2b)), and deletions by missing, required steps (**mls**, step (2a)). To find the amount of deviance (**aod**) in a subcontext we collect the *step* fields of the subnodes into a list then compare that list with the strategy list of the

subcontext's strategy. Using the strategy's annotation nodes, the comparison allows us to compute the number of **mls** and **tls**. The amount of deviance for the context is then computed according to the formula in step (2d).

Note that the presence or absence of unresolved segments has no effect on the number of **tls**, but can effect whether an **ml** is considered an additional source of deviation or not. If a **ur** is present then it may fill in for the missing step; under these circumstances we do not need to add a deviation point because the projected substitution uses the deviation point made available in the unresolved subnode when it was created with a *dlevel* of one. In other words, if the unknown phrase is to be used as a synonym for a required lexeme then only one expectation has been violated, not two.[30] If there are no unresolved segments, or the number of missing, required steps is greater than the number of nodes available to fill them, then we count one deviation point for every required but unsatisfied step. Observe, however, that ISCOMPLETE neither makes the substitution nor decides whether the substituted step will occur in a permissible order. At the time we compute the amount of deviance in a subcontext, we do not know if the current strategy will be among those in the least-deviant subset or not, so we leave the additional error detection to the recovery action that performs the substitution.

Once all of the strategies in the context have been partitioned by the amount of deviation they add to the subtree, the lowest-valued set is chosen. If the value is less than or equal to the current deviation-level, ISCOMPLETE returns a triple for each strategy in the set (step (3a)), giving Error Recovery a head start on the information it will need to choose an appropriate recovery action. If, on the other hand, the best explanation that could be offered for the context has an **aod** greater than the current deviation-level, ISCOMPLETE returns a "retry message" that tells recovery the lowest deviation-level required for a subcontext to succeed (step (3b)). Since the cached value for EXPAND is intended to reflect the success of error detection and recovery, we cache "true" if one or more subcontexts are returned, and the retry message otherwise. In this way we retry EXPANDing a failed path only when enough deviation points are available for it to succeed.

[30]Indeed, not using the unresolved segment as a substitution for a missing step would produce a more deviant explanation, requiring one point for the **ur** as an insertion or substitution elsewhere, and a separate point for the current **ml** as a deletion.

6.4. A Detailed Example: Error Detection during the Parse of "Schedule a meeting at 3 pm June 7"

Let us now reexamine the search space for the utterance "Schedule meeting at 3 pm June 7" in light of what we know about error detection. Figures 6-13, 6-14, and 6-15 demonstrate how the least-deviant-first search progresses. Each figure shows a subset of the contexts considered at a particular deviation-level. For each context the results of the calls to ISCOMPLETE (ISC) and EXPAND (EXP) are displayed along with the value cached. As in Figure 6-11, when a context fails to EXPAND, the reason for the failure is given in the EXPAND column.

Context	ISC (strat mls tls)	EXP/cache
[(3 pm) (HR1 HR2)]	(HR1 – –)	u-hour@0/t
[(June 7) (DATE1)]	(DATE1 – –)	u-date@0/t
[(at u-hour@0) (HR0)]	(HR0 – –)	m-hour@0/t
[(u-date@0) (DATE0)]	(Retry 1)	missing marker/1
[(meeting m-hour@0 u-date@0) (GG1)]	(Retry 1)	transposed u-date/1

Figure 6-13: Error detection at Deviation-level 0 for "Schedule meeting at 3 pm June 7"

Figure 6-13 shows how the paths from Deviation-level 0 of Figure 6-11 failed. In the fourth row the unmarked date is proposed as a candidate marked date using DATE0. Because of the missing preposition, ISCOMPLETE finds one **ml** in the subcontext. Since the deviation-level is zero, the violated expectation cannot be tolerated and ISCOMPLETE translates the single deviation into a message to retry the context at Deviation-level 1. In row five, the same unmarked date is proposed to ISCOMPLETE as a postnominal modifier in GG1 (which expects unmarked cases to occur prenominally). This time ISCOMPLETE detects one **tl** and, again, a retry message is returned.

Figure 6-13 also gives us the opportunity to point out that when more than one strategy is included in a context, error detection's ability to control search may be augmented by *critical differences* between the strategies. In Chapter 5 we defined a

- **Critical Difference**: a form, X, contains a critical difference with respect to another form, Y, if X and Y contain the same steps with different ordering constraints, or if X and Y differ by at least one step.

Under a strict interpretation of the definition, a critical difference exists between strategies independent of any utterance. The limiting effect of a critical difference on search, however, is manifested only if the utterance contains the critical constituents. An utterance causes the termination of one or more subcontexts via a critical difference if:

1. The utterance satisfies the ordering constraints in some of the strategies but not in others (**ordering**); or

2. The utterance contains a constituent sought by some of the strategies but not by others (**presence**); or

3. The utterance does not contain a constituent required by some of the strategies but not by others (**absence**).

We saw an example of the second type of termination in Section 5.2; a JOINed node representing prenominal modifiers carried strategies for the classes **meetingforms** and **mealforms** in its *strats* field until the word "meeting" was encountered. The word "meeting" manifested the critical differences between the *strats* because only GG1 seeks a **meetingwd**. As a result of the critical difference, the JOINed node built to encompass "meeting" had only GG1 in its *strats* field. Thus, the **presence** of a distinguishing constituent is detected by COALESCABLE, and the winnowing of the search space accomplished by JOIN. The justification for reducing the context is the design decision to EXPAND only maximally coalescable sequences; since some strategies can account for the extra token, those subcontexts continue while subcontexts represented by strategies that cannot account for the token are terminated.

Like COALESCABLE, ISCOMPLETE may also reduce the context when the strategies contain critical differences. The justification in ISCOMPLETE, however, is the need to advance only the least-deviant subcontexts. A difference in **ordering** constraints between two forms translates into **tls** for any utterance that contains the relevant constituents. Since every transposition requires a recovery action, a subcontext that does not need a transposition recovery to explain the segment must be preferred over a subcontext that does.

Mls play a role in the **absence** condition similar to the role played by **tls** in the **ordering** condition. Consider as an example the critical difference between HR1 and HR2 in row one of Figure 6-13. HR1 recognizes times of the form "number [clockwd | daywd | nightwd]," while HR2 recognizes "number colonsymb number [clockwd | daywd | nightwd]." In the subcontext for HR2 and "3 pm," ISCOMPLETE detects the absence of the required constituents (colon and minutes) as two **mls**. Since HR1 requires no recovery actions to explain the segment and HR2 requires two deletions, only HR1 remains in the context at the end of error detection.

Critical differences also play a role in parsing at Deviation-level 1 for our sample sentence. The first row of Figure 6-14 shows how the path for the marked date that failed previously succeeds at Deviation-level 1 by using the available deviation point for the deletion. In turn, meeting_1 is created by coalescing the just created m-date@1 with the pcs for "meeting" and "at 3 pm" that still reside on the Agenda. Since the step picking up the marked date is included in GG1's *snode*, meeting_{1a} is also created a this time (row 3). Note that the subtree represented by meeting_{1a} has a *dlevel* of zero because the constituent that contributed deviance to the context has been removed in response to the *snode*. Along a different path, meeting_2 is created using the available deviation point to

account for the transposition of step 303 (the unmarked date) detected by ISCOMPLETE. In the next six rows, CHAMP tries to EXPAND each of the three **meetingforms** using IGG1 and DGG1—two kernel forms distinguished by a critical difference. IGG1 has required steps seeking an indefinite article and object, while DGG1 seeks a definite article and object. In Figure 6-14, as in Figure 6-11, both contexts fail to EXPAND for $meeting_1$ and $meeting_2$ because neither article is present in the utterance. Both paths succeed for $meeting_{1a}$, however, because that subtree can use the available deviation point to compensate for the **ml** found in each context by ISCOMPLETE. Both contexts succeed because the user's deviation negates the critical difference between the strategies.

Although the critical difference between IGG1 and DGG1 is not manifested by the utterance, the critical difference between ACT1 and ACT2 is (last two rows of Figure 6-14). Because the **addform** ACT1 requires an object marked by an indefinite article, only m-i-gg_{1a} can coalesce with **schedule**. This path fails to become a *ROOT* node, however, because it does not account for the entire utterance (only "schedule meeting at 3 p.m."), a non-recoverable error. Notice that the cached value for EXPAND under these circumstances is "true" so that the path will never be retried. The **deleteform** (ACT2) requires a **deletewd** and an object marked by a definite article. Since m-d-gg_{1a} is available but an appropriate verb is not, the global failure of the path for ACT2 is delayed until Deviation-level 2 (see row 7 of Figure 6-15).

Context	ISC (strat mls tls)	EXP/cache
[(u-date@0) (DATE0)]	(DATE0 201 –)	m-date@1/t
[(meeting m-hour@0 m-date@1) (GG1)]	(GG1 – –)	$meeting_1$@1/t
[(meeting m-hour@0) (GG1)]	(GG1 – –)	$meeting_{1a}$@1/t
[(meeting m-hour@0 u-date@0) (GG1)]	(GG1 – 303)	$meeting_2$@1/t
[($meeting_1$@1) (IGG1)]	(Retry 2)	missing indefinite article/2
[($meeting_1$@1) (DGG1)]	(Retry 2)	missing definite article/2
[($meeting_2$@1) (IGG1)]	(Retry 2)	missing indefinite article/2
[($meeting_2$@1) (DGG1)]	(Retry 2)	missing definite article/2
[($meeting_{1a}$@0 (IGG1)]	(IGG1 402 –)	m-i-gg_{1a}@1/t
[($meeting_{1a}$@0 (DGG1)]	(DGG1 401 –)	m-d-gg_{1a}@1/t
[(m-d-gg_{1a}@1) (ACT2)]	(Retry 2)	missing verb/2
[(schedule m-i-gg_{1a}@1) (ACT1)]	(ACT1 – –)	incomplete *ROOT*/t

Figure 6-14: Error detection at Deviation-level 1 for "Schedule meeting at 3 pm June 7"

Unlike $meeting_{1a}$, $meeting_1$ and $meeting_2$ each have a *dlevel* of one. Both **meetingforms** explain the segment, "meeting at 3 pm June 7." Because this is a

different segment than "meeting at 3 pm," (explained by $meeting_{1a}$), $meeting_1$ and $meeting_2$ represent paths distinct from $meeting_{1a}$. In addition, even though $meeting_1$ and $meeting_2$ explain the same segment, they nonetheless represent different subtrees by incorporating different explanations. As different subtrees they are represented by different root pcs which in turn may coalesce with other subnodes to form distinct contexts. Thus, the failure of both $meeting_1$ and $meeting_2$ to EXPAND to either **m-i-ggforms** or **m-d-ggforms** creates four paths to be retried at Deviation-level 2.

The fate of those four paths can be seen in Figure 6-15. Each context reenters ISCOMPLETE with a *dlevel* of one and uses the extra deviation point available at Deviation-level 2 to recover from its missing, required step. Again, the critical difference between IGG1 and DGG1 is not manifested by the utterance, but the critical difference between ACT1 and ACT2 is: only the **m-i-ggforms** can coalesce with **schedule**, while the absence of a **deletewd** requires an additional recovery action for the **m-d-ggforms** that have been produced. Thus, each of m-i-gg_1 and m-i-gg_2 produces a valid **addform** that accounts for all the tokens in the utterance. The complete **addforms** become *ROOT* nodes for two APTs. Observe that the context [(schedule m-i-gg_{1a}) (ACT1)] that failed at Deviation-level 1 is *not* retried at Deviation-level 2—the cached value of "true" for EXPAND prevents re-exploring a context with a non-recoverable error.

Context	ISC (strat mls tls)	EXP/cache
[($meeting_1$@1) (IGG1)]	(IGG1 402 –)	m-i-gg_1@2/t
[($meeting_1$@1) (DGG1)]	(DGG1 401 –)	m-d-gg_1@2/t
[($meeting_2$@1) (IGG1)]	(IGG1 402 –)	m-i-gg_2@2/t
[($meeting_2$@1) (DGG1)]	(DGG2 401 –)	m-d-gg_2@2/t
[(schedule m-i-gg_1@2) (ACT1)]	(ACT1 – –)	add_1@2/t (*ROOT*)
[(schedule m-i-gg_2@2, (ACT1)	(ACT1 – –)	add_2@2/t (*ROOT*)
[(m-d-gg_{1a}@1) (ACT2)]	(ACT2 705 –)	incomplete *ROOT*/t
[(m-d-gg_1@2) (ACT2)]	(Retry 3)	retry level > maxdev/3
[(m-d-gg_2@2) (ACT2)]	(Retry 3)	retry level > maxdev/3

Figure 6-15: Error detection at Deviation-level 2 for "Schedule meeting at 3 pm June 7"

When a *ROOT* has a *dlevel* greater than zero we call it a hypothetical explanation of the utterance or, more simply, an hypothesis. Although we have found hypotheses at Deviation-level 2 for the example, ISCOMPLETE is clearly a general mechanism capable of performing error detection at higher deviation-levels. Unfortunately, the greater the amount of deviance required to explain an utterance, the greater the chance of constructing multiple hypotheses that differ only in terms of their recovery notations. The correlation occurs because recovery is the process by which we relax the very

expectations that distinguish grammatical forms. Figure 6-16 shows that the annotated parse trees built by CHAMP for the example are a case in point.[31]

The figure also gives us the opportunity to distinguish between the actual meaning of an utterance and its

- **Effective meaning**: an interpretation of the utterance in terms of the effect it produces in the task domain. It is possible for sentences that are effectively equivalent to differ in actual meaning (because the difference in meaning cannot be reflected by the actions available to the system). CHAMP, like other natural language interfaces, builds structures to capture an utterance's effective meaning; this reflects the assumption that the system's primary goal is to perform the action intended by the user.

The hypotheses in pc62 and pc63 capture the same effective meaning but differ in their explanations of the user's errors. Pc62 explains the utterance using a deletion of the indefinite article usually sought by step 402 of IGG1 (see the *recovery notation* in pc54) with the transposition of the step seeking an unmarked date in GG1 (see the recovery notation in pc42). Pc63, on the other hand, combines the deletion of the indefinite marker (pc55) with the deletion of the date marker usually required by step 201 of DATE0 (notated at pc35).

After the error detection performed by ISCOMPLETE but before the creation of higher level constituents in EXPAND, a deviant context must be explained. In the remainder of this chapter we examine how that explanation is constructed: where recovery notations come from and how they are incorporated into the APT. We postpone discussion of how to choose among competing hypotheses like those represented by pc62 and pc63 to Chapter 7. How we convert an APT into a set of grammatical components that recognize a deviant form directly is a discussion left to Chapter 8.

6.5. Error Recovery

The model in Chapter 2 gave us four classes of errors: deletion, insertion, substitution and transposition. In the previous section we implemented error detection by creating a fairly simple correspondence between each class and the values computed for the variables **urs**, **tls**, and **mls** within a subcontext. To compensate for an error, the model also gives us four general recovery actions: insertion, deletion, substitution and transposition. Unfortunately, the correspondence between recovery action and the values of implementation variables is more complex than it was for error detection. The difficulties arise from the need to embed a recovery notation at the correct position in an

[31]The format for displaying a pc was introduced in Section 5.1. After the pc identifier comes the step followed by the pnode (the segment's start position, end position, and class). When non-nil the *strats* field and *dlevel* fields come next. Subnodes are indicated by indentation.

```
62: *ROOT* (0 6 addforms) (ACT1) 2
  0: 704 (0 0 addwd)
  54: 708 (1 6 m-i-ggforms) (IGG1) (2 (DELETE 402))
    42: 403 (1 6 meetingforms) (GG1) (1 (TRANSPOSE pc27))
      1: 309 (1 1 meetingwd)
      24: 315 (2 4 m-hourforms) (HR0)
        2: 111 (2 2 hrmkr)
        14: 113 (3 4 u-hourforms) (HR1)
          4: 4 (3 3 number)
          7: 16 (4 4 nightwd)
      27: 303 (5 6 u-dateforms) (DATE1)
        8: 101 (5 5 monthwd)
        11: 102 (6 6 number)

63: *ROOT* (0 6 addforms) (ACT1) 2
  0: 704 (0 0 addwd)
  55: 708 (1 6 m-i-ggforms) (IGG1) (2 (DELETE 402))
    40: 403 (1 6 meetingforms) (GG1) 1
      1: 309 (1 1 meetingwd)
      24: 315 (2 4 m-hourforms) (HR0)
        2: 111 (2 2 hrmkr)
        14: 113 (3 4 u-hourforms) (HR1)
          4: 4 (3 3 number)
          7: 16 (4 4 nightwd)
      35: 316 (5 6 m-dateforms) (DATE0) (1 (DELETE 201))
        26: 203 (5 6 u-dateforms) (DATE1)
          8: 101 (5 5 monthwd)
          11: 102 (6 6 number)
```

Figure 6-16: The Annotated Parse Trees (APTs) constructed under CHAMP's kernel for "Schedule meeting at 3 pm June 7."

APT that may be only partially constructed. As we examine the implementation of each of the four general types of recovery we will see that, in CHAMP, choosing the appropriate recovery action for a subcontext depends upon six factors:

1. The number of unresolved subnodes (**urs**).
2. The number of missing, required steps identified by ISCOMPLETE (**mls**).
3. The number of transpositions independent of substitutions identified by ISCOMPLETE (**tls**).
4. The number of additional transpositions if a substitution is made (**atls**).
5. The number of deviation points available (**dpa**) as calculated by: the

Deviation-level - the *dlevel* of the subcontext + the number of **urs**.[32]

6. The presence of recovery-time constraints (**rtc**).

We know from the EXPAND algorithm in Figure 6-10 that the gateway to error detection and recovery is the function CLOSEOFF. CLOSEOFF's first step is to call ISCOMPLETE for a preliminary assessment of the violated expectations in the context. The purpose of error recovery is to choose for each subcontext that ISCOMPLETE returns the recovery action that promotes the least-deviant global parse. As a result, one way to view CLOSEOFF is as a large discrimination net. At the leaves of the net are one or more of CHAMP's six general recovery functions: OK, RETRY, DELETE, TRANSPOSE, INSERT, and SUBSTITUTE. In the model, the general recovery action for an insertion error is a deletion and for a deletion error, an insertion. The recovery notation left in an APT, however, is the description of the user error required by the adaptation mechanism. To avoid confusion when reading APTs, we pair the names of the recovery functions with the errors they detect and the notations they produce. Thus, INSERT recovers from insertion errors and leaves an insertion notation, while DELETE recovers from deletion errors and leaves a deletion notation. Because the discrimination logic in CLOSEOFF is a function of the six factors listed above, our discussion of error recovery will be simpler if we proceed "bottom-up," describing the leaf functions first and the root function last. Thus, once we have introduced the individual recovery functions we will conclude the chapter by presenting the CLOSEOFF algorithm and demonstrating how recovery notations are incorporated into an APT.

OK and RETRY represent the extreme conditions that may occur during error detection: no violated expectations and too many violated expectations. OK is the function used when no error recovery is required. If the context given to CLOSEOFF contains no unresolved nodes, and the subcontexts returned by ISCOMPLETE contain no missing or transposed steps, then CLOSEOFF simply returns a list of strategy names taken from the preserved subcontexts. In columns labelled "ISC" in Figures 6-13, 6-14, and 6-15 subcontexts that are OK look like: (strategyname – –).

At the opposite end of the spectrum, CLOSEOFF may use the function RETRY. When ISCOMPLETE returns a retry message because every subcontext required more additional deviation points than were available, CLOSEOFF passes the retry value back

---

[32]A **ur** carries its deviation point in order to guarantee that during error detection and recovery the number of unresolved pcs in the context is no greater than the number of deviations currently permitted along a path. In other words, a *dlevel* of one is given to a **ur** to reserve for that node the minimum number of deviation points required to explain it. When the deviation-level is high enough for the **ur** to coalesce with other nodes, the reserved deviation point must be contributed to the total number of deviation points available in order to offset the increase in the *dlevel* of the context caused by the **ur**. In this way, a context with one **ur** and a *dlevel* of one, created at Deviation-level 1, nonetheless has one deviation point available for recovery (1 - 1 + 1).

to EXPAND. CLOSEOFF may also produce a retry message on its own—as when, for example, a substitution creates an added transposition but only one deviation point is available. Regardless of which function detects the retry condition, we have seen that the cache remembers the retry value to keep CHAMP from reEXPANDing a failed path before the deviation-level is high enough for the path to progress.

When error recovery may be able to explain a context, ISCOMPLETE (step (3a)) returns a list in which each least-deviant subcontext is represented by a triple: the strategy name, a value for **mls**, and a value for **tls**. If the context is free of unresolved nodes but the subcontext has missing, required steps, CLOSEOFF will respond by invoking DELETE. Similarly, if there are no unresolved nodes but **tls** have been detected, CLOSEOFF creates a recovery notation using TRANSPOSE. Note that under either set of circumstances ISCOMPLETE will have already checked for the appropriate type of recovery-time constraints (step (2c)). We have seen examples of the recovery notations produced by each of DELETE and TRANSPOSE in the APT rooted at pc62 in Figure 6-16.

The logic that discriminates between the remaining two recovery functions presupposes that the context passed to CLOSEOFF contains one or more unresolved subnodes. The complexity of response entailed by this one factor can be seen from the example in Figure 6-17. The figure shows a sentence taken from User 1's log file for the experiments described in Chapter 3.[33] The sentence is displayed in string form, as the user typed it, as well as in CHAMP's tokenized form (see Section 5.1).

During segmentation the unknown token **th** will not be resolvable as a new instance of an extendable class. Hence, during the Parse/Recovery Cycle the token will be represented by an unresolved pc with a *dlevel* of one. The first time CLOSEOFF encounters the node will be at Deviation-level 1 (otherwise the pc could not have coalesced) and the context will represent the segment "June 12th" as a candidate **u-dateform**. The presence of the unresolved node makes the *dlevel* of the context one, but also contributes the one deviation point available during recovery. Using the strategy list for DATE1 and the formula from step (2d), ISCOMPLETE finds that DATE1 requires no more than the one deviation point available and returns the subcontext (DATE1 – –). With no missing, required step in the returned subcontext, CLOSEOFF has three alternatives for interpreting the unresolved node:

---

[33]Examples taken from user data will always be preceded by a label of the form U*i*S*jk*, where *i* is the user's number, *j* is the session number (which ranges from 1 to 9), and *k* is the number of the sentence within the log file for the session. User numbers 1, 2, 3, 4, 5, and 7 correspond to data for the users in the two adaptive conditions of the hidden-operator experiments described in Chapter 3. User numbers 9 and 10 correspond to data for two new users performing the original experimental task in interactions with CHAMP. The complete list of sentences from all users participating in the experiments is given in Appendix B.

U1S39: "Cancel June 12th meeting at AISys"
Tokens: (cancel june 12 th meeting at aisys)

DATE1
 isa u-dateforms
 mode (101 102)
 strategy
  101: class monthwd bindvar month
  102: class number bindvar day
  103: class commasymb
  105: class number bindvar year

Deviation-level 1:
Context: [(june 12 th@1) (DATE1)]
ISCOMPLETE: aod = *dlevel* + tls + max(0, mls - urs)
 aod(DATE1) = 1 + 0 + 0
 returns (DATE1 – –)

Figure 6-17: The first context in which the token **th** is available to recovery.

1. Treat **th** as an insertion in this constituent.
2. Treat **th** as a substitution for a non-required link.
3. Assume **th** is needed by a higher level constituent.

It is not difficult to imagine a segment for which each of these alternatives is the appropriate action following "June 12": the current value of "th," replacing "th" with "nineteen eighty-eight," and replacing "th" with "business" provide justifications for (1), (2), and (3), respectively. Note, however, that the first and second alternatives create almost the same constituents in terms of the global parse; they consume the same amount of input and EXPAND to the same formclass. The difference between (1) and (2) is in their ability to capture the actual meaning of the sentence. Choosing the second alternative may allow us to learn the meaning of the token, but it does so at the cost of additional interactions with the user. Choosing the first alternative requires no additional effort from the user but is likely to produce only the effectively correct meaning. In addition, choosing either of the first two alternatives precludes the possibility of the third—we are not willing to create a distinct path for each interpretation and experience the exponential blow-up caused by the creation of three paths at each level of constituent in the grammar.

The solution implemented in CHAMP is justified by the goal of finding the least-deviant effective meaning of an utterance. Because we are concerned with finding only the effective meaning, we ignore the possibility of substituting the unresolved token for an unrequired step. If we discover a missing, *required* step during error detection, we try to satisfy it using the unknown phrase and the recovery function SUBSTITUTE. If there are no **mls**, however, then the unknown phrase is explained using INSERT if it is

embedded, or is passed back, if it is not embedded. In this way, if the unknown phrase *is* required by a higher level constituent, we will have made it available and a path that would have failed at the current deviation-level will succeed using substitution. If the unknown phrase is *not* required anywhere, it will continue to "bubble up" until it can be captured within a constituent as an insertion.

Figure 6-18 shows the fate of **th** using INSERT. As in Figure 6-17, the **ur** is allowed to coalesce with **june** and **12** at Deviation-level 1. Since the subcontext contains no missing, required steps and **th** is an end subsegment, CLOSEOFF must return the pc for **th** so it can be made available to higher level constituents. The figure shows that when the value returned by CLOSEOFF is not a retry level it is a list of modified contexts. The first position in each modified context is filled with the name of the strategy (or strategies) that succeeded via the modifications. The second position is used to pass back **urs** representing end subsegments that were not needed by the context to explain the user's error. The third position indicates the number of additional deviation points charged to the context and the recovery notation that should be included in the EXPANDed nodes. At Agenda-level 1 in our example, error recovery returns DATE1 as the successful strategy with the modification that the **ur** representing "th" be removed from the context. When EXPAND reduces a context it removes one deviation point for every **ur** removed (the **ur** takes its point away with it).

U1S39: (cancel june 12 th meeting at aisys)

Coalesce/Expand Agenda-level 1:
EXPAND Context: [(june 12 th@1) (DATE1)], *dlevel* = 1
Error Detection returns: (DATE1 – –)
Error Recovery returns: ([DATE1 (th) –])

Coalesce/Expand Agenda-level 2:
th@1 moved to next Agenda-level (step (1b) of the Coalesce/Expand Cycle)

Coalesce/Expand Agenda-level 3:
EXPAND Context: [(u-date@0 th@1 meeting m-location@0) (GG1)], *dlevel* = 1
Error Detection returns: (GG1 – –)
Error Recovery returns: ([GG1 – (0 (INSERT th))]) via INSERT

Figure 6-18: Bubbling **th** up the Agenda until an insertion is detected during the parse of "Cancel June 12th meeting at AISys" at Deviation-level 1.

At Agenda-level 2 the pc for "th" is discontiguous with the other nodes. Since it cannot coalesce it is simply moved up to Agenda-level 3 where it JOINs with other constituents as a candidate **meetingform**. When **th** JOINs with its co-constituents it raises the *dlevel* of the context to one. CLOSEOFF receives the context and passes it to

ISCOMPLETE which detects no additional violated expectations. Without a missing, required step, error recovery compensates for the unknown and now embedded subsegment using INSERT. Thus, the modified context created by error recovery has no **urs** to pass back and the deviation point made available by **th** is consumed by the insertion. The recovery notation in the modified context will be included in the *dlevel* field of every pc created as a **meetingform** in the context.

The utterance in the example occurs in User 1's protocol after CHAMP has learned that she expresses **m-d-ggforms** without the definite article. Using the derived form, DGG1'[34], the system produces the APT in Figure 6-19 at Deviation-level 1. Although the APT in the figure explains the unknown segment as an insertion in GG1, we will see in Chapter 8 that CHAMP does not truly take such a narrow view—adaptation and generalization of the APT rooted at pc56 will create a grammatical component that treats "th" as an insertion no matter where it appears in the future.

```
56: *ROOT* (0 6 deleteforms) (ACT2) 1
  1: 705 (0 0 deletewd)
  49: 709 (1 6 m-d-ggforms) (DGG1') 1
    44: 403 (1 6 meetingforms) (GG1) (1 (INSERT pc0))
      16: 303 (1 2 u-dateforms) (DATE1)
        2: 101 (1 1 monthwd)
        5: 102 (2 2 number)
      0: 0 (3 3 unknown) nil 1
      6: 309 (4 4 meetingwd)
      22: 317 (5 6 m-locationforms) (LOC0)
        8: 211 (5 5 locmkr)
        19: 212 (6 6 u-locationforms) (IFLOC2)
          9: 114 (6 6 businessname)
```

Figure 6-19: The APT constructed by CHAMP using an adapted grammar containing DGG1' for "Cancel June 12th meeting at AISys"

The final recovery function left to describe is SUBSTITUTE. It is invoked when there are both missing, required steps and available unresolved segments to fill them. The function may be invoked, however, only if the missing, required step is not precluded from a substitution by a recovery-time **no-sub** constraint, and if the substitution itself does not create an additional transposition (**atl**). Consider the situation in Figure 6-20 as an example. In the travel domain grammar, CHAMP expects only the word "departing" or the phrase "departing from" as markers for the source location case in a flight object.

[34]CHAMP generates non-mnemonic internal symbols for the names of user forms. For the convenience of the reader those symbols have been replaced in the example APTs by names that show the origin of the derived form.

User 4 consistently preferred the lexeme "leaving." Given the sentence in the example and the kernel grammar for the travel domain, "leaving" will appear as an unknown that will not be resolved as a new instance of an extendable class. At Deviation-level 0 **Chicago** will expand to an unmarked location but will be unable to either coalesce with **leaving** or EXPAND to a marked source location. At Deviation-level 1 EXPAND is called with a context for "leaving Chicago." In turn, ISCOMPLETE detects the missing marker (step 211) and no transpositions. CLOSEOFF finds the unresolved node, then checks to make sure that the unsatisfied step permits substitution and that using "leaving" to satisfy step 211 does not create a transposition (0 **atls**). When all the conditions have been satisfied, SUBSTITUTE is invoked and returns the modified context in the figure.

U4S29: "schedule flight #103 on June 13 *leaving* Chicago at 11 p.m. arriving in NY at 2 a.m."
Tokens: (schedule flight %poundsymb 103 on june 13 leaving chicago at 11 p %period m %period arriving in ny at 2 a %period m %period)

Deviation-level 0:
EXPAND Context: [(u-location) (LOC0)], *dlevel* = 0
Error Detection returns: (RETRY 1)
Error Recovery returns: (RETRY 1)

Deviation-level 1:
EXPAND Context: [(leaving@1 u-location) (LOC0)], *dlevel* = 1
Error Detection returns: (LOC0 211 –)
Error Recovery returns: ([LOC0 – (0 (SUBST leaving 211))]) via SUBSTITUTE

Figure 6-20: Identifying **leaving** as a source location marker using SUBSTITUTE.

With the introduction of the SUBSTITUTE action we have exhausted the recovery strategies needed when one deviation point is available to the context. If the available deviation point is contributed by an unresolved node, then a global parse at the current deviation-level must explain that node using either INSERT or SUBSTITUTE. If, on the other hand, there is no unresolved segment left in the utterance, then a global parse at the current deviation-level must require either a DELETE or TRANSPOSE. Table 6-1 summarizes the conditions under which each of the four basic recovery functions is invoked *by itself* from CLOSEOFF. The conditions are expressed in terms of the six discrimination criteria listed at the beginning of this section (page 111). Recall that **dpa** translates as "deviation points available," **urs** is the number of unresolved pcs in the context, **mls** is the number of missing, required steps in the least-deviant subcontext, **tls** is the number of transposed steps in the least-deviant subcontext without substitutions, **atls** is the number of transpositions considering substitutions, and **rtc** stands for "recovery-time constraints." A value of "na" for an entry indicates that the factor does

not apply. **Atls**, for example, are pertinent only if a substitution has taken place. The value of "done" in the **rtc** entries for DELETE and TRANSPOSE indicates that ISCOMPLETE has already performed the relevant constraint checks.

Recovery Function	dpa	urs	mls	tls	atls	rtc
DELETE	1	0	1	0	na	done
TRANSPOSE	1	0	0	1	na	done
INSERT	1	1	0	0	na	na
SUBSTITUTE	1	1	1	0	0	no-sub

Table 6-1: The four general recovery functions and the values of the six discrimination factors required for their use ("na" indicates the factor does not apply).

An utterance that succeeds at Deviation-level 2 (or higher) may accumulate deviation points along paths representing different constituents (as in Figure 6-19) or within the same constituent. In the latter case the recovery actions taken for a subcontext are composed from the basic recovery functions we have just described. Since the Maximum Deviation-level in CHAMP is two, no more than two recovery actions may be used to explain a single constituent. Thus there should be ten pairs of recovery functions displayed in Table 6-2 (four functions choose two plus the four pairs that invoke the same function twice). The extra pair occurs because TRANS/SUB is considered distinct from SUB/TRANS; the former is employed when a **tl** is detected by ISCOMPLETE independent of the substitution, and the latter is employed when the transposition is caused by the substitution during error recovery.

Error recovery when more than one deviation point is available differs from error recovery with a single deviation point in three important respects. First, it may be possible for a single context to satisfy the requirements for a composite recovery action in more than one way. This is impossible for a single deviation point and a single recovery action. SUB/SUB and DEL/SUB are the relevant examples at Deviation-level 2. To invoke SUB/SUB there must be two **mls** and two **urs** in the least-deviant subcontext. Each unresolved node must be tried as a substitution for each missing, required step; if both substitutions are free of consequent transpositions, both paths must be returned to EXPAND. In the kernel grammar for each of the two implemented domains it is impossible for both substitutions to be free of transpositions because all required steps are ordered. Through adaptation, however, a derived grammar may be created that permits either ordering. If that happens then each strategy will succeed using a different binding of **urs** to **mls**, creating distinct paths despite the fact that the effective meanings of the resulting APTs are probably the same. In the case of DEL/SUB there are two **mls** and only one **ur**. It is clear that the unresolved node must be used for a substitution regardless

Recovery Function	dpa	urs	mls	tls	atls	rtc
DEL/DEL	2	0	2	0	na	done
DEL/TRANS	2	0	1	1	na	done
DEL/INS	2	1	1	0	1	no-del
DEL/SUB	2	1	2	0	0	no-del, no-sub
TRANS/TRANS	2	0	0	2	na	done
TRANS/INS	2	1	0	1	na	done
TRANS/SUB	2	1	1	1	0	no-sub
INS/INS	2	2	0	0	na	none
INS/SUB	2	2	1	0	0	no-sub
SUB/SUB	2	2	2	0	0	no-sub
SUB/TRANS	2	1	1	0	1	no-sub, no-trans

Table 6-2: Combinations of recovery functions applicable when two deviation points are available in a context and the values of the six discrimination factors required for their use (''na'' indicates the factor does not apply).

of its position in the segment (otherwise the subcontext would use two deviation points for the two deletions, leaving no points to explain the unresolved node). Yet it is not clear which link should be considered deleted and which substituted. Thus DEL/SUB may produce two paths via a kernel form at Deviation-level 2. As long as neither substitution produces a transposition, both paths must be EXPANDed.

The second difference when more than one deviation point is available, is that a single subcontext may satisfy the conditions for more than one composite recovery action. Again, this was impossible for one deviation point and one recovery action—the basic recovery actions are mutually exclusive. Figure 6-21 demonstrates, however, that any time a SUB/TRANS is possible, a DEL/INS explanation must succeed as well (providing there is no **no-del** recovery time constraint). The utterance is similar to the one in Figure 6-18 except that a head noun, required in the kernel calendar grammar, is missing. At Deviation-level 1 **th** will coalesce with each of **june 16**, **on june 16**, and **at aisys** but will be passed back each time. The **ur** becomes embedded at Agenda-level 3. At that time, contexts for **meetingform**, **seminarform**, **classform**, and **mealform** will all be EXPANDed, although only **meetingform** (recognized via GG1) is shown in the figure. In each case the required head noun will be absent but the token **th** will be in the wrong position to substitute. Thus each subcontext will result in a retry message.

At Deviation-level 2, the four subcontexts are retried and each will result in two paths. On the first path the two available deviation points are used for the substitution of **th** as the head noun in a transposed location. On the second path, it is assumed that the user has

Utterance: (cancel 3 pm on june 16 th at aisys)

Deviation-level 1:
EXPAND context: [(u-hour m-date th@1 m-location) (GG1)], *dlevel* = 1
Error detection returns: (GG1 309 –)
Error recovery returns: (RETRY 2)

Deviation-level 2:
EXPAND context: [(u-hour m-date th@1 m-location) (GG1)], *dlevel* = 1
Error detection returns: (GG1 309 –)
Error recovery returns:
([GG1 – (1 (SUBST th 309) (TRANS th))] via SUB/TRANS
[GG1 – (1 (DELETE 309) (INSERT th))]) via DEL/INS

Figure 6-21: Equivalent hypotheses for "Cancel 3 pm on June 16th at AISys" via SUB/TRANS and DEL/INS using an adapted grammar containing DGG1'.

deleted the head noun and that **th** is an insertion. In all, eight APTs are created as competing hypothetical explanations for the utterance.

When one subcontext can be explained in two ways, either by different bindings within a single composite recovery action or by satisfying the discrimination conditions of different actions, it is possible, indeed likely, that effectively equivalent meanings are created along the EXPANDed paths. If the meanings are equivalent, why do we produce more than one? We will see in Chapter 8 that different hypotheses lead to different derived forms in the adapted grammar. By producing all explanations at the lowest possible deviation-level we avoid biasing the outcome of the learning algorithm unnecessarily.

The third point we need to raise with respect to composite recovery functions concerns a hidden inconsistency. Consider a sentence containing two deletion errors in separate constituents and only one unresolved segment. The least-deviant explanation for such an utterance contains one deletion and one substitution. Depending upon the boundaries between the constituents present in the utterance, it is possible that the unresolved segment sits on the end of two subsegments, each representing one of the constituents with a deletion (for an example, see Figure 8-15 in Chapter 8). Moreover, one constituent may be embeddable inside the other in the Formclass Hierarchy. At Deviation-level 1, the lower level context will EXPAND successfully using SUBSTITUTE. As a result the **ur** will be unavailable to the higher context and only the explanation reflecting a deletion in the higher constituent and a substitution in the lower constituent will be produced. Given that the subsegment was on the end of the higher constituent as well, it should have been possible to interpret the sentence with a top-level substitution and an embedded deletion.We compensate for the inconsistency by having CLOSEOFF return two

modified contexts when the substituting node is an end segment. The first modified context reflects the substitution, as in Figure 6-20. The other modified context looks like: [strategyname (**ur**) (1 (DELETE **ml**))]. This context represents one step down a path that requires two deviation points to succeed—one for the deletion and another for the unresolved pc that must still be accounted for. Since it does not account for the tokens in the unresolved segment the second modified context will be unable to coalesce at the current deviation-level with any path that does. Thus, it will be prevented from contributing to the global parse unless the deviation-level is incremented and a least-deviant-first explanation is still guaranteed.

In an implementation that permits more than two deviations in an utterance, a discrimination algorithm capturing a more general analysis than the breakdown by cases in Tables 6-1 and 6-2 would be appropriate. Figure 6-22 shows that in CHAMP the case analysis suffices for the error recovery routine, CLOSEOFF. By this time the algorithm should seem straightforward; essentially, the results of error detection are used to select a single or composite recovery function. If ISCOMPLETE returns a list of deviant subcontexts, the **mls** and **tls** computed for each strategy are augmented by values for the unresolved segments (**urs**), available deviation points (**adp**), additional transpositions (**atls**), and recovery-time constraints (**rtc**). Taken together the six factors determine how many and which of the recovery functions create the least-deviant explanation. If a recovery function does not require a **ur** representing an end subsegment, the **ur** is passed back. When the **adp** is one, only a single function will be indexed, although we create a second modified context under the special circumstances in step (5a) to compensate for the anomaly described above. When two deviation points are available more than one composite action may be invoked; each modified context produced for each least-deviant strategy is returned to EXPAND (step (4c)).

CLOSEOFF returns one of two values to EXPAND: a retry message or a list of modified contexts (OK returns a modified context without modifications). As promised in Section 6.1, Figure 6-23 shows EXPAND's response in each situation. If a retry message is returned, the candidate constituent failed to create a path at the current deviation-level. Under these circumstances, EXPAND caches the retry value and returns without building any higher level nodes (step (2)). If, on the other hand, CLOSEOFF returns a list of modified contexts then EXPAND caches the value "true" for the context and constructs its higher level constituents by incorporating the changes required to explain the user's error with respect to a particular strategy.

Imagine the set of utterances recognizable by a given grammar as defining a search space. If the user's utterance is not within the space we rely on the concept of a least-deviant-first parse to extend our search to a larger space in a principled way. Each time through the Parse/Recovery Cycle we extend the search space outward to include more deviant utterances until the best explanation for the grammar is produced or the

```
CLOSEOFF (context)
Let error-info = ISCOMPLETE (context)                          (1)
IF error-info = retry message                                  (2)
THEN RETURN (RETRY (message))
Compute urs and adp
IF error-info has no mls or tls AND context has no urs         (3)
THEN RETURN (OK (strategy names))
FOR EACH triple in error-info, DO                              (4)
  use the values for strategyname, mls and tls in the triple
  IF adp = 1                                                   (4a)
  THEN choose recovery action using chart in Figure 6-1,
          computing atls, rtc and pass back as needed
    add to the modified contexts the value returned by the recovery function
    IF the recovery function is SUBSTITUTE AND the ur is on the end   (4a)
    THEN add modified context: (strategyname (ur) (1 (DELETE ml)))
  ELSE choose composite recovery action(s) using the chart in Figure 6-2,   (4c)
          computing atls, rtc and pass back as needed
       FOR EACH composite action chosen, DO
         add to the modified contexts the value returned by the composite
RETURN list of modified contexts                               (6)
```

Figure 6-22: CHAMP's CLOSEOFF algorithm for performing error recovery.

Maximum Deviation-level for the implementation is reached. To traverse the space defined by a given deviation-level efficiently, we must take advantage of every possible source of constraint, eliminating unprofitable paths as soon as we are certain they cannot explain the utterance. In the model underlying CHAMP, constraint is provided by the violation of expectations. Some types of violations are non-recoverable, including the violation of semantic constraints found in the Concept Hierarchy, the contiguity required of co-constituents, and the need to account for all the words in the sentence. An interpretation that exposes a non-recoverable error will always be terminated and its path never re-explored. In contrast, the violation of expectations embodied in the forms that comprise the grammar are considered recoverable. A constituent requiring one or more insertions, deletions, substitutions, or transpositions may be constructed using appropriate recovery functions if the resulting path still lies within the extended search space. In essence, Error Detection & Recovery transforms an explainable path into one that has been explained.

The goal of efficient search is facilitated by creating kernel forms with critical differences. During both phases of the Parse/Recovery Cycle the manifestation of a critical difference by constituents in the utterance eliminates portions of the search space.

```
EXPAND (context)
Let closeval = CLOSEOFF(context)                                        (1)
IF closeval is a retry message                                          (2)
THEN cache the retry value for the context and RETURN
ELSE cache "true" for the context
FOR EACH modified context, mc=(strat (urs) (d-adjust notation)), DO     (3)
 IF the mc has passed back urs
 THEN remove them from the subnodes and move them up the Agenda
        decrease the dlevel of the context by 1 per passed back ur
        cache "true" for the reduced context
  IF the subnodes satisfy all expand-time constraints for the context's class
  THEN create a pnode for the class that spans the subnodes
       IF there is an expand-time ig for the class
       THEN invoke it AND cache the canonical value for the pnode
  FOR EACH active step that seeks the pnode's class, DO
    IF the step has a bind-time constraint AND the pnode satisfies it
    THEN create a pc out of
                the step
                the pnode
                the (possibly modified) subnodes
                the mc's strat
                the dlevel of the context + the mc's d-adjust
                the notation
         and place it on the appropriate level of the Agenda
```

Figure 6-23: CHAMP's EXPAND Algorithm reexamined with special attention to the results of Error Detection and Recovery (Extension of Figure 6-10).

The presence of a segment in an utterance will terminate those paths using strategies that do not seek the segment in favor of paths using strategies that do. Alternatively, a segment may partition strategies in the same formclass according to the amount of deviation required to explain the segment as an instance of the class. Under these circumstances, only the least-deviant paths are preserved.

CHAMP's cache also facilitates efficient search. As the deviation-level increases, the cache prevents the more deviant interpretations of an explained segment from being explored. In addition, CHAMP's cache prevents useless work during each new Coalesce/Expand Cycle by directing search only along paths that previously failed but for which there is evidence of potential success at the current deviation-level.

Whether the result of the Parse/Recovery Cycle is a single explanation of the user's

utterance or a set of competing hypotheses concerning the user's intent, the understanding process is not yet complete. In the next chapter we examine Resolution—the phase in which a final effective meaning is assigned to the utterance. Resolving the effective meaning may require any or all of: dividing a set of APTs into equivalence classes based on effect, checking for the consistency of an interpretation against events in the databases, and finding default values for missing information using the Concept Hierarchy, simple inferences, or the user's help.

Chapter 7
The Resolution Phase: Choosing Among Explanations

When an utterance is understandable using the current grammar, the result of the Parse/Recovery Cycle is one or more meaning representations of the sentence (APTs) at a particular deviation-level. Although each APT represents an action to be performed in the domain, there is no guarantee that all APTs represent the same action or that any represented action is meaningful in the context established by the contents of the calendar and airline schedule. The Resolution phase of processing is responsible for enforcing database constraints, establishing a single meaning for the utterance, and performing the database action represented by that meaning (to review the position of the Resolution phase in the complete understanding process see Figure 4-1). There are three essential issues in resolution. First, do any of the represented actions correspond to the user's intent? This question must be asked when more than one effect is present or when there is a single effect at a non-zero deviation-level. Second, can the intended action be performed? This question is always pertinent. Third, what form should interaction with the user take? This question relates to the first one—determining the user's intent often requires user interaction.

In many ways the third question is the most difficult. How do we enlist the user's aid intelligently—without inundating her with choices or requiring her to act as linguist or programmer? Consider that a sentence as simple as

"Cancel the mtg June 5 at 3"

produces *twelve* APTs when CHAMP uses the kernel grammar for the calendar domain.[35] Figure 7-1 displays the explanations divided into four sets; explanations within a set differ only by the type of **groupgathering** they identify. In keeping with the notation introduced in Section 2.2, the first group of four corresponds to the hypothesis that **mtg** is an insertion and the head noun has been omitted. In the second set, **mtg** is treated as a substitution for the head noun and the marked form of the date contains a deletion. In the final group, the head noun is omitted and **mtg** substitutes for the missing date marker. No interpretation is shown corresponding to a substitution of **mtg** for the

[35]Eighteen explanations are produced when both domain kernels are used; in each group in Figure 7-1 there would be an extra explanation for each kind of trip (air and nonair).

head noun plus a transposition of the unmarked prenominal date because the transposition is prevented by a **no-trans** recovery-time constraint (we suspended that constraint for demonstrative purposes only in Chapter 5).

1. Cancel the >mtg< June 5 [**meetingwd**] at 3
2. Cancel the >mtg< June 5 [**seminarwd**] at 3
3. Cancel the >mtg< June 5 [**classwd**] at 3
4. Cancel the >mtg< June 5 [**mealwd**] at 3

5. Cancel the *mtg*/**meetingwd** [on] June 5 at 3
6. Cancel the *mtg*/**seminarwd** [on] June 5 at 3
7. Cancel the *mtg*/**classwd** [on] June 5 at 3
8. Cancel the *mtg*/**mealwd** [on] June 5 at 3

9. Cancel the [**meetingwd**] *mtg*/**datemkr** June 5 at 3
10. Cancel the [**seminarwd**] *mtg*/**datemkr** June 5 at 3
11. Cancel the [**classwd**] *mtg*/**datemkr** June 5 at 3
12. Cancel the [**mealwd**] *mtg*/**datemkr** June 5 at 3

Figure 7-1: The twelve explanations produced by the Parse/Recovery Cycle for "Cancel the mtg June 5 at 3" using the calendar kernel.

What is an appropriate method for presenting such a set of alternatives to the user? Certainly the list in Figure 7-1 is inappropriate—it requires that the user be familiar with the internal representation of the grammar as well as a number of notational conventions. An interaction such as the one in Figure 7-2 is somewhat better; the list is shorter and contains neither notation nor obscure references to internal grammatical categories. The format in Figure 7-2 uses a partial error analysis to distinguish alternatives. In this case, we have created the choices solely from hypothesized substitutions. The result is reasonably "friendly" although incomplete. Had we tried to capture the other information in the choices in Figure 7-1 we would have been forced to ask questions such as "Do you want to omit the marker for the date case?" and "Do you want to omit the head noun?" Of course, utterances with errors that are not substitutions will occasion questions like these as well. It seems inappropriate to assume that the user could, in general, answer such questions accurately. If we wish to assume any expertise on the part of the user it should be task expertise, not linguistic expertise.

What is wrong with interaction formats like those in Figures 7-1 and 7-2 is that they are designed to help the system, not the user. In contrast, CHAMP's approach to resolution incorporates the belief that for an interaction to make sense to the user it must be conducted within her frame of reference. In short, the user is concerned with accomplishing her task—scheduling calendar events—not with teaching grammar. The question in her mind is not "Will the system learn the substitution?" but "Will the

Do you want:
(1) "mtg" to be a synonym for "meeting"?
(2) "mtg" to be a synonym for "seminar"?
(3) "mtg" to be a synonym for "class"?
(4) "mtg" to be a synonym for "breakfast"?
(5) "mtg" to be a synonym for "lunch"?
(6) "mtg" to be a synonym for "dinner"?
(7) "mtg" to be a synonym for "on"?
(8) none of the above
[number between 0 and 8]:

Figure 7-2: Using error analysis to create an alternative format for user interaction when resolving the meaning of "Cancel the mtg June 5 at 3."

system do the right thing?" Thus, from the user's point of view the goal of resolution is not establishing a unique explanation for the utterance but establishing her intended effect.

The difference this focus makes in the nature of an interaction can be seen in Figure 7-3. The interaction shown in each of the three examples results from mapping the set of explanations produced by the Parse/Recovery Cycle into subsets that have the same effect in the domain. It is the potential effects that are then presented to the user. The user's choice, in turn, determines the subset of root nodes that are passed from the Resolution phase to the adaptation functions. The differences between the interactions result from the differences in the assumed contents of the database.

Let us look at the three scenarios in Figure 7-3 more closely. In EXAMPLE 1 we assume that the database contains the entries shown at the top of the figure: an AI seminar scheduled from 10 a.m. to noon in room 5409, lunch from noon to 1 p.m. in an unspecified location, and a meeting with Ed in the professor's office from 3 to 4 p.m. Under these circumstances CHAMP finds only one effect both present in the list of twelve explanations and possible within the constraints enforced by the database. Note that pinpointing the intended effect reduces the hypothesis set from twelve to three: the returned root nodes (pc204, pc208, and pc200) correspond to choices 1, 5, and 9 in Figure 7-1—one of each of the three hypothesized error combinations applied to the **meeting** object.

In EXAMPLE 2 of Figure 7-3, the database is changed slightly so that the meeting with Ed now takes place from 4 to 5 p.m. As a result, no calendar entry is a perfect match to the information contained in any of the explanations. On the other hand, each of the entries matches in part—although the times are wrong, there are explanations that expect to find a meeting, a seminar, and lunch. The partial matches create the choice list shown; when the the user chooses the cancellation of the meeting, the hypothesis set is reduced

```
EXAMPLE 1 assumes the following calendar entries for June 5:
(date (6 5 *) starthr 1000 endhr 1200 ggtype seminar location (* 5409) general-topic ai)
(date (6 5 *) starthr 1200 endhr 1300 ggtype lunch location (* *))
(date (6 5 *) starthr 1500 endhr 1600 ggtype meeting location (* office) participants ed)

Do you want:
  to delete: JUNE 5  3:00pm - 4:00pm MEETING in/at OFFICE with ED ?
[y or n]: y
Ok, done.
(returning roots: 204 208 200)

EXAMPLE 2 assumes the following calendar entries for June 5:
(date (6 5 *) starthr 1000 endhr 1200 ggtype seminar location (* 5409) general-topic ai)
(date (6 5 *) starthr 1200 endhr 1300 ggtype lunch location (* *))
(date (6 5 *) starthr 1600 endhr 1700 ggtype meeting location (* office) participants ed)

Do you want:
(0) to delete: JUNE 5  12:00pm - 1:00pm LUNCH ?
(1) to delete: JUNE 5  10:00am - 12:00pm AI SEMINAR in/at 5409 ?
(2) to delete: JUNE 5  4:00pm - 5:00pm MEETING in/at OFFICE with ED ?
(3) none of the above
[number between 0 and 3]: 2
Ok, done.
(returning roots: 204 208 200)

EXAMPLE 3 assumes no calendar entries for June 5.

There are no events scheduled on June 5.
Please try again.
(returning roots: nil)
```

Figure 7-3: Resolution in CHAMP: the confirmation of a subset of explanations for "Cancel the mtg June 5 at 3" based on the user's selection of her intended effect.

to the same subset of explanations as in EXAMPLE 1. Note that although the choices offered to the user are different in EXAMPLE 1 and EXAMPLE 2, the returned root lists are the same because the chosen effects are the same.

The final example in the figure shows one variant of CHAMP's behavior when none of the explanations correspond to an effect that is possible. Since the action of deleting an entry is meaningless if no entries are present, the system detects the error condition and returns the most informative message it can. In contrast to the displayed behavior, if there had been entries present, but no perfect or partial matches, the message would have been "No calendar entry fits that description on June 5."

There are two advantages to this approach to resolution, which we call *confirmation by*

effect. First, the interaction demands no more of the user than the knowledge of what she intended by the utterance. Second, the system has access to an additional knowledge source to focus the user's choices: the state of the "real world" as represented by the databases. In other words, in the process of trying to answer the question "Do any of the represented actions correspond to the user's intent?" we will automatically answer the question "Can the intended action be performed?" The main disadvantage to the approach is that it does not guarantee a unique explanation in terms of the grammar (although it does guarantee a unique meaning in terms of the database).

Figure 7-4 displays the resolution algorithm in CHAMP. The process is composed of five steps: the transformation of explanations into database action records, the application of default reasoning and inference to incomplete action records, the transformation of complete records into virtual effects, the selection of the intended effect by the user (if necessary), and the performance of the database action. In the remainder of this chapter we will examine each of these steps in turn.

```
RESOLVE (rootlist)
FOR EACH root node, DO                                              (1)
  Transform the root node into a database action record
  Group root nodes with identical records
FOR EACH record, DO
  APPLY-DEFAULT-LOGIC to the record                                 (2)
  IF the record contains all the information required by its database action
  THEN convert the record to a virtual effect                       (3)
  Group the roots with identical effects
IF there is a single non-error effect AND the deviation-level is 0  (4)
THEN perform the action and RETURN the roots corresponding to the effect
ELSE IF the effect is an error                                      (5)
      THEN display the appropriate error message
ELSE pass the effects to USERQUERY                                  (6)
      IF the user confirmed an effect
      THEN RETURN the corresponding roots
```

Figure 7-4: CHAMP's RESOLVE algorithm which controls the Resolution phase.

7.1. Converting Meaning to Action

An explanation of an utterance is created by the Parse/Recovery Cycle in the form of an annotated parse tree (APT). The first step in resolution requires that we convert each APT into a *database action record*. Doing so allows the system to place root nodes producing identical database action records into the same preliminary equivalence class.

A database action record contains three pieces of information: the name of the action, a source *database record*, and a target database record (see Figure 7-5). The list of possible actions is the same in both of CHAMP's domains: **insert**, **remove**, **alter**, and **retrieve**, corresponding to the action concepts **add**, **delete**, **change**, and **show** in the Concept Hierarchy. A detailed discussion of the database functions can be found in Section 7.3.

Each database action record requires a source database record, but **alter** requires a target record as well (for the other actions the target record is empty). Figure 7-5 shows that the fields in a source or target database record depend upon the domain. EXAMPLE1 and EXAMPLE 2 in Figure 7-3 show instances of database records from the calendar domain in which fields containing nil values have been suppressed.

Database Action Record	Calendar Domain Database Record	Travel Domain Database Record
action	date	date
source database record	ggtype	triptype
target database record	starthr	departure
	endhr	arrival
	location	origin
	participants	destination
	subject	flightnumber
	general-topic	

Figure 7-5: The record structures used in database interactions.

The creation of a database action record from an APT occurs with the help of *bindvars* and instance generators (*igs*), both of which were introduced in Section 4.2. Each record field has one or more identifying labels associated with it, labels that are placed in the *bindvar* lists of the steps that may fill the fields. In this way a tree-walk over the APT allows us to retrieve the values to associate with the *action*, *source* and *target* fields regardless of where or how they occur in the utterance. The actual value placed in a field is either the canonical value for the pc satisfying the step, or the tokens associated with the pc if a canonical value is not available. A canonical value may be computable from a resolution-time ig or may have been previously computed by an expand-time generator. Figure 7-6 demonstrates how the APT for the fifth explanation in Figure 7-1 becomes a database action record (the actual record is shown in Figure 7-7).

Utterance: Cancel the mtg June 5 at 3
Explanation 5: Cancel the *mtg*/**meetingwd** [on] June 5 at 3

APT:
204: *ROOT* (0 6 deleteforms) (ACT2) 2
daction *in *ROOT*'s bindvar list sets action to* **remove** *via* **ig-act**
1: 705 (0 0 deletewd)
125: 709 (1 6 m-d-ggforms) (DGG1) 2
2: 401 (1 1 defmkr)
56: 403 (2 6 meetingforms) (GG1) (2 (SUBST 0 309))
dggtype *in 403's bindvar list sets the source ggtype to* **meeting** *via* **ig-mrgg**
0: 0 (2 2 unknown) nil 1
34: 316 (3 4 m-dateforms) (DATE0) (1 (DELETE 201))
22: 203 (3 4 u-dateforms) (DATE1)
ddate *in 203's bindvar list sets the source date to (6 5 *) via* **ig-date**
3: 101 (3 3 monthwd)
6: 102 (4 4 number)
24: 315 (5 6 m-hourforms) (HR0)
7: 111 (5 5 hrmkr)
16: 113 (6 6 u-hourforms) (HR1)
dshour *in 113's bindvar list sets the source starthr to (3 0 ?) via* **ig-hour**
9: 4 (6 6 number)

Figure 7-6: Converting an APT to a database action record using *bindvars* to indicate record fields and instance generators to produce canonical values.

The figure shows that the value of the *action* field is computed by the resolution-time instance generator, **ig-act**, when the token **daction** is found on the *ROOT* step's *bindvar* list. **Ig-act** chooses the database action **remove** for pc204 predicated upon the class of the root node (**deleteforms**). Steps 705, 709, and 401 do not contribute to the record but step 403's *bindvar* list contains **dggtype**. Since no *bindvar* list containing the label **target** has been encountered in our recursive descent, we know that we should fill the *ggtype* field of the *source* database record. Again, a resolution-time ig is invoked (**ig-mrgg**) to return a canonical value—in this case the token **meeting**. Just as the value produced for the *action* field relied on the class of the root node rather than the token acting as verb, producing the value **meeting** for the *ggtype* field relies on the class of pc56 (**meetingforms**) rather than the token bound to the step seeking the head noun (step 309). By building database records with canonical values we preserve the important information in the meaning structure even when the referring phrase is unknown, as in the substitution in our example, or when the referring phrase is missing, as in a deletion of the head noun. In other words, every APT refers to some kind of action and object in the domain—the APT rooted at pc204 hypothesizes the existence of a meeting regardless of the actual referring phrase. The canonical value makes comparison against records in the calendar database both easier and more constraining. Consider that if we had filled

the *ggtype* field with the previously unencountered token **mtg** instead of **meeting**, comparison against actual database records could result in a partial match at best.

Both the *date* and *starthr* fields in the source record are filled by canonical values computed (and cached) by expand-time instance generators. Step 303 provides the *source date* through **ddate** and **ig-date**. Similarly, step 113 provides the *source starthr* through **dshour** and **ig-hour**. Interpreting the single hour ("at 3") as the *starthr* rather than the *endhr* is an inference that is forced by placing the label **dshour** in step 113's *bindvar* list. The inference stems from the observation that people often do not indicate a definite ending time for an appointment when they schedule it [33].

When the twelve APTs produced by the Parse/Recovery Cycle for our sample sentence have been converted to database action records, four equivalence classes are formed. Figure 7-7 shows that, in terms of effect, the discriminating factor among the explanations is the kind of **groupgathering** object each expects to find in the database. Thus, explanations 1, 5, and 9 (from Figure 7-1) correspond to the removal of a meeting entry from the database, explanations 2, 6, and 10 correspond to the removal of a seminar, 3, 7, and 11 correspond to the removal of a class, and 4, 8, and 12 to the removal of a meal. These equivalence classes stand in contrast with the original grouping in Figure 7-1 which organized the explanations according to similarities in the error recovery combinations.

Action Record for 1, 5 & 9	Action Record for 2, 6 & 10	Action Record for 3, 7 & 11	Action Record for 4, 8 & 12
action remove	action remove	action remove	action remove
source	source	source	source
date (6 5 *)	date (6 5 *)	date (6 5 *)	date (6 5 *)
ggtype meeting	ggtype seminar	ggtype class	ggtype meal
starthr (3 0 ?)	starthr (3 0 ?)	starthr (3 0 ?)	starthr (3 0 ?)
endhr nil	endhr nil	endhr nil	endhr nil
target nil	target nil	target nil	target nil

Figure 7-7: The initial database action records defining four equivalence classes for the explanations in Figure 7-1 (some nil fields suppressed).

As a final observation concerning the database action records produced by the first step in resolution, note that the value left by **ig-hour** remains unresolved in terms of the twenty-four hour clock representation that CHAMP uses internally, and that the *ggtype* **meal** remains unresolved in terms of the choices defined for the domain (breakfast, lunch, and dinner). The problem of transforming unresolved values into values that can be compared against the database brings us to the topic of default knowledge and inference.

7.2. Using Default Knowledge and Inference

One of the early arguments against natural language interfaces to database systems was that user utterances would fail to express legal database commands (see, for example, [46], [61], and [44]). Critics observed that in ordinary language use people tend to leave some necessary information implicit, relying on the listener to make use of common knowledge and commonsense reasoning.[36] Interface designers have reacted to the criticism by providing rudimentary forms of the same abilities in their systems. In CHAMP we allow for the user's imprecision by detecting missing information and unresolved values during resolution and using default information and inference to compensate.

Exactly what constitutes necessary but missing information depends upon how the database functions are implemented. In CHAMP the database record fields that require values are determined by the database function in the *action* field and the domain represented by the type field (*ggtype* indicates the calendar domain, *triptype* indicates the travel domain). Figure 7-8 displays the relationships. In most cases, only a date must be present in the utterance. When adding an entry to the calendar, however, the *starthr* and *location* fields must also be filled if the object is a **groupgathering**, while the *departure* and *destination* fields must also be filled if the object is a type of trip. **Alter** inherits its requirements from **remove** and **insert** because an **alter** action is implemented as a **remove** of the source followed by an **insert** of the target.

	calendar	travel
retrieve:	date	none
remove:	date	date
insert	date starthr location	date departure destination
alter:	source same as remove	source same as remove
	target same as insert	target same as insert

Figure 7-8: The source database record fields that require values, as determined by the database action and the type of database object.

Since the entry for "remove" and "calendar" in Figure 7-8 contains only **date**, we conclude that the database action records in Figure 7-7 have no missing information. The records do, however, have unresolved values. What constitutes an unresolved value also depends upon what the database functions expect. CHAMP's database functions expect dates to be represented as a list containing integers for the day, month, and year, although

[36]Critics also argue that natural language encourages users to form requests outside the system's capabilities—an issue we will not address.

the year may be represented by a wildcard (*) which indicates "any value." Hours are expected to be represented as wildcards or single integers between zero and 2400 that encode the hour and minutes in the obvious way. Values filling the *location* field must be a list with two elements in the calendar domain, either element may be a wildcard. The grammar and instance generators enforce the convention that the first element in the list corresponds to a major location (a building, school, or business) while the second element is reserved for a room name or room number (5409 or office, for example). In the travel domain, a location must be a major location or the name of a city.

The database functions also expect that one of a fixed set of values will fill the *action*, *ggtype*, and *triptype* fields. As we have mentioned, the *action* field must contain one of **remove**, **insert**, **alter**, or **retrieve**. The *ggtype* must be one of **meeting**, **seminar**, **class**, **breakfast**, **lunch**, or **dinner**. The *triptype* field must hold one of **air** or **nonair**.

Given these expectations we see that there are three unresolved values among the database action records in Figure 7-7: neither the *starthr* nor *endhr* have the appropriate form, and the token **meal** is an invalid *ggtype*.

Missing or unresolved values may be corrected by copying default values from the Concept Hierarchy (Section 4.3) or by inference. Consider, for example, the sentence

"Schedule a Prodigy meeting on June 12."

Figure 7-8 shows that the database action record for this utterance must have a *starthr* and *location* in addition to the date that is present. The **meeting** concept provides the default location (* **office**) but no time information to fill the *starthr* or resolve the **nil** *endhr*. We change a **nil** *endhr* to a wildcard based upon the the domain-dependent inference (justified in [33]) that a missing *endhr* reflects an intent by the user to leave the ending time open. To set the *starthr*, CHAMP uses a domain-independent inference and tries substituting the last *starthr* mentioned.

In total, CHAMP has seven strategies to compensate for missing or unresolved values:

- APPLY-DEFAULTS retrieves default values from the Concept Hierarchy based upon the concepts represented in the record fields (another reason that canonical values are important).
- ADJUST-GIVEN-TIMES converts a **groupgathering** time in the form (hour min ?) to a 24-hour value using the inference that an hour between one and seven should be post-meridian. The function also converts unresolved times for non-air trips by user interaction (unresolved times for flights are inferred by ADJUST-FLIGHTINFO).
- ADJUST-EMPTY-TIMES converts to wildcards all unspecified times that are not required by the domain and action.
- ADJUST-FLIGHTINFO is used when the *triptype* is **air** to fill in the complete airline database record from the information in the utterance. If, for example, a

flight number is given by the user, then the destination and departure time can be retrieved by looking up the flight in the airline schedule. Similarly, if a destination and departure are given, a unique flight number may be available.

- ADJUST-MEAL is invoked only when the *ggtype* is the unresolved value **meal**. This strategy checks the *starthr* to ascertain the most likely meal. If no *starthr* is given then it replaces the database action record containing *ggtype* **meal** with three new database action records: one each for breakfast, lunch, and dinner.
- LAST-VALUE tries to retrieve a missing, required value from the database action record chosen for the last utterance CHAMP successfully understood. Clearly, LAST-VALUE is not guaranteed to succeed. If the previous interaction was of the wrong type, for example, the desired value will not be available (as when the current utterance is missing a flight destination but the previous utterance concerned a seminar).
- TRANSFER-VALUE is used only for the **alter** action which requires a full target database record; the strategy is responsible for preserving the appropriate values from the source in the target.

The first five strategies in the list are domain-dependent, the last two are domain-independent. Taken together, LAST-VALUE and TRANSFER-VALUE are a weak mechanism for handling some types of ellipsis.

The order in which the seven inference strategies are applied will, in general, affect the contents of the database record produced. If, for example, we apply ADJUST-MEAL before APPLY-DEFAULTS we will have access to the default *starthr* available in the Concept Hierarchy for each of breakfast, lunch, and dinner. If, on the other hand, we do not apply ADJUST-GIVEN-TIMES before ADJUST-MEAL we may not be able to use the *starthr* to determine which meal is most likely. The application order used by CHAMP is shown in the algorithm APPLY-DEFAULT-LOGIC (Figure 7-9). It is not the only possible ordering, nor does it always produce correct results, but it does something useful most of the time.[37] The strategy TRANSFER-VALUE does not appear in the algorithm; it is used only by the database function ALTER (described in Section 7.3).

Notice that the algorithm returns only completed database records (step (7)). This guarantees that the database functions will have available the information they need to perform their actions. It also means that APPLY-DEFAULT-LOGIC may change the number of equivalence classes formed by the conversion process in the previous section. To see how this might happen, consider the sentence

"Forget meeting June 12."

in which "forget" is unknown. Using the kernel grammar, the Parse/Recovery Cycle will produce two roots for this sentence: an **addform** and a **deleteform**. During the first

[37]We will evaluate the degree to which CHAMP's accuracy suffers because of its inferences in Chapter 9.

APPLY-DEFAULT-LOGIC (database action records)
FOR EACH database action record, DO
  ADJUST-GIVEN-TIMES (1)
  IF the type of the record is **air** THEN ADJUST-FLIGHTINFO (2)
  IF the type of the record is **meal** THEN ADJUST-MEAL (3)
  IF there are missing required values THEN APPLY-DEFAULTS (4)
  ADJUST-EMPTY-TIMES (5)
UNLESS there are records that are complete
  try to fill missing fields in each record using LAST-VALUE (6)
RETURN only records without missing information (7)

Figure 7-9: CHAMP's algorithm for applying default knowledge and inference.

step of resolution, each root, in turn, becomes a distinct database action record (one with the action set to **insert**, the other with the action set to **remove**), creating two equivalence classes of roots with one member in each class. The **remove** action requires only the date that is present. The **insert** action, however, requires each of a date, location, and start time. Thus, after applying default logic only one equivalence class will remain—the existence of the complete **remove** record blocks the call to LAST-VALUE (step (6)) that might have compensated for the missing values in the **insert** record. Since the explanation hypothesizing a **remove** requires fewer inferences than the explanation hypothesizing an **insert**, **remove** is preferred.

The effect of the inference process on the database action records in Figure 7-7 is shown in Figure 7-10. ADJUST-GIVEN-TIMES infers that the value **(3 0 ?)** should be 3 p.m., and replaces the old value with **1500**. Subsequently, ADJUST-MEAL notices that 3 p.m. is more likely to be lunch than dinner or breakfast and replaces **meal** with **lunch**. Since the *endhr* field is not required by the combination of **remove** and the calendar domain, ADJUST-EMPTY-TIMES sets the *endhr* to a wildcard.

Final Record for 1, 5 & 9	Final Record for 2, 6 & 10	Final Record for 3, 7 & 11	Final Record for 4, 8 & 12
action remove	action remove	action remove	action remove
source	source	source	source
date (6 5 *)	date (6 5 *)	date (6 5 *)	date (6 5 *)
ggtype meeting	ggtype seminar	ggtype class	ggtype lunch
starthr 1500	starthr 1500	starthr 1500	starthr 1500
endhr *	endhr *	endhr *	endhr *
target nil	target nil	target nil	target nil

Figure 7-10: The database action records defining the four equivalence classes for the explanations in Figure 7-1 after default logic is applied (compare with Figure 7-7).

With the conversion and inference steps complete a set of database action records are

available that are "syntactically correct" from the point of view of the database functions. We are ready, therefore, to tackle the question, "Can the intended action be performed?"

7.3. Interacting with the Databases

At the beginning of a session CHAMP must be told which domain to work with: calendar, travel or both. If the choice is "calendar," the system will have access to a collection of database records, organized by their *date* fields, which we have been calling the calendar. If the response is "travel" or "both," CHAMP will have access to the calendar as well as an auxiliary collection of database records representing an airline schedule. Acting on these databases are the four database functions that serve CHAMP's two domains: REMOVE, INSERT, ALTER, and RETRIEVE. All four functions are defined as operations on the calendar. RETRIEVE has the distinction of being defined as an operation on the airline schedule as well.

In the process of performing the user's intended database action it may be necessary to call any of the database functions in either of two ways: for virtual effect or for actual effect. A synopsis of each function's response under each condition is shown in Figure 7-11. Although the action ultimately taken by each database function is fairly straightforward, the "virtual" effect may not be. In essence, when we call a database function for virtual effect we are looking for the result of a comparison between the information in the source database record and the entries in the calendar on the source date (or in the airline schedule, regardless of date). The purpose of the comparison depends upon the function and is reflected by the kind of error that may occur. If we ultimately wish to INSERT a new calendar entry then we must make sure that the new entry times do not overlap with previously scheduled appointments. If we are trying to REMOVE a calendar entry then such an entry must exist, and the match between the entry we wish to remove and the entry that is present must be close, although not necessarily exact. ALTERing an entry is essentially the same as removing one record and inserting another; thus, any of the errors that might occur during a REMOVE or INSERT might occur during an ALTER as well. In contrast to REMOVE and ALTER, an empty entry for a particular calendar day is not considered an error by RETRIEVE; a response of "There are no appointments for that day." may well have been what the user wanted.

Comparisons against the calendar for REMOVE and ALTER are permitted to be only partially successful. This allows CHAMP to compensate for erroneous knowledge on the part of the user (as in EXAMPLE 2 of Figure 7-3) and to overcome the need for canonical values in all record fields. By permitting a partial match, an appointment "about finances" may still be cancelled by an utterance referring to "the financial meeting." EXAMPLE 1 of Figure 7-3 demonstrates that when a perfect match occurs, those alternatives are offered to the user first.

INSERT (database action record)
virtual effect: returns database record to be added or overlap error
actual effect: inserts record in calendar on date

REMOVE (database action record)
virtual effect: returns database record(s) matched, nomatch error, or noevents error
actual effect: removes intended record from calendar

ALTER (database action record)
virtual effect: returns one or more pairs of database records in which the first record matched and the second record is to be added, or overlap, nomatch, or noevents error
actual effect: removes intended record and inserts corresponding new record

RETRIEVE (database action record)
virtual effect: returns record(s) to be displayed from appropriate database
actual effect: displays record(s) or emptyday message

Figure 7-11: The four database functions and the effects they may produce.

To see how the match process works, we will switch our attention from the sentence in Figure 7-3 to the sentence

"Change the June 5 mtg from 3-4 pm to 1-2 pm."

In this case the Parse/Recovery Cycle produces only four explanations, interpreting "mtg" as each of a possible **seminarwd**, **meetingwd**, **classwd**, and **mealwd**. The four database action records produced by conversion and inference are shown in Figure 7-12. Notice that at this point the target database records contain only the *starthr* and *endhr*—the values that are explicitly given in the utterance.

Record1 (pc381)	Record2 (pc382)	Record3 (pc383)	Record4 (pc384)
action alter	action alter	action alter	action alter
source	source	source	source
date (6 5 *)	date (6 5 *)	date (6 5 *)	date (6 5 *)
ggtype meeting	ggtype seminar	ggtype class	ggtype lunch
starthr 1500	starthr 1500	starthr 1500	starthr 1500
endhr 1600	endhr 1600	endhr 1600	endhr 1600
target	target	target	target
starthr 1300	starthr 1300	starthr 1300	starthr 1300
endhr 1400	endhr 1400	endhr 1400	endhr 1400

Figure 7-12: The database action records defining the four equivalence classes for explanations of "Change the June 5 mtg from 3-4 pm to 1-2 pm."

The virtual effect of a call to ALTER for each of the four database action records is

shown in Figure 7-13 for one set of June fifth calendar entries. For CALENDAR1, Record1 of Figure 7-12 is a perfect match to entry r3. ALTER records the fact that the match was perfect and that r3 is the record to be removed from the calendar if the effect being constructed is eventually chosen. Record2, Record3, and Record4 of Figure 7-12 produce only partial matches to CALENDAR1; the matches for Record2 and Record4 come from their *ggtypes*, while Record3's match stems from its *starthr* and *endhr* values.

CALENDAR1 for (6 5 *)
r1 starthr 1000 endhr 1200 ggtype seminar location (* 5409) general-topic ai
r2 starthr 1200 endhr 1300 ggtype lunch location (* *)
r3 starthr 1500 endhr 1600 ggtype meeting location (* office) participants ed

ALTER (Record1):
(perfect r3 (date (6 5 *) starthr 1300 endhr 1400 ggtype meeting
location (* office) participants ed))

ALTER (Record2):
(partial r1 (date (6 5 *) starthr 1300 endhr 1400 ggtype seminar
location (* 5409) general-topic ai))

ALTER (Record3):
(partial r3 (date (6 5 *) starthr 1300 endhr 1400 ggtype class location (* 5409))

ALTER (Record4):
(partial r2 (date (6 5 *) starthr 1300 endhr 1400 ggtype lunch location (* *)))

Figure 7-13: The values returned by calls to the database function ALTER for virtual effect for each of the database action records in Figure 7-12 under CALENDAR1.

Along with the degree of match and a pointer to the record to be removed, ALTER's virtual effect includes a record to be added to the calendar. To create the new record, ALTER calls TRANSFER-VALUE (see Figure 7-14) which uses the source record and the matched record to fill in fields in the target. In general, the algorithm works well if the source type and matched type are the same (Record1, Record2, and Record4 in Figure 7-12, for example), or if the target type differs from the matched type but is given explicitly in the utterance (as in "Change the meeting at 4 to a seminar at 5"). The algorithm tends to give less reliable results when the target type is copied from the source and differs from the matched type (as for Record3 in Figure 7-13 where TRANSFER-VALUE changes the partially matched meeting from 3 p.m. to 4 p.m. in the office with Ed to a class from 1 p.m. to 2 p.m. in room 5409).

The virtual effects produced by ALTER for the same database action records but a different set of calendar entries are shown in Figure 7-15. CALENDAR2 contains the same seminar entry as CALENDAR1 in Figure 7-13 but does not contain

```
TRANSFER-VALUE (source record, target record, match record)
IF the type field of the target is unspecified                    (1)
THEN copy the source type
FOR EACH empty target field, DO                                   (2)
  IF the target type and match type are the same                  (3)
    THEN IF the source field is empty or a default and the match field is not empty
           THEN copy the match value to the target
           ELSE copy the source value to the target
    ELSE IF the field is required                                 (4)
           THEN IF the source field is specified
                  THEN copy the source value to the target
                  ELSE copy the match value to the target
```

Figure 7-14: The TRANSFER-VALUE algorithm for preserving information when ALTERing a value in a database record.

CALENDAR1's lunch entry and has different times for the meeting entry. In fact, since r5's *starthr* and *endhr* no longer correspond to those expected by any of the database action records, Record1 is now only partially matched by r5, while Record3 and Record4 have no match at all.

```
CALENDAR2 for (6 5 *)
r4 starthr 1000 endhr 1200 ggtype seminar location (* 5409) general-topic ai
r5 starthr 1600 endhr 1700 ggtype meeting location (* office) participants ed

ALTER (Record1):
(partial r5 (date (6 5 *) starthr 1300 endhr 1400 ggtype meeting
            location (* office) participants ed))

ALTER (Record2):
(partial r4 (date (6 5 *) starthr 1300 endhr 1400 ggtype seminar
            location (* 5409) general-topic ai))

ALTER (Record3):
(error nomatch 5)

ALTER (Record4):
(error nomatch 5)
```

Figure 7-15: The values returned by calls to the database function ALTER for virtual effect for each of the database action records in Figure 7-12 under CALENDAR2.

If we look back at Figure 7-4, the RESOLVE algorithm, we see that the equivalence

classes that were originally created by distinctions in the database action records must ultimately be organized by virtual effect (step (3)). The results returned by ALTER did not change the root's equivalence classes under CALENDAR1—each database action record (representing a unique root) produced a distinct effect. Under CALENDAR2, however, Record3 and Record4 produce the same effect, a "nomatch" error. Figure 7-12 showed the correspondence between roots and database action records. Figure 7-16 shows the pairings of roots to effects under each set of calendar entries.

Roots	Effect under CALENDAR1
(381)	(perfect r3 (date (6 5 *) starthr 1300 endhr 1400 ggtype meeting...)
(382)	(partial r1 (date (6 5 *) starthr 1300 endhr 1400 ggtype seminar...)
(383)	(partial r3 (date (6 5 *) starthr 1300 endhr 1400 ggtype meeting...)
(384)	(partial r2 (date (6 5 *) starthr 1300 endhr 1400 ggtype lunch...)

Roots	Effect under CALENDAR2
(381)	(partial r4 (date (6 5 *) starthr 1300 endhr 1400 ggtype meeting...)
(382)	(partial r5 (date (6 5 *) starthr 1300 endhr 1400 ggtype seminar...)
(383 384)	(error nomatch 5)

Figure 7-16: The equivalence classes of roots for "Change the mtg from 3-4 p.m. to 1-2 p.m." as determined by effect under two sets of calendar entries.

The first three steps in the resolution process are responsible for creating database action records, applying inferences and default reasoning, and mapping completed records into sets of equivalent virtual effects. If these steps result in a single set for all roots then all the explanations produced by the Parse/Recovery Cycle were effectively equivalent. If the roots were at Deviation-level 0 and the effect is not an error, then confidence in the explanation is high enough to simply take the proposed action (step (4) of the RESOLVE algorithm). Under the circumstances where all the roots lead to errors, an error message is chosen and no action is taken (step (5)). The roots for "Change the mtg from 3-4 p.m. to 1-2 p.m.," correspond to neither of these conditions; the roots succeed at Deviation-level 1 and create multiple possible effects (not all of which are errors) under both CALENDAR1 and CALENDAR2. With no more knowledge available to help the system determine the correct action, CHAMP must turn to the user in the final step of the resolution process.

7.4. Confirmation by Effect

By the time it reaches the last stage of resolution, the system knows which roots correspond to possible database actions and which do not. It also knows which of the possible effects resulted from perfect matches against the contents of the calendar (or airline schedule) and which from only partial matches. These two pieces of information make the USERQUERY algorithm (called in step (6) of Figure 7-4) extremely simple: first ask the user to choose from among the effects caused by perfect matches, then, if no choice was acceptable (or no matches perfect), offer the effects caused by partial matches. In this way the system receives confirmation of a subset of explanations by displaying the subset's corresponding effect. Once a single, legal database action has been determined, the appropriate database function is reinvoked for "actual effect."

Figure 7-3 showed the choices offered by USERQUERY for the sentence, "Cancel the mtg June 5 at 3" under three sets of calendar contents. The choices offered by USERQUERY for "Change the June 5 mtg from 3-4 p.m. to 1-2 p.m." under CALENDAR1 and CALENDAR2 are shown in Figure 7-17. In EXAMPLE 4, the user chooses the single effect that has survived the resolution process to this point. Effects corresponding to the partial matches in ALTER for CALENDAR1 would be offered to the user only if she finds the perfect match unacceptable. By confirming the effect, the user causes the change to be made to the calendar. She also unknowingly reduces the number of explanations for the utterance from four to one; only the tree rooted at pc381 is passed along to the adaptation and generalization mechanism.

In EXAMPLE 5, none of the alternatives offered to the user were what she intended by the utterance. Since the choices correspond to partial matches, the system has no options left but to continue to search for new explanations at higher deviation-levels. The continuation message is given when the deviation-level of the roots is less than the maximum deviation-level permitted by the system (two in CHAMP). If the maximum deviation-level has already been reached the user is told instead to "Please try again" (see EXAMPLE 3 in Figure 7-3). When no effect is confirmed by the user, no database action takes place and no root nodes are passed to the adaptation and generalization mechanism.

We pointed out in Chapter 4 that although the user has a single task (making changes to the calendar), CHAMP has two tasks: performing the database actions requested by the user and learning the user's language. The purpose of resolution from the user's point of view (indeed, the purpose of the system) is to produce her intended effect. The purpose of resolution from CHAMP's point of view, however, is both to produce the intended effect and to find the single correct explanation of the utterance. What role should the system's added task play in user interaction? The answer in CHAMP is: none. The interaction is designed to offer the user the most intelligent choices possible by using inference and the

EXAMPLE 4 assumes CALENDAR1 from Figure 7-13:

```
Do you want:
  to change: JUNE 5  3:00pm - 4:00pm MEETING in/at OFFICE with ED
          to  JUNE 5  1:00pm - 2:00pm MEETING in/at OFFICE with ED?
[y or n]: y

Ok, done.
(returning roots: 381)
```

EXAMPLE 5 assumes CALENDAR2 from Figure 7-15:

```
Do you want:
(0) to change: JUNE 5  4:00pm - 5:00pm MEETING in/at OFFICE with ED
          to  JUNE 5  1:00pm - 2:00pm MEETING in/at OFFICE with ED?
(1) to change: JUNE 5  10:00am - 12:00pm AI SEMINAR in/at 5409
          to  JUNE 5  1:00pm - 2:00pm AI SEMINAR in/at 5409?
(2) none of the above
[number between 0 and 2]: 2

Ok, continuing...
(returning roots: nil)
```

Figure 7-17: Examples of user interaction for "Change the June 5 mtg from 3-4 p.m. to 1-2 p.m."

constraints imposed by the contents of the calendar and airline databases. By considering the nature of the interaction from the user's point of view we gain ease of interaction for the user at the cost of guaranteeing a unique explanation for the utterance. Still, the examples in this chapter demonstrate how the process of arriving at the user's intended effect can also significantly reduce the number of explanations that must be considered meaningful. As a result, we know that *the APTs preserved by resolution correspond to the minimum set of explanations generated by the current grammar that capture the effective meaning of the utterance*. It is from this minimum set that user derived forms are constructed through adaptation and generalization.

Chapter 8
Adaptation and Generalization

In Chapter 6 we transformed CHAMP from a bottom-up parser to a least-deviant-first parser by adding error recovery to the system. While error recovery allows CHAMP to accept a larger language than the kernel grammar alone, the average cost of acceptance (and rejection) increases; although we apply constraints at every opportunity, the fact is that search expanded to a non-zero deviation-level is always larger than a grammatical search would be for the same utterance. If we can modify the grammar to recognize a deviant form directly, subsequent encounters with that deviation will not require error recovery at all. Thus, future sentences containing the deviation will succeed at lower deviation-levels, requiring less search. Giving CHAMP the ability to learn new grammatical components transforms the system from a simple least-deviant-first parser to a least-deviant-first *adaptive* parser.

Modifications to the grammar come about after error recovery has formed one or more hypotheses explaining a deviation (see Figure 4-1 at the beginning of Chapter 4). The process of transforming one or more hypothetical explanations (APTs) into an appropriate set of new grammar components is what we mean by *adaptation.* Out of the potential set of component types—lexemes, lexical definitions, wordclasses, steps, forms, form annotation nodes, and formclasses—we must choose the subset that best responds to each type of recovery notation.

To understand the issues involved in adaptation, consider Figure 8-1, the APT produced at Deviation-level 2 for User 1's third sentence in her first experimental session (U1S13). The APT rooted at pc81 contains the four pieces of information that define the *adaptation context* for U1S13:

1. The particular grammatical constituents needed to understand the utterance.
2. The order in which the constituents appeared in the sentence.
3. The particular strategies used to understand the constituents.
4. The recovery actions needed to modify one or more of the strategies.

An extremely conservative approach to adaptation would capture as much of the adaptation context as possible. The result would be a set of highly specialized new components derived, for example, from the steps in boldface in Figure 8-1. Following the structure of the APT, first we would create a new **deleteform** with two ordered steps, 705

and 709', where 709' seeks a new subclass of **m-d-ggforms** with a form whose strategy list contains only 403'. In turn, 403' would seek a new subclass of **meetingforms** with a form whose strategy list contains, in order, step 305, a new step created to look for the tokens **%apostrophe s**, and steps 307, 309, and 316.

U1S13: "Cancel [the] John>'s< speech research meeting on June 9."
Tokens: (cancel john %apostrophe s speech research meeting on june 9 %period)

81: *ROOT* (0 9 deleteforms) (ACT2) 2
  1: **705** (0 0 deletewd)
  70: **709** (1 9 m-d-ggforms) (DGG1) (2 (DELETE 401))
    46: **403** (1 9 meetingforms) (GG1) (1 (INSERT 0))
      17: **305** (1 1 u-personforms) (IFPERSON1)
        2: 118 (1 1 studentname)
      0: **0** (2 3 unknown) nil 1
      19: **307** (4 5 u-subjectforms) (IFSUBJ)
        3: 129 (4 5 subjname)
      4: **309** (6 6 meetingwd)
      23: **316** (7 9 m-dateforms) (DATE0)
        5: 201 (7 7 dtmkr)
        20: 203 (8 9 u-dateforms) (DATE1)
          6: 101 (8 8 monthwd)
          9: 102 (9 9 number)
  10: 799 (10 10 eosmkr)

Figure 8-1: User 1's first sentence and its explanation
(steps in boldface are sources of adaptation in an extremely conservative approach).

This sort of interpretation, corresponding to little more than rote memorization of the utterance, is clearly too extreme. Yet, how much of the information in the adaptation context should we pay attention to? Expressed differently, which of the available pieces of information correspond to the appropriate conditions on usage? Will User 1 always drop the article when the action is **delete** and the object is a **groupgathering**? Will she drop the article *only* under these circumstances? Can the tokens **%apostrophe s** appear anywhere among the prenominal segments of a **meetingform**, or are those tokens always a reliable indication of a preceding unmarked participant? Must the participant always be a student? Must the object always be a **meetingform**? Do we gain anything in terms of reduced search or increased accuracy of interpretation if we can make that discrimination?

What we want to extract from an explained deviation is a set of new grammatical components with two properties: first, the components must be accessible during future parses to understand this sentence *and others like it* directly. Second, and just as important, the new components must not add unduly to the cost of understanding

sentences in which they ultimately play no role. Thus, the main issue in adaptive parsing, as in most kinds of learning, is the issue of generalization.

What we mean by generalization in CHAMP is the transformation of the adaptation context into new grammatical components that capture the correct conditions on usage of the deviation. Our goal is to construct the new components for the grammar without either undergeneralizing or overgeneralizing the recovery. Undergeneralization occurs when a component that can explain a segment of the utterance is inaccessible to the parser. For example, when User 1 types, "Move Anderson seminar on June 10 to room 7220" (U1S19), we might tie the previously unknown token **move** to a form that expects a change in the *location* field. Such an adaptation undergeneralizes the actual conditions on usage, as demonstrated by U1S39: "Move PRODIGY meeting on June 11 to 3-4." By tying **move** to the expectation of a change in location, we made it impossible to understand without further adaptation that **move** may also indicate a change in time.

Overgeneralization occurs when a component that plays no role in the APT for an utterance nevertheless creates search paths during the parse. Suppose we had separate kernel forms for the **change** action for each field in a calendar database record. In other words, CHANGE1 would alter the *location* field, CHANGE2 would alter the *participants* field, and so on. Suppose further that all the CHANGE forms shared a step seeking a **changewd**. If we add **move** to the wordclass **changewd** we will prevent the undergeneralization described previously—the parser will treat **move** as a valid verb for either a change in time or a change in location. The new lexical definition overgeneralizes, however; each time **move** is encountered in the future, the parser will have to consider the token as a potential reference to each **changeform**, despite the fact that the word is used in only two ways.

While some type of generalization is needed, the **move** example shows that any mechanism we choose is likely to include or omit appropriate conditions some of the time. When generalizing from a single example this is to be expected. Since there is likely to be a discrepancy between the user's true idiosyncratic conditions on usage and those we can represent, our goal in choosing an adaptation method cannot be absolute accuracy. Instead, we will strive for a principled trade-off between increasing linguistic coverage and controlling the increase in search that can result from extending the grammar. We achieve our goal by way of three assumptions:

1. The only portion of the adaptation context that is relevant to learning is the constituent in which the recovery notation occurs (the *local context*).
2. The shared hierarchical structure of the grammar enforces the appropriate degree of generalization.
3. When more than one method of adaptation can bring the deviation into the grammar in a reasonable way, the method that minimizes the actual or potential increase in the size and ambiguity of the grammar is preferable.

The first assumption offers some protection from undergeneralization by reducing the set of potentially relevant conditions on usage to those found locally in the deviant constituent. The second assumption offers some protection from overgeneralization by tying the conditions on usage into the relationships already present in the Formclass Hierarchy. The third assumption offers a guideline for choosing among adaptation methods that conform to the first two assumptions. The third assumption can be paraphrased as: keep the size of the search space at Deviation-level 0 under control. To do this we may have to consider not just the immediate effect of an adaptation on search, but also long-term effects that stem from the introduction of learning biases. When we consider the effects of adaptation on search, we often find that CHAMP is better suited to compensate for overgeneralization than for undergeneralization in its learning methods.

The remainder of this chapter explores the co-processes of adaptation and generalization in CHAMP. In the next four sections we examine in detail the specific grammatical augmentations associated with each of the recovery actions. In Section 8.5 we tie the individual types of adaptation together under the control of the ADAPT algorithm. Section 8.6 discusses the potential explosion in the size of the grammar that can occur when the Parse/Recovery Cycle produces multiple, effectively equivalent hypotheses; the explosion is controlled by adding a competition mechanism to the basic process of adaptation. In the last section in this chapter we present a detailed example of the effects of adaptation on both system and user performance.

8.1. Adapting to Substitution Deviations

A substitution is identified by error recovery when there is an unknown word or phrase available to fill an unsatisfied, required step. At first glance the appropriate adaptation for this situation seems simple: create a new lexical definition for the unknown word or phrase as a member of the class sought by the required step. We call this a *liberal* approach to adaptation for substitutions because it ignores the adaptation context completely and relies solely on the Formclass Hierarchy to enforce conditions on usage. Although this is exactly the solution implemented in CHAMP, there are other possible methods for adapting to substitution deviations that are less extreme. It is important to understand what we gain and what we lose by a liberal choice.

To illustrate the trade-offs, let us suppose that we have an explanation requiring a substitution in step *k* of form S1 in Figure 8-2. If we add the new word or phrase to the lexicon as an instance of **class4**, every form containing a step that seeks that wordclass will now recognize the new phrase. This generalization across the grammar is a natural effect of the grammar's shared structure.

In particular, the new phrase will be accepted by S2. Observe, however, that S1 requires

S1	S2
isa class1	isa class1
rnode (i k)	rnode (j k)
strategy	strategy
i class class2	j class class3
k class class4	k class class4

Figure 8-2: An example of potential overgeneralization under the liberal approach. Adding a phrase to **class4** as a response to a substitution in S1 generalizes to S2 despite the critical differences between the forms.

step *i* and S2 requires step *j*—a critical difference between the two forms. Since the hypothetical utterance is explainable by a substitution in S1 (and not by a substitution in S2) we can conclude that step *i* was satisfied and step *j* was not. Thus, the liberal approach ignores part of the adaptation context inherent in the presence of *i* and the absence of *j*. Yet it is precisely the retention of this type of local context that we assumed was important in the previous section.

Ignoring local context eliminates the possibility of capturing a reliable co-occurrence of co-constituents. To the extent that the co-occurrence is a real condition on usage, ignoring it overgeneralizes the conditions under which paths relying on the new component are considered during search. User 1, for example, substitutes the word "move" for a kernel member of **changewd** at the end of her first experimental session. The substitution occurs in the context of ACT3, a **changeform** whose steps seek a **changewd**, a marked **groupgathering**, and a source-target pair. The fifteen subsequent occurrences of **move** in User 1's log file also take place in the context of ACT3. Because CHAMP uses the liberal approach, however, the system simply learns "move" as a member of the class **changewd**; the reliable co-occurrence of "move" with an **m-d-ggform** and **stpform** is lost. Since **changewd** is sought by a step shared among six kernel forms (three for each kernel domain), each time "move" is encountered after adaptation the list of forms it indexes may include all six, instead of just ACT3.

The liberal approach also prevents capturing the interdependence of a substitution deviation and another deviation in the constituent. Consider User 3's tenth sentence during session two (U3S210, Figure 8-3). The APT in the figure is only one of three explanations offered for the utterance at that point in the development of her grammar, but it is the only explanation relevant here (for the other explanations, see Figure 8-17 in Section 8.6). The sentence is explained by a substitution and transposition in ACT3 which expects an imperative form of the **change** command. In this utterance there is a clear relationship between the change in the verb's position and the change in its voice and tense. Unfortunately it is a relationship that is not captured by CHAMP. Learning the phrase "has been changed" as a member of **changewd** divorces the two deviations

and overgeneralizes both the substitution and the transposition. Instead of suggesting only a highly specific set of predictions when the phrase is seen in subsequent interactions, the new lexical definition simply invokes the same predictions in the presence of "has been changed" as those invoked for any **changewd**. After adaptation those predictions include, but are not restricted to, the combined substitution and transposition.

U3210: "Lunch with VC on June 13, 1986 *has been changed* from Station Square to VC Inc."

Abbreviated APT:

```
349: *ROOT* (0 17 changeforms) (ACT3) (2 (SUBST 0 706) (TRANSPOSE (0 80)))
  80: 709 (0 7 m-d-ggforms) (DGG2)
    66: 404 (0 7 mealforms) (GG4)
      1: 313 (0 0 mealwd)
      55: 319 (1 2 m-participantforms) (PART0) ...
      56: 316 (3 7 m-dateforms) (DATE0) ...
  0: 0 (8 10 unknown) nil 1
  86: 711 (11 16 stpforms) (STP1)
    70: 406 (11 13 locsrcforms) (LOCSRC1)
      12: 321 (11 11 sourcemkr)
      21: 304 (12 13 buildingforms) (LOC7)
        13: 133 (12 13 buildingname)
    72: 407 (14 16 loctgtforms) (LOCTGT1)
      15: 322 (14 14 targetmkr)
      49: 326 (15 16 u-locationforms) (IFLOC2)
        17: 114 (15 16 businessname)
  18: 799 (17 17 eosmkr)
```

Figure 8-3: An example of the interdependence of multiple deviations.

A final limitation of the liberal approach to substitution adaptations is that it is insensitive to the value of the referring phrase. In other words, we cannot make discriminations based on the tokens themselves. We can only inherit from the shared structure of the grammar those discriminations already associated with the substituted wordclass.

Is it possible to capture critical differences between forms as conditions on usage, incorporate dependent adaptations into the same derived component, make discriminations based on the tokens themselves rather than their wordclass, and still rely on the shared structure of the grammar for generalization? The answer is "yes" if we adapt by the following procedure (the *conservative* approach):

1. Create a new wordclass, $\mathbf{subst_i}$.
2. Add the new word or phrase to the lexicon as an instance of $\mathbf{subst_i}$.

3. From the form that succeeded via the substitution (the *parent*), derive a new form to add to the grammar. The derived form is identical to its parent with two exceptions: first, the derived form must incorporate any other deviations in the constituent. Second, the strategy list of the derived form contains a step seeking $\mathbf{subst_i}$ in place of the step that required the substitution.

How does the conservative method compensate for the inadequacies of the liberal approach? By creating a new wordclass with only the new phrase as its member, the tokens themselves index the specific set of expectations represented by the derived form. Those expectations capture the local context in the deviant constituent by inheriting all the relationships present in the parent form (class, unodes, and so on) except the ones that gave rise to the substitution and any other deviation. By inheriting the parent's class, generalization occurs naturally through the Formclass Hierarchy; every location in the grammar that seeks a member the new form's class now accepts the new form as a valid way of recognizing a member of that class.

The conservative approach solves some problems, but introduces others. Specifically, it undergeneralizes in a particularly harmful manner. To see how, consider Figure 8-4. The substitution of **mtg** for a recognizable **meetingwd** in the first sentence creates five APTs during the Parse/Recovery Cycle: one for each type of object left active after segmentation (the **flight** object is eliminated because AISys is not a legitimate flight destination). Interaction with the user during Resolution reduces the set to the single APT containing the substitution for a **meetingwd**. Conservative adaptation, in turn, adds GG1' to the grammar. GG1' looks for the token **mtg** specifically—a critical difference with respect to GG1. Note that the liberal approach would simply add **mtg** to **changewd**.

At some future point in time, the user types sentence S*j*. Now *six* interpretations are produced by Parse/Recovery, the extra one stemming from a substitution in GG1' *because the critical difference between GG1 and GG1' is not manifested in the utterance.* Resolution can narrow the field only as far as an equivalence class containing GG1 and GG1'. In turn, conservative adaptation creates a derived form for each parent strategy. We assume that the system is smart enough to detect that the two derived forms are identical (because they posit the same substitution for the step that distinguishes their parents), and add only one of them (GG1'') to the adapted grammar. It is possible, however, to create a slight variation on GG1' through a transposition in a non-required case. Such a change would appear to be an additional critical difference between GG1 and GG1' to any straightforward test procedure, but would not be manifested by the utterances shown. Under those circumstances, derived forms for both GG1 and GG1' would be added to the grammar. In other words, had we chosen to make the example in the figure more complex, we could have created an exponential increase at each stage

Si: "schedule AISys *mtg* on june 3 at 4 pm"
P/R: mtg/meetingwd, mtg/seminarwd, mtg/classwd, mtg/mealwd, mtg/tripwd
Resolution: mtg/meetingwd
Adaptation: mtg/$subst_1$, GG1' seeking $subst_1$

Sj: "schedule AISys *appt* on june 8 at 7 pm"
P/R: appt/meetingwd, appt/seminarwd, appt/classwd, appt/mealwd, appt/tripwd, appt/$subst_1$
Resolution: appt/meetingwd and appt/$subst_1$
Adaptation: appt/$subst_2$, GG1'' seeking $subst_2$

Sk: "schedule AISys [meeting] on june 11 at 11 am"
P/R: delete meetingwd, delete seminarwd, delete classwd, delete mealwd, delete tripwd, delete $subst_1$, delete $subst_2$
Resolution: delete meetingwd, delete $subst_1$, delete $subst_2$
Adaptation: GG1''' deleting meetingwd

Sl: "schedule *taxi* on june 15 at 3 pm"
P/R: taxi/meetingwd, taxi/seminarwd, taxi/classwd, taxi/mealwd, taxi/tripwd, taxi/$subst_1$, taxi/$subst_2$, insert taxi/GG1'''
Resolution: taxi/tripwd
Adaptation: taxi/$subst_3$, TRIP1' seeking $subst_3$

Figure 8-4: An example of the explosive growth in the search space caused by the conservative approach.

rather than a simple linear increase.[38] Note that under the liberal method, the amount of processing required to explain the second sentence would have been unaffected by the adaptation from the first sentence: the same four paths would have succeeded during Parse/Recovery, one explanation would have survived Resolution, and **appt** would have been added to the lexicon as a **meetingwd**.

If we move forward in time to sentence *Sk* we find that the steady increase in processing time caused by the conservative approach may continue via other deviations as well. The figure shows that when the user drops the head noun altogether (a common occurrence), interpretations must be followed for each of *seven* paths despite the fact that three of the paths now correspond to the same effective meaning. Again, a smart algorithm would notice that the same derived component suffices for all three paths in this example and would create only GG1''' in response. Under a liberal approach, however, we would still have had only four paths succeed in Parse/Recovery and only one APT survive Resolution.

[38]Conservative adaptation algorithms for substitution and insertion were implemented in an early version of CHAMP. They produced the same kind of harmful expansion in the grammar (including instances of exponential growth) that is shown in this simplified example.

In the last sentence of the example, another substitution appears, but in a new object. To find the correct explanation we must now generate *eight* hypotheses at Deviation-level 1: seven substitutions and one insertion into the form that no longer requires the head noun. Under the liberal method we generate only five (the original four plus the insertion). Observe that the last incident of adaptation in the figure looks just like the first. Sentence S*l* starts the propagation of effectively equivalent forms in a new portion of the grammar.

In general we characterize the trade-off between the liberal and conservative approaches as one of overgeneralization versus undergeneralization (of course, less liberal or less conservative methods than those discussed would contain some combination of the problems of each the extremes). In CHAMP, the balance tips in favor of overgeneralization when we consider the relative increase in search under each approach. Figure 8-4 demonstrates that CHAMP is poorly suited to compensate for the effects of undergeneralization. When a substitution is tied to a critical difference between a parent and its derived form and that critical difference does not reflect a true condition on usage, future adaptations create multiple paths during Parse/Recovery. The equivalence of the interpretations along the paths cannot be established until the Resolution phase, *after* a lengthy search.

On the other hand, CHAMP is well-suited to compensate for overgeneralization. The system begins with a kernel grammar in which each form contains a critical difference with respect to every other form. The liberal method of adapting to substitutions does not change that situation; adding a new phrase to a wordclass leaves the relationships between the forms unaffected. If the user employs the new phrase in the full range of structures that seek the phrase's class, then liberal adaptation has generalized correctly. Even if the user employs the new phrase in only a subset of the structures indexed by its class, those structures are still critically different from the others in the set; the critical differences guarantee that the unsuccessful forms will be weeded out early in the Parse/Recovery Cycle by COALESCABLE. In short, the shared structure of the grammar permits full generalization by the simple addition of a lexical definition, while the parsing algorithm takes advantage of the actual conditions on usage during search.

8.2. Adapting to Insertion Deviations

An insertion notation is added to an APT by error recovery whenever there is an unknown word or phrase in the utterance and at least one path through the search space with no unsatisfied, required steps. Since the inserted phrase is not tied to a step in the grammar, it has no associated *bindvar* and cannot contribute to the effective meaning constructed for the utterance during resolution. A liberal response to an insertion deviation would simply add the phrase to the lexicon as a member of the wordclass **insert**, then throw away segments in that class in subsequent parses. In this way, the

"insertion" generalizes to all locations in the grammar. The approach has two advantages: first, it is extremely efficient in both its implementation and performance. Second, it is intuitively appealing if the majority of insertions are truly content free.

In what sense might an inserted node carry information if it has no associated **bindvar** to map its tokens into a database field? If the deviation is employed consistently by the user, and only in certain contexts, then its presence could be taken as an indicator of those contexts. By acting as a predictor of a context, the existence of that segment in a future utterance would influence the space of explanations to be explored.

Unfortunately, an insertion deviation usually represents a family of contexts. Recall that the Parse/Recovery Cycle is designed to bubble the inserted node up to the highest constituent that captures it (see Section 6.5). The resulting APT is just a shorthand for a set of hypotheses that are recoverable by a procedure akin to a tree-walk (see Figure 8-5). We cannot know *a priori* which constituent, if any, will be consistently indexed by the inserted phrase. Consequently, to use the presence of the phrase as a reliable indicator of a context, we would have to create critically different new forms for all the interpretations and let future utterances determine the correct one (Section 8.6 demonstrates how this could be done).

It should come as no surprise that a conservative approach to insertion adaptations—one that tries to tie the inserted phrase to critical differences among strategies—carries with it the same type of harmful undergeneralization we saw with conservative substitutions. Moreover, the types of insertions found in the data from our user experiments served as very poor indicators of context, both with respect to forms in the kernel grammar and with respect to forms in each user's adapted grammar. Consequently, CHAMP uses the liberal approach and "throws away" inserted segments during the parse by allowing them to JOIN with any pc (in Section 8.6 it will become clear why we cannot actually remove the phrase from future utterances during segmentation).

8.3. Adapting to Deletion Deviations

A deletion is detected when a required step cannot be satisfied by a segment in the utterance. We label a step "required" in the kernel grammar for one of two reasons: either we believe that satisfying the step makes a strong prediction for the presence of the complete form, or we believe that the value to be bound to the step is critical to understanding. Steps seeking markers are required for the first reason while a step that binds to the object of a verb is required for the second reason. As in the cases of substitution and insertion adaptations, we can contrast liberal and conservative methods for responding to a deletion recovery notation. Unlike the previous cases, however, this time we take the conservative path; the change in attitude stems from a desire to maintain the unique properties of required steps in the grammar.

U5S23: "Cancel class 15-731 on June 16>th< from 1:30-3:00."
Tokens: (cancel class 15 %hyphen 731 on june 16 th from 1 %colon 30 %hyphen 3 %colon 0 %period)

Abbreviated APT:
127: *ROOT* (0 17 deleteforms) (ACT2) 1
  1: 705 (0 0 deletewd)
  111: 709 (1 16 m-d-ggforms) (DGG1') 1
    109: 403 (1 16 classforms) (GG3) (1 (INSERT 0))
      2: 311 (1 1 classwd)
      37: 312 (2 4 classnumforms) (CLNUM1) ...
      55: 316 (5 7 m-dateforms) (DATE0) ...
        9: 201 (5 5 dtmkr)
        49: 203 (6 7 u-dateforms) (DATE1)
          10: 101 (6 6 monthwd)
          13: 102 (7 7 number)
      0: 0 (8 8 unknown) nil 1
      57: 314 (9 16 m-intervalforms) (INT0)
        14: 204 (9 9 ivlmkr)
        51: 206 (10 16 u-intervalforms) (INT1) ...
  28: 799 (17 17 eosmkr)

Hypotheses based on members of the insertion family:
  th indicates GG3: classforms m-dateforms **th** m-intervalforms
  th indicates DATE0: dtmkr u-dateforms **th**
  th indicates DATE1: monthwd number **th**
  th indicates INT0: **th** ivlmkr u-intervalforms

Figure 8-5: The family of insertions represented by the APT for U5S23. The members can be retrieved by a modified infix tree-walk for the subtree rooted at pc109 by considering only leftmost paths in a right subtree and rightmost paths in a left subtree.

Recall from Section 4.2.2 that a step is required by a form if it appears in the form's *rnode* list. Let us assume that the grammar contains a strategy, S1, with a required step, *k*, unsatisfied by the utterance (Figure 8-6(a)). A liberal adaptation would modify S1 directly by deleting *k* from the form's *rnode* list (producing $S1_{new}$ in the figure). Such an action does create a strategy in which *k* is no longer required. At the same time, however, it removes a source of context from the adapted grammar that was considered important when the kernel grammar was designed.

Removing that source of context has two effects: first, it introduces a bias toward overgeneralization in future explanations. To see how, assume we simply remove the step from the *rnode* as in $S1_{new}$. Now suppose that an utterance appears requiring an adaptation of $S1_{new}$. That adaptation occurs either in the presence or the absence of a member of *k*'s class. By having removed the *rnode*, $S1_{new}$ now succeeds at the same

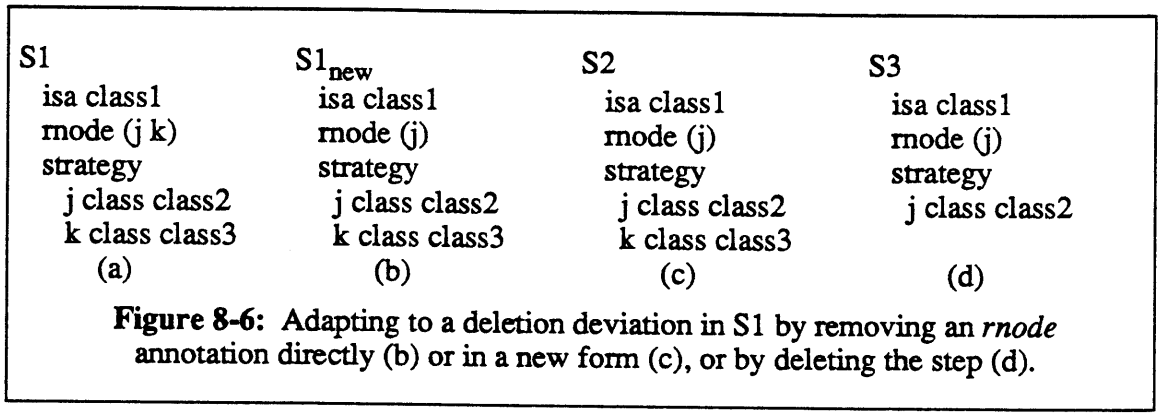

Figure 8-6: Adapting to a deletion deviation in S1 by removing an *rnode* annotation directly (b) or in a new form (c), or by deleting the step (d).

deviation-level under either set of circumstances—any further adaptation of $S1_{new}$ generalizes to both conditions. Since *k* was considered to be strongly predictive in the kernel, we want to be careful about how we remove its predictive capabilities: saying that a member of **class1** can be present without *k* should be different from saying that the presence of *k* is no longer a good indicator of **class1**.

The second effect of removing the *rnode* annotation for *k* also biases learning. If *k* is no longer required, then **class3** can no longer be the target of substitutions. Recall from Section 6.5 that we decided to treat substitution as an appropriate recovery action only for required steps because considering substitutions into all the steps in a strategy would have created an unreasonable number of deviation hypotheses. Thus, once S1 is replaced by a form in which *k* is no longer required, all new phrases that legitimately play the same role as members of **class3** will be learned as insertions. The presence of a segment in the class **insert** produces no constraint on search during the parse. The presence of a segment in any other wordclass produces whatever constraint is inherent in the grammar for that class at the time of use. Thus, removing the possibility of learning the substitution also removes the constraint that substitution might have on search.

A less liberal approach is shown in Figure 8-6(c): we create a new form from S1 without the *rnode* annotation for *k*. Thus, S2 is identical to $S1_{new}$ except that it exists in the grammar in addition to S1 rather than in place of S1. With *k* removed from its *rnode*, S2 succeeds whenever S1 fails because step *k* is unsatisfied; the grammar as a whole has been extended to include constituents of S1's class that do not require *k*. In addition, the extension to the grammar maintains a form in which *k* is a required step so that substitutions into **class3** are still possible. Unfortunately, this solution preserves the context in S1 at the cost of introducing a strategy without a critical difference. Whenever S1 succeeds, so will S2 because S2 can (but need not) recognize the sub-constituent in step *k*. In essence, S2 will introduce the same bias toward overgeneralization in future explanations as did $S1_{new}$.

Figure 8-6(d) presents the conservative approach to deletion adaptations implemented in

CHAMP. The adaptation leaves the *rnode* annotation for *k* intact in S1 and creates a new strategy from S1 that omits step *k* altogether. This approach uses critically different strategies to preserve the separate contexts corresponding to the presence or absence of the deleted step. Generalization occurs through the Formclass Hierarchy because the parent (S1) and its derived form (S3) share the same class; any constituent seeking a subsegment recognizable by the parent will now accept candidates recognized by the derived form as well.

Figure 8-7 shows a ubiquitous example of deletion deviation and subsequent adaptation. Although the sentence is taken from User 7's protocol (U7S19), every user studied eventually dropped the articles from some of her utterances. Two APTs are initially constructed for the utterance by the Parse/Recovery Cycle because **AISys** is defined in the lexicon as a **businessname** (a business may refer to either a location or a set of participants). Resolution finds that only the interpretation of **AISys** as a set of participants matches a calendar entry. Since the APT contains a deviation it is only a hypothetical explanation of the utterance; consequently the user is asked to confirm the system's interpretation despite the fact that there is a perfect match. When confirmation is given, adaptation creates USTRAT0 from DGG1 by deleting the step in DGG1 that seeks the definite article. Note that USTRAT0 inherits from DGG1 all those field values that are not directly affected by the deletion.

```
next> cancel AISys meeting on June 14
P/R: (39 38)

Do you want:
   to delete: JUNE 14   10:00am - 12:00pm MEETING with AISYS ?
 [y or n] y

Creating USTRAT0 from DGG1 via (DELETE)
Ok, done.
```

Parent Form	Derived Form
DGG1	USTRAT0
isa m-d-ggforms	isa m-d-ggforms
strategy (401 403)	strategy (403)
rnode (401 403)	rnode (403)
snode (all)	snode (all)

Figure 8-7: A common type of deletion detection and adaptation.

8.4. Adapting to Transposition Deviations

When the order relation on two steps is violated in an utterance we say that a transposition error has occurred. Since there is direct evidence for the new order and either direct or *a priori* evidence (in kernel forms) for the old order, we could adapt by joining the steps with a *unode* annotation in the deviant form. Alternatively, we could create a new strategy that reflects the transposed order directly. The latter method is implemented in CHAMP.

To see why, let us assume that the user's grammar contains strategy S1 in Figure 8-8(a). We further assume that the user's utterance does not contain the optional constituent in step *j* but does contain constituents *i* and *k*, in the wrong order. Figure 8-8(b) and (c) demonstrate how each solution introduces a different bias into the hypothesis space.

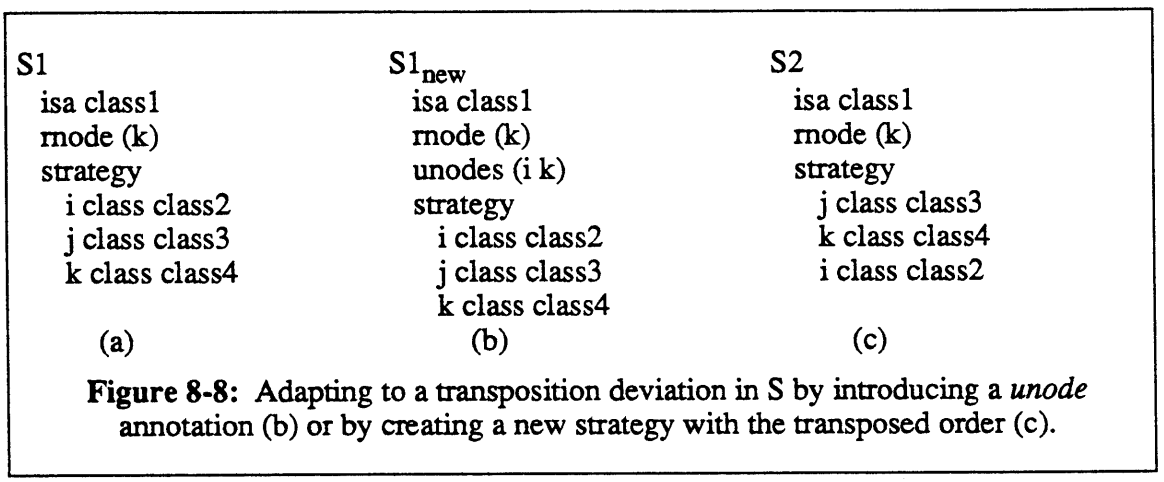

Figure 8-8: Adapting to a transposition deviation in S by introducing a *unode* annotation (b) or by creating a new strategy with the transposed order (c).

In Figure 8-8(b) we reflect the transposition by adding a *unode* annotation between *i* and *k* directly in S1. In Figure 8-8(c) we record the transposition explicitly in a new strategy, S2. Now consider the sets of utterances accepted by (b) and (c). Each solution results in an adapted grammar in which both the order *i*, *k* and the order *k*, *i* are accepted when step *j* is unsatisfied. Only S2 accepts the new order when a constituent for *j* is present, however. For $S1_{new}$ to accept both orderings on *i* and *k* when *j* is present, we would have to add *j* to the *unode* as well, but the utterance gives no evidence for that interpretation. The solution in (c) permits both orderings for which there has been evidence and establishes a critical difference between the parent and derived form when *i* and *k* are present in the utterance. Unfortunately, when *i* is not present both strategies will succeed.

When constructing a derived form for a transposition, other ordering relations in the parent form must be considered. Figure 8-9 shows the transposition learned for U2S91. At Deviation-level 0, CHAMP finds an explanation for the sentence by attaching the marked date to the target seminar description and using the default date provided at the

start of each session (June 5) to infer the source seminar date. Since no calendar entry corresponds to this interpretation during Resolution, CHAMP continues its search into Deviation-level 1. With one deviation point available, the system produces an APT in which the introductory adverbial date has been transposed to the end of the sentence. Because of the *bindvar* in the transposed step, this interpretation maps the date to both source and target during Resolution, allowing the desired calendar entry to be found. Since the APT contains a deviation, the user is asked for confirmation of the effect. When confirmation is given, the calendar is updated and USTRAT17 is created by adaptation, reflecting the new ordering in its strategy list. Observe that USTRAT17 contains another difference with respect to its parent: 703 has been removed from the *unode* inherited from ACT5. In general, the transposed step loses its old ordering relations and takes its new ordering relations from its new position. If the new position is contiguous to ordered steps, the transposition is ordered with respect to those steps. If the new step is inserted between unordered steps, it shares the *unode* annotation of those steps in the new form.

8.5. A Summary of Adaptation in CHAMP

The purpose of adaptation is to bring a deviation into the grammar by deriving new grammatical components from the adaptation context that will parse the deviation directly in future utterances. The particular set of new components that are added to the grammar depends upon which deviations are present in the utterance. The choices implemented in CHAMP are shown in Figures 8-10 through 8-13. In addition to describing the type of component added, each figure also reviews the method of generalization and summarizes the implications of the adaptation for learning and future search.

With respect to generalization, the four figures show that a derived component that recognizes a new way of referring to a particular class of constituents is "inherited upward" through the Formclass Hierarchy to all locations that seek constituents of that class. The main advantage to relying on the Hierarchy for generalization is that it provides a simple, uniform mechanism. The main disadvantage is that making generalizations across established boundaries in the grammar requires discrete episodes of learning. Consider, as an example, that the kernel grammar distinguishes **groupgatherings** marked by definite articles from those marked by indefinite articles by assigning the former to the class **m-d-ggforms** and the latter to the class **m-i-ggforms**. The first time a user drops the article from her utterance, she will do so in one context or the other; CHAMP will learn either a new **m-d-ggform** or a new **m-i-ggform**, but not both. The deletion in the other class must be learned separately because the discrimination provided by the separate classes cannot be ignored by the same mechanism that is relying on that discrimination for generalization.

```
next> change Natural Language Interfaces Seminar to AI Seminar on June 19
P/R roots: (58)
No calendar entry fits that description on June 5.

P/R roots: (73)

Do you want:
  to change: JUNE 19  10:30am - 11:30am
                (NATURAL LANGUAGE INTERFACES) SEMINAR in/at 5409
        to: JUNE 19  10:30am - 11:30am
                AI SEMINAR in/at 5409
[y or n] y

Creating USTRAT17 from ACT5 via (TRANSPOSE)
Ok, done.

Abbreviated APT:
73: *ROOT* (0 10 changeforms) (ACT5) (1 (TRANSPOSE (22 0)))
  0: 706 (0 0 changewd)
  35: 709 (1 4 m-d-ggforms) (USTRAT1)
    27: 403 (1 4 seminarforms) (GG2) ...
  5: 723 (5 5 targetmkr)
  41: 724 (6 7 m-i-ggforms) (USTRAT0)
    32: 403 (6 7 seminarforms) (GG2) ...
  22: 703 (8 10 m-dateforms) (DATE0)
    8: 201 (8 8 dtmkr)
    17: 203 (9 10 u-dateforms) (DATE1) ...
```

Parent Form	Derived Form
ACT5	USTRAT17
isa changeforms	isa changeforms
strategy (701 702 703 706 709 723 724 799)	strategy (701 702 706 709 723 724 703 799)
mode (706 709 723 724)	mode (706 709 723 724)
mnodes ((701 702))	mnodes ((701 702))
unodes ((701 702 703))	unodes ((701 702))

Figure 8-9: An example of deviation detection and adaptation for transpositions.

Figures 8-10 through 8-13 make it clear that CHAMP learns discriminations based on the presence, absence, or position of categories in its kernel grammar. The adapted grammar that evolves for a particular user depends, of course, on the history of interactions with that user. *Since none of the adaptations introduces new constituents into a derived form, however, some portions of the user's natural language may be out of reach.* As an example, consider the following sentences taken from User 10's first session:

Substitution

- Adaptation: a new lexical definition is added to the grammar for the unknown phrase as a member of the wordclass sought by the step requiring the substitution.
- Generalization: occurs through the Formclass Hierarchy. Everywhere the wordclass was previously sought, the new lexeme is recognized.
- Implications for learning: extends the set of indices for discriminations already present in the grammar—those attached to the substituted wordclass. Loses potential discriminations based upon the tokens themselves, or the co-occurrence of the tokens or substituted class with other classes or deviations present in the adaptation context.
- Effect on search: reduces the deviation-level required to understand future occurrences of the lexeme. If the lexeme was already defined in the lexicon, the new definition increases the amount of lexical ambiguity in the system. This may create additional search paths for utterances that contain the lexeme, but has no affect on the search space for utterances that do not contain it.

Figure 8-10: Summary of adaptation to substitutions.

Insertion

- Adaptation: a new lexical definition is added to the grammar for the unknown phrase as member of the special wordclass **insert**.
- Generalization: occurs throughout the grammar. The new lexeme is allowed to occur without deviation anywhere in subsequent utterances.
- Implications for learning: loses potential discriminations based upon the co-occurrence of the tokens with other classes or deviations present in the adaptation context.
- Effect on search: reduces the deviation-level required to understand future utterances containing the inserted phrase. If the lexeme was already defined in the lexicon, the new definition increases the amount of lexical ambiguity in the system. This may create additional search paths for utterances that contain the lexeme, but has no affect on the search space for utterances that do not contain it.

Figure 8-11: Summary of adaptation to insertions.

U10S120: Change June 11 NY to Pittsburgh from flight 82 to flight 265
U10S121: Change from flight 82 to flight 265 on June 11 NY to Pittsburgh
U10S122: Change flight 82 to flight 265 on June 11

User 10's first two attempts to perform the subtask require learning that is beyond the scope of CHAMP's adaptation mechanism. The problematic segment is "from flight 82" which the parser tries to explain as a source in a source-target pair because of the marker "from." No kernel form permits a full flight object in the source (only a flight attribute), and the system provides no way to introduce the possibility into the language. Thus, in U10S120 the system can compensate for the missing head noun ("Change June 11 NY to Pittsburgh [flight]"), and in U10S121 the system can compensate for the transposed date

Deletion

- Adaptation: a new form is added to the grammar. The new form is derived from the form in which the deviation was explained. The new form inherits all the information present in its parent that does not relate to the deleted step.
- Generalization: occurs through the Formclass Hierarchy. The derived form may be used anywhere the parent form is used to recognize a constituent. If two deletions or a deletion and a transposition occur in the same constituent, they are tied together in the same derived form.
- Implications for learning: permits future discriminations based on the presence or absence of a class. Permits discrimination based upon some co-occurrences of deviations.
- Effect on search: reduces the deviation-level required to understand future utterances containing the deletion. May introduce structural ambiguity into the grammar if the deleted step represents a critical difference between formclasses.

Figure 8-12: Summary of adaptation to deletions.

Transposition

- Adaptation: a new form is added to the grammar that explicitly captures the new ordering of steps. The new form inherits all the information present in the parent that does not relate to the ordering of the transposed step. The ordering relations on the transposed step depend on the steps surrounding it after transposition. If the surrounding steps are unordered, the transposed step is added to their *unode* annotation.
- Generalization: occurs through the Formclass Hierarchy. The derived form may be used anywhere the parent form is used to recognize a constituent. If two transpositions or a transposition and a deletion occur in the same constituent they are tied together in the same derived form.
- Implications for learning: permits discriminations based on ordering. Permits discrimination based upon some co-occurrences of deviations.
- Effect on search: reduces the deviation-level required to understand future utterances that reflect the new ordering. If the transposed step is not required, the adaptation may introduce ambiguity into the search space at non-zero deviation-levels whenever the transposed step is not needed to understand the utterance.

Figure 8-13: Summary of adaptation to transpositions.

and origin-destination pair, but in neither instance can the system construct an explanation of the sentence. U10S122 does not contain the misleading marker and is parsed by a kernel form. Giving CHAMP the ability to construct new sets of previously uncombined constituents would be a non-trivial but valuable extension to the system. One mechanism providing that ability is *constructive inference* which is discussed further in Chapter 11.

With respect to search, the summary figures reveal that each type of adaptation may introduce ambiguity into the adapted grammar. Substitution and insertion adaptations may introduce lexical ambiguity if the new definitions act as alternatives to definitions already in the lexicon.[39] The cost of the ambiguity during search depends initially upon the number of additional forms indexed by the new definition. The degree to which the ambiguity propagates through the search space depends upon what other constraining information is available in the utterance.

Deletion adaptations may introduce structural ambiguity into the grammar. Let us reconsider the case of **m-i-ggforms** and **m-d-ggforms**. In the kernel there is only one **m-d-ggform**, DGG1, and only one **m-i-ggform**, IGG1. The forms share the step that seeks the object; the critical difference between the forms is the step in each that seeks the marker. If the user drops both definite and indefinite articles, then the system will derive forms omitting each kind of marker: DGG1' will seek only an object and IGG1' will seek only an object. In other words, any unmarked object will succeed at Deviation-level 0 in the adapted grammar as both a member of **m-d-ggforms** and **m-i-ggforms**. Again, the cost of the ambiguity during search depends initially upon the number of additional forms indexed by the incorrect class assignment. The degree to which the cost propagates depends upon the other constituents in the sentence.

Transpositions may introduce ambiguity at non-zero deviation-levels if the critical difference between the parent and the derived form is in an ordering relation that is not manifested by the utterance. In Figure 8-9, for example, USTRAT17 differs from ACT5 by a transposition of the marked date. The sentence, "change the June 20 seminar to a meeting" does not manifest this difference—it would succeed via both strategies. At Deviation-level 0 the success of both forms has no effect on processing—EXPAND simply creates a pc with both strategies in the *strats* field. On the other hand, if there is a deviation in the constituent, ISCOMPLETE will build a separate modified context for each strategy (see Section 6.3). Since the separate contexts then EXPAND to distinct sets of higher level constituents, this type of ambiguity will always propagate through the search space.

It is important to note that even though each type of adaptation may increase the amount of ambiguity in the current grammar, the future search space for utterances containing the deviation is always significantly smaller than it would have been had we not adapted. The reason is simple: any ambiguous paths at Deviation-level 0 are also part of the much larger search space at higher deviation-levels. Of course the cost of parsing some utterances that were in the grammar prior to the adaptation will have increased, but three

---

[39]In CHAMP, it is impossible to introduce a new lexical definition for a word or phrase that is already defined without employing extra-linguistic conventions. The method of compensating for the *lexical extension problem* is discussed in Chapter 9. Other aspects of the problem are discussed in Chapter 10.

factors make the trade-off worthwhile. First, since adaptations reflect preferred forms of expression, utterances relying on the adaptations are likely to reappear—the more often an adaptation is reused the more favorable the trade-off becomes. Second, adaptation brings more of the user's language into the grammar, resulting in fewer rejected parses over time. Since rejecting a sentence requires a search through Deviation-level 2 *plus* the search associated with understanding any subsequent rephrasings, any action that prevents rejections must reduce search overall. Finally, since the frequent user's language is self-bounded, whatever increase in ambiguity results from adaptation must be bounded as well.

Figure 8-14 displays the simple algorithm implementing adaptation in CHAMP: each hypothetical explanation for the user's utterance has its annotated nodes transformed into the grammatical components appropriate for that recovery notation. Recall that every APT passed to ADAPT corresponds to the single effective meaning confirmed by Resolution; if more than one root node is passed to ADAPT, each APT must differ from the others in some aspect of its explanation of the utterance. Only part of the explanation need be different, however. Thus, as we traverse an APT looking for subnodes with recovery notations, we must avoid creating new components for a given context if that context is shared by a previously traversed tree (step (1a)).

```
ADAPT (roots)
FOR EACH root passed on by Resolution, DO
  Descend through the tree for this root and                          (1)
    FOR EACH subnode that contains one or more recovery notations, DO
      IF the adaptation doesn't duplicate components created for another root   (1a)
      THEN perform the adaptation required by the recovery notation(s) and
             associate the result of the adaptation(s) with the subnode
      ELSE associate the result of the original adaptation(s) with the subnode
  Associate the new components with the root                          (2)
  FOR EACH derived component used to explain the sentence, DO         (3)
    IF the component required no adaptation
    THEN try to resolve old competitions associated with the component
IF more than one root node created new components                     (4)
THEN set the components from different roots in competition
```

Figure 8-14: CHAMP's ADAPT algorithm.

When there are multiple hypotheses explaining an utterance, each root APT is likely to correspond to a different set of new grammatical components after adaptation. At least one set of components must be added to the grammar to bring the current sentence into Deviation-level 0, but which set of components should we add? If we choose arbitrarily we may choose the least useful set, adding to the grammar adaptations that may increase ambiguity without significantly increasing the system's explanatory power. If we simply

add all the components to the grammar then every sentence that is structurally similar to the current one will always succeed in multiple ways. Since each successful path corresponds to the same meaning, this approach is even more likely to increase ambiguity without increasing explanatory power. What we need is some method of delaying our decision until we have more evidence about which set of changes to the grammar is the most useful. In CHAMP, that method is called *competition*. When the current utterance lends equal support to more than one set of changes, we add all the changes to the grammar as competitors (step (4)). We use subsequent utterances that rely on the derived components to resolve the competition (step (3)).

8.6. Controlling Growth in the Grammar through Competition

Competition helps attain the goal of maximal coverage with minimal ambiguity by delaying commitment to a single set of changes until information is available to make an intelligent choice. The competition mechanism is based on a simple idea: just as one sentence is enough to cause a set of competitors to be added to the grammar, so one sentence is enough to cause a subset to be removed. To preserve the relationships established by multiple APTs, we create a competition relation between grammar components from different APTs and a support relation between grammar components from the same APT. Figure 8-15 illustrates the process.

With respect to the kernel grammar, the first sentence in the figure (U7S118) contains one unknown segment (''what is my''), and two deletions (a verb and a definite article). Due to its position in the utterance, the unknown segment may fill either deleted step; thus, U7S118 creates two APTs during the Parse/Recovery Cycle. Since the APTs correspond to the same effective meaning, both are preserved by Resolution. The first explanation produces two new grammar components during adaptation: the lexical definition for the segment ''what is my'' as a **defmkr** compensates for the unknown segment and the deletion of the article, while USTRAT5 compensates for the deletion of the verb. The second explanation for U7S118 uses the unknown segment as a substitute for the missing verb and captures the deleted article in USTRAT6. Since only one set of components is required to parse the sentence, but we do not know which set will prove more useful, the last step in adaptation is to place the two sets of components in competition.

In contrast to the competitive relationship between the sets of components created for different APTs, the individual components within a competitor are said to *support* each other. In Figure 8-15, for example, the **defmkr** definition and USTRAT5 support one another. The support relationship reflects the fact that both components are required to explain the current sentence. If a component wins a permanent place in the grammar through competition, it also wins a place for any components it supports. In this way we

guarantee that reducing the grammar through competition resolution does not cost us the ability to understand a previous utterance.

U7S118: "what is my schedule between 9:00 and 12:00 on JUne 12"

Parse/Recovery and Resolution:
APT substituting "what is my" for article and detecting showwd deletion
APT substituting "what is my" for showwd and detecting article deletion

Adaptation:
(what is my)/defmkr
USTRAT5 from ACT6 via deletion of showwd

(what is my)/showwd
USTRAT6 from MCALSEG1 via deletion of defmkr

Competitors: ((what is my)/defmkr & USTRAT5) vs.
((what is my)/showwd & USTRAT6)

U7S41: "please list schedule for june 12"

Parse/Recovery and Resolution:
APT using USTRAT6 and treating "please" as an insertion

Adaptation:
(please)/insert
remove USTRAT5 and (what is my)/defmkr

Figure 8-15: A simple example of the introduction and resolution of a competition.

The four new components remain in the grammar until User 7's fourth session. When she types U7S41, User 7 creates the situation that decides the competition. As the figure shows, Parse/Recovery produces an explanation of U7S41 at Deviation-level 1 by using USTRAT6 as a non-deviant explanation of the missing article and an insertion deviation to explain the unknown segment "please." When the APT is processed by ADAPT, step (3) of the algorithm notices that USTRAT6 is a derived form that succeeded without recovery notations. Since USTRAT6 has an associated competition relation, the APT is examined for evidence of each competitor. In the single APT for U7S41, only USTRAT6 is present. We now have available additional information to decide which explanation to prefer for U7S118: the combination of USTRAT6 and "what is my" as **showwd** helps to understand both utterances, whereas the combination of USTRAT5 and "what is my" as **defmkr** has a more limited use. Since the former combination subsumes the explanatory power of the latter (given the evidence to this point) we preserve only USTRAT6 and the component it supports; its competitor is removed from the grammar. Notice that the system preserves the definition of "what is my" as **showwd** despite the fact that it does not occur in U7S41 because it is supported by USTRAT6.

Competitions are sometimes resolved despite the fact that no new information is available. This learning bias stems from an interaction between the competition mechanism and the Maximal Subsequence Heuristic (Section 5.2). Figure 8-16 illustrates the problem. As shown, U4S15 causes adaptations analogous to those produced in response to U7S118 in Figure 8-15 (two explanations corresponding to alternative substitutions and deletions). The competition set up by U4S15 should not be resolved by U4S31: the two sentences are structurally identical with respect to the portions in competition.[40] Therefore APTs incorporating both competitors should succeed. Only one APT is constructed, however, because the constituent that picks up "commuting time" as a **tripwd** is formed on a lower Agenda-level than the constituent that would use the segment as a **defmkr**. Specifically, at Agenda-level 3 USTRAT3 creates a **tripform** without using the segment "commuting time" (because the head noun is not required in its subcontext). A continuation of this path would pick up the unused segment as a **defmkr**. At the same Agenda-level, however, TRIP1 coalesces "commuting time" as a **tripwd**. Since only the maximal segment is EXPANDed, no path is created that requires the segment as a **defmkr**. At Agenda-level 4, USTRAT2 EXPANDs the maximal subsegment as a marked trip without noticing the absent marker. During adaptation, the success of USTRAT2 and "commuting time" as **tripwd** without the simultaneous success of either USTRAT3 or "commuting time" as **defmkr** in another APT eliminates USTRAT2's competitors from the grammar.

It would be wrong to infer from the examples in Figures 8-15 and 8-16 that CHAMP's competition mechanism always preserves the components that correspond to linguistically correct analyses of the sentences. Figure 8-17 shows a complex sequence of adaptations and competition resolutions that provides a counterexample. We include the first sentence in the example (U3S110), despite the fact that it creates no competition, because USTRAT1 (a form that does not require an **addwd** for the **schedule** action) contributes to the outcome of a competition further on.

The second sentence in the figure (U3S210) is highly ambiguous at Deviation-level 2. Let us refer to the competitors that are created in response to this sentence by the derived form they incorporate. Thus, C4 incorporates USTRAT4 and corresponds to the explanation that "has been changed" is a substituted and transposed **changewd**. C5 explains U3S210 with the same substitution for the verb but considers the **groupgathering** to be the transposed element. C6 pairs a deletion of the verb with an insertion of the unknown segment. C7 deletes the verb in a kernel form that seeks a **slotform** rather than a **groupgathering** (ACT4) and substitutes "has been changed" for a valid **locationslotwd**.

---

[40]CHAMP offers the spelling correction of "commuting time" to "comuting time" for U4S31 during segmentation. When the correction is accepted, the system asks if it should remember the corrected version as well. An affirmative response equates the correct spelling with the original misspelling learned for U4S15.

U4S15: "change comuting time from CMU to Aisys on june 12 from 12p.m. to 2p.m."

Parse/Recovery and Resolution:
APT substituting "comuting time" for tripwd and detecting article deletion
APT substituting "comuting time" for article and detecting tripwd deletion

Adaptation:
(comuting time)/tripwd
USTRAT2 from DTRIP1 via deletion of defmkr

(comuting time)/defmkr
USTRAT3 from TRIP1 via deletion of tripwd

Competitors: ((comuting time)/tripwd & USTRAT2) vs.
((comuting time)/defmkr & USTRAT3)

U4S31:"cancel commuting time from CMU to Aisys on June 12 at 2 p.m."

Parse/Recovery and Resolution:
APT using USTRAT2 and (comuting time)/tripwd succeeds at Deviation-level 0

Adaptation:
remove USTRAT3 and (comuting time)/defmkr from grammar

Figure 8-16: An example of a competition resolved by maximal subsequences.

U3S311 resolves the competition set up for U3S210. Thirteen APTs are produced by the Parse/Recovery Cycle for the sentence, but only three of the APTs survive Resolution. One uses USTRAT6 (which requires no verb), a substitution of "reschedule" for the **defmkr** required by the kernel form DGG1, and a substitution of "to be from" for the **targetmkr**. A second explanation uses a substitution of "reschedule" for the **changewd** still required by ACT3, a form derived from DGG1 that does not require the article, and the **targetmkr** substitution. The final explanation uses USTRAT6 and the form derived from DGG1 to treat "reschedule" as an insertion, and the **targetmkr** substitution to explain "to be from." Since USTRAT6 was used without further adaptation to help explain U3S311, and none of its competitors contributed to an alternative explanation, the competitors are removed from the grammar. This means that the only explanation left in the grammar for U3S210 is the one in which the verb was considered deleted and "has been changed" was considered a meaningless insertion. Although this explanation is inaccurate from a linguistic point of view, it does preserve the most useful subset of adaptations for the two examples.

In addition to resolving an old competition, U3S311 sets up a new competition among its explanations. The new competition is resolved by U3S345, which uses the word

U3S110: "Dinner with Dad June 14, 1986 at 6:00."

Adaptation:
USTRAT1 from ACT1 via deletion of addwd
USTRAT2 from DATE0 via deletion of datemkr

U3S210: "Lunch with VC on June 13, 1986 has been changed from Station Square to VC Inc."

Adaptation:
(has been changed)/changewd
USTRAT4 from ACT3 via transposition of changewd
USTRAT5 from ACT3 via transposition of m-d-ggform
(has been changed)/insert
USTRAT6 from ACT3 via deletion of changewd
USTRAT7 from ACT4 via deletion of changewd
(has been changed)/locationslotwd

Competitors: ((has been changed)/changewd & USTRAT4) vs.
((has been changed)/changewd & USTRAT5) vs.
((has been changed)/insert & USTRAT6) vs.
((has been changed)/locationslotwd & USTRAT7)

U3S311: "Reschedule Craigs meeting on June 11, 1986 to be from 3:00 to 4:00."

Adaptation:
(to be from)/targetmkr
reschedule/defmkr
reschedule/changewd
reschedule/insert

remove USTRAT4, USTRAT5, USTRAT7, (has been changed)/changewd and (has been changed)/locationslotwd

Competitors: (reschedule/defmkr & (to be from)/targetmkr) vs.
(reschedule/changewd & (to be from)/targetmkr) vs.
(reschedule/insert & (to be from)/targetmkr)

U3S345: "reschedule a meeting with Jaime at 2:00 on June 12th."

Adaptation:
remove reschedule/changewd and reschedule/defmkr

Figure 8-17: Using competition to control the growth in the grammar caused by multiple hypothetical explanations.

"reschedule" as an **addwd** (U3S311 used it as a **changewd**). The existence of USTRAT1 (deleted **addwd**) and the definition of "reschedule" as an insertion (along

with other prior adaptations) allow the sentence to be understood correctly at Deviation-level 0. Even though no deviations occurred, the APT for the utterance is passed to adaptation to see if old competitions can be resolved. As a result, the prior definitions of "reschedule" as a possible **changewd** or **defmkr** are removed from the grammar. The only definition consistent with both U3S311 and U3S45 at Deviation-level 0 in the adapted grammar is "reschedule" as an insertion.

The example in Figure 8-17 also serves to illustrate why we cannot simply remove segments in the class **insert** from the utterance during segmentation. By removing the segment, we circumvent any competition relations attached to the insertion; the insertion interpretation and any supported components effectively win without having to compete. Since the insertion's competitors are probably not insertions themselves, they are adaptations that index expectations in the grammar, expectations that act as search constraint. Thus, leaving the segment in future utterances permits the more meaningful adaptations originally associated with the segment to compete for a place in the grammar.

Competition in CHAMP is actually just a type of generalizing from multiple examples. While a competition may be fully resolved by a single sentence, it is also possible that more than one sentence will be required to eliminate all but one competitor. It is even possible that a subset of competitors will never be removed from the grammar; they may, for example, all share a support relation with a common component that co-occurs any time any of the other components is present. At the other extreme, it is possible (although very rare in our experiments) to learn a component, remove it through competition, and then have to learn it again.

In essence, both the advantages and disadvantages of using competition as a method of controlling grammar growth stem from the mechanism's decisiveness: a single piece of evidence favoring one competitor is considered enough to remove its rivals from the grammar. A quick decision is an advantage if the competitors contribute a great deal of ambiguity to the search—but the same decision can be a disadvantage if it removes a set of components that would have eliminated search in the future. There is no simple, generally correct way to prevent this problem within the mechanism described. In the next chapter we examine the degree to which competition is an effective means of controlling growth in six idiosyncratic grammars. We postpone a discussion of alternative mechanisms to Chapter 11.

8.7. The Effects of Adaptation on Performance

With a clear understanding of how learning is accomplished in CHAMP, we can now examine how adaptation improves the system's performance in the task of understanding the user, as well as the user's performance in the task of keeping an on-line calendar. To begin, let us consider Figure 8-18 which contains a small subset of User 3's sentences, none of which can be understood at Deviation-level 0 using the kernel grammar. Two of the ten sentences can be parsed at Deviation-level 2 (S14 and S210), but the remainder contain three or more deviations with respect to the kernel. If we consider these utterances as delineating a portion of User 3's language space, then CHAMP's calendar kernel covers none of that portion.[41] Even with error detection and recovery for two deviations, the system's coverage increases to only twenty percent.

S14 "Cancel John's Speech Research Meeting at 9:00 on June 9, 1986."
S210 "Lunch with VC on June 13, 1986 has been changed from Station Square to VC Inc."
S33 "Lisp Tutorial class will be held in Room 8220 on June 11, 1986."
S311 "Reschedule Craigs meeting on June 11, 1986 to be from 3:00 to 4:00."
S314 "Schedule time for Jill in the User Studies Lab from 3:00 to 4:30 on June 12th."
S49 "Change Andy's lunch on June 12th to begin at 12:30."
S71 "Speech Project on June 17th will be held in room 7220 instead of 8220."
S716 "Class 15-731 will be held in Room 7220 on June 23rd."
S91 "reschedule AI Seminar to begin at 9:00 instead of 9:30 on June 19th."
S93 "Class 15-731 will be held in Room 5409 instead of 7220 on June 23rd."

Figure 8-18: A subset of User 3's utterances delineating a portion of her language space. None of the utterances can be parsed directly with the kernel grammar.

In terms of the kernel system's performance during processing for these ten sentences, observe that each sentence requires a complete search through Deviation-level 2 before it can be accepted or rejected. The situation is more serious from the user's point of view—her performance is affected not only by the system's response time (slow for all the utterances), but also by the total number of interactions required to perform the task. For the sample in Figure 8-18, she must generate at least one alternative phrasing eighty percent of the time.

The next two figures demonstrate how adaptation improves the situation. Figure 8-19 shows some of the components added to the grammar as each utterance is encountered in turn (only the winning set of components is shown for sentences that produce competitions). The column labelled "Relies On" lists the sentences from our sample that

[41] Figure 8-18 is only a slight exaggeration of the real situation; actually the kernel is able to parse seven percent of User 3's utterances at Deviation-level 0 (see Section 9.1.4).

earlier produced new components used to understand the current utterance. The frequent appearance of previously derived components in the explanations of subsequent utterances clearly demonstrates adaptation's bootstrapping effect. Although eight utterances were originally unparsable, Figure 8-19 shows that with adaptation each sentence is within two deviations of the extant grammar when it is encountered.

Sentence	Relies On	Produces
S14		DGG1' derived from DGG1 by deleting **defmkr** (%apostrophe s)/insert added to lexicon
S210		ACT3' derived from ACT3 by deleting **changewd** (has been changed)/insert added to lexicon
S33	S14, S210	ACT3'' derived from ACT3' transposing marked date (will be held in)/targetmkr added to lexicon
S311	S14, S210	reschedule/insert added to lexicon (to be from)/targetmkr added to lexicon
S314		th/insert added to lexicon
S49	S14, S314	(to begin at)/targetmkr added to lexicon
S71	S14, S210, S33, S314	LOC6' derived from LOC6 by deleting **roomwd** (instead of)/sourcemkr added to lexicon
S716	S14, S210, S33	rd/insert added to lexicon
S91	S14, S210, S311, S314, S71	
S93	S14, S210, S33, S71, S716	

Figure 8-19: An example of bootstrapping in User 3's grammar. (The second column mentions only sentences from our sample. The third column shows only the winning set of competitors.)

Figure 8-20 shows the bootstrapping effect more clearly. In the first column the ten sentences are partitioned according to their degree of deviation with respect to the kernel (K). The remaining columns show the continual repartitioning of User 3's language space as each sentence is understood in turn. The second column, for example, shows that after adaptation and generalization for S14, three utterances that would have been unparsable are now within the reach of error recovery: understanding S33 and S311 at Deviation-level 2 requires the form DGG1' that was produced by S14, while understanding S49 at

Deviation-level 2 requires S14's lexical definition of **%apostrophe s** as a member of the class **insert**. After the first eight sentences in the sample have been understood, all of the original utterances are directly parsable by the grammar. More importantly, a great many variations of those sentences, in the same "style" as the originals, are also at Deviation-level 0 due to generalization.

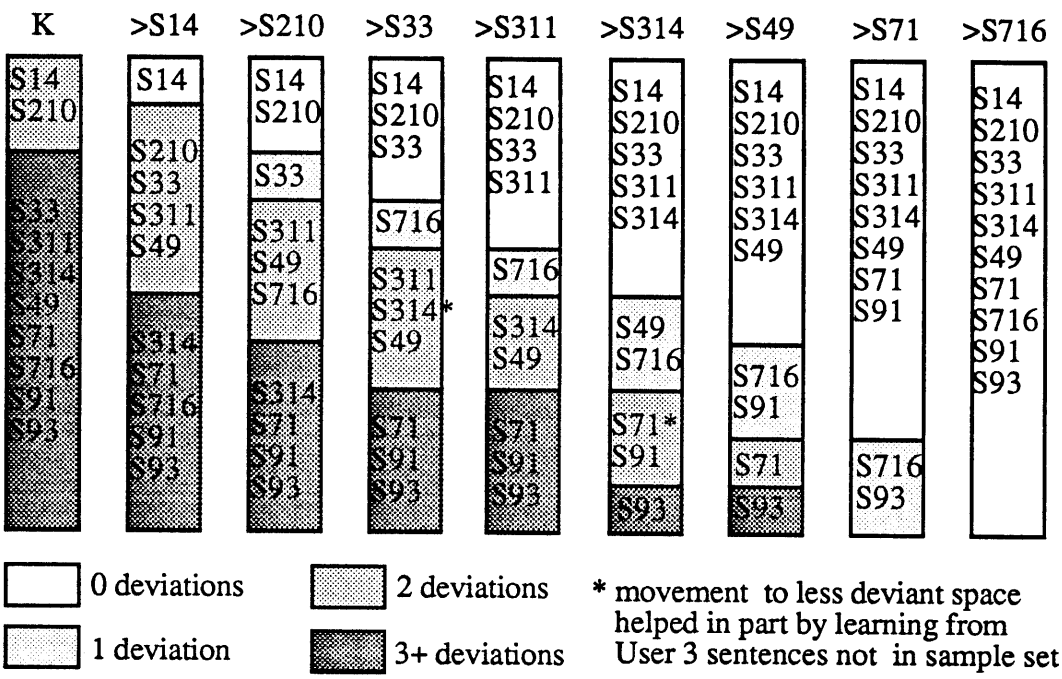

Figure 8-20: The effect of adaptation on the shape of the search space for a portion of User 3's language: as each deviant sentence is brought into the grammar, its derived components reduce the amount of search needed to understand future utterances.

Without adaptation, CHAMP's performance for the sample sentences would have been quite poor, requiring search through Deviation-level 2 for all ten sentences and additional search after rejection for eight. With adaptation, we see an enormous improvement, both in the amount of search required to process the utterances (only half the sentences require search at Deviation-level 2) and the number of utterances accepted (all ten). It is possible to argue, however, that the performance of a non-adaptive interface with a large grammar would be as good or better than CHAMP's adaptive performance. The validity of this argument depends upon two factors: first, whether or not the larger grammar corresponds to the user's forms of expression, and second, the user's ability to adapt to the subset of language provided.

Given the variety of expression in Figure 8-18 we must question how good the initial correspondence between the user's language and the static system's language is likely to be. With poor correspondence, performance values are dominated by the number of parse

failures and required rephrasings. Although it is impossible to predict in the abstract how many rephrasings will be needed by a given interface, observe that for the subset of User 3's expressions we have examined CHAMP required none.[42] Of course, to minimize rephrasings, static interfaces rely on both their large grammars and the *user's* ability to adapt—to continue our comparison we assume that the user can adjust to the interface's demands (despite evidence to the contrary presented in Chapter 3).

The Regularity Hypothesis tells us that in the course of adjusting to the non-adaptive system's language, the frequent user comes to rely on those utterances that she remembers as having worked in the past. Regularity on the part of the user causes a dilemma for the static interface: if the grammar is large enough to permit a reasonable subset of "natural language" for every user, then the Regularity Hypothesis implies that each particular user is likely to employ only a fraction of that grammar. At the same time, each user must continually pay the search costs over the parts of the grammar she does not use. How does this compare with CHAMP's behavior, where both the user *and* the system adapt? Since the initial kernel is small and the grammar is expanded only in direct response to user expressions, the Regularity Hypothesis tells us that relatively little inefficiency results. In short, when the grammar contains primarily what has worked in the past and the user relies on primarily what has worked in the past, the average cost of an interaction in an adaptive system is likely to be lower than the average cost of an interaction in a large static system.

We began this chapter by claiming that we could improve the interface's performance if we turned the simple least-deviant-first parser from Chapter 6 into a least-deviant-first adaptive parser. Believing that deviations reliably indicate idiosyncratic preference, we have chosen to augment the grammar with new components in response to the explanations of a deviation produced by error recovery. By bringing the deviation into the grammar, future instances of the deviation no longer require extended search. In addition, by tying the new component into the Formclass Hierarchy, the explanation of the deviation generalizes to contexts other than the one which gave rise to the adaptation. The limited performance analysis in this section demonstrates that our method of adaptation and generalization can greatly improve the interaction between the user and system in two ways. First, by reducing the number of interactions required by the user to accomplish her task, and second, by reducing the average amount of search required by the system for each interaction.

This evaluation nevertheless leaves a great many questions unanswered. How does

---

[42]The actual number of User 3's sentences that CHAMP could not parse at the time of use was 32/138 (23%). Of these, 20/32 were sentences that would have been accepted by a perfect implementation of the adaptation model described in Chapter 2. Thus, with ideal adaptation, only 12/138 (9%) of User 3's utterances would have required rephrasing.

CHAMP perform on User 3's full data set? How does CHAMP's performance compare to the performance of the ideal adaptive system specified by the model in Chapter 2? Is the sort of improvement in performance shown here characteristic across users? Is our assumption that deviations reliably indicate idiosyncratic preference true (in other words, are the components added by adaptation reused)? Is our prediction that an idiosyncratic grammar requires less search on the average than a large static grammar demonstrable? In the next chapter, we answer these questions and others through a detailed analysis of CHAMP's performance for six idiosyncratic grammars.

Chapter 9
Evaluating the Interface

The hidden-operator experiments described in Chapter 3 tested the underlying behavioral assumptions of our model for language adaptation. The experiments demonstrated that an ideal realization of the model could:

1. Capture the within-user consistency that arises from the frequent user's natural, self-bounded language.
2. Accomodate the across-user variability arising from idiosyncratic preferences with a single, general, recovery and adaptation mechanism.
3. Provide a more responsive environment than that provided by a static interface with its fixed subset of natural language.

Recall, however, that the model itself leaves many aspects of any implementation unspecified; it makes no commitment to a particular generalization method, set of knowledge structures, grammar decomposition, or set of inference procedures. Nor does the model require us to confront the computational complexities that lead to the Single Segment Assumption or the Maximal Subsequence Heuristic.

To implement the model in a working interface the unspecified and underspecified must be made concrete. In the previous five chapters we examined one possible implementation, CHAMP, incorporating one possible set of decisions and trade-offs. It is natural to ask how well CHAMP realizes the ideal, or, equivalently, how the implemented system's performance compares to that of the ideal system simulated in the hidden-operator experiments. This comparison is the subject of Section 9.1.

It is also useful to evaluate the system on its own, asking how well CHAMP performs in real user interactions. Section 9.2 examines the system's performance in on-line experiments with two new users (User 9 and User 10). The new experiments extend our sample size without qualitatively changing the results found for the original data.

Overall, our analysis reveals that CHAMP is a good approximation of the model in terms of capturing within-user consistency and accommodating across-user variability. The system provides a slightly less friendly environment than the model predicts due to its higher rate of rejected sentences, but still provides a more responsive environment than does a fixed, monolithic grammar.

9.1. CHAMP's Performance on Data from the Hidden-operator Experiments

A full description of the hidden-operator experiments is given in Chapter 3. To summarize briefly: users were asked to perform a calendar scheduling task for a fictitious professor/entrepreneur. Each user participated in multiple sessions (three, five, or nine) responding to ten to twelve pictorial subtasks each session. Users in the Adapt and Adapt/Echo conditions of the experiment interacted with a simulated adaptive interface based on the model in Chapter 2 and a grammar that was functionally equivalent to CHAMP's kernel (with a few exceptions, as noted in the next section). Users in the No-Adapt condition interacted with a simulated interface that permitted two-deviation recovery but no adaptation.

The kernel grammar used in the hidden-operator experiments was constructed by simply writing down an informal, BNF-style grammar with one or two forms for each type of action and object required by the stimuli. CHAMP's calendar kernel consisted of functionally equivalent structures represented via steps, forms, and formclasses. The transformation from the BNF representation into CHAMP's caseframe grammar took into consideration the patterns and regularities in the *development set*: eighty-five sentences drawn from the log files for User 1 and User 2 and pertaining only to the calendar domain. Once a working version of the system had been constructed, CHAMP's kernel grammar for the travel domain was developed; the design of the kernel travel grammar was based only on the travel grammar in the experiments and the representational scheme already established for the calendar domain.

To evaluate CHAMP's performance on the data from the hidden-operator experiments, the system was tested with the 657 sentences from the log files for users in the adaptive conditions (User 1, User 2, User 3, User 4, User 5, and User 7). The 657 sentences in the test set include the eight-five sentences in the development set.

One major change to the system occurred after evaluation began: replacement of conservative adaptations with liberal adaptations for substitution and insertion deviations. The change was in response to the severe undergeneralization that arises from the conservative approach for those types of deviations (see Sections 8.1 and 8.2). It is interesting to note that the problem did not become apparent from working with the development set alone. All measures reported here, and in Appendix C, reflect the mixed liberal-conservative adaptation and generalization method described in Chapter 8.

In the next section we examine the differences between CHAMP and the system simulated during the experiments, discussing the ways in which we compensated for those differences in evaluating CHAMP's performance on the test set. In the remaining subsections of Section 9.1, we look at a number of performance measures, including a

comparison of the learning curves for the simulation versus the implementation, an analysis of the reasons for the increased rejection rate in the implementation over the model, and quantification of within-user consistency and across-user variability.

Although CHAMP was tested on the complete set of 657 sentences, our analysis concentrates on CHAMP's performance for the 549 sentences from the log files of the four users in the Adapt condition (User 1, User 2, User 3, and User 4). Each of these users participated in nine experimental sessions, saw the full stimuli set, and had the same opportunities for idiosyncratic expression. In contrast, User 5 and User 7 (Adapt/Echo condition) participated in three and five sessions, respectively, making their data difficult to contrast with the data from Users 1 through 4. The interested reader will find the data for User 5 and User 7 (along with raw data for all users presented in this chapter) in Appendix C. For the remainder of this section, when we refer to the "test set" we mean the 549 sentences from the four users in the Adapt condition.

9.1.1. Differences between CHAMP and the Simulated System

In the process of developing the system it became apparent that there would be sentences accepted during the experiment that CHAMP would be unable to parse as well as sentences rejected during the experiment that CHAMP could understand. To make a meaningful comparison of CHAMP's performance to the ideal performance possible, some way of compensating for these differences had to be found. To understand why, consider that in an adaptive environment the outcome of every interaction may be felt in future interactions: the user avoids forms that lead to rejection and relies on those that are accepted. If a sentence accepted by the hidden-operator lead to adaptations during the experiment but CHAMP cannot accept the sentence, then CHAMP cannot learn the adaptations. If CHAMP does not learn the adaptations then all future sentences relying on those adaptations are likely to be rejected as well. Thus, when a set of adaptations rely on each other (the "bootstrapping effect" we saw in Section 8.7), a difference in learning early in the protocol may be magnified unrealistically. Had the user received negative reinforcement immediately, she would not have been as likely to reuse the form in the future, and the effects of the missing adaptations would not have propagated.

Differences between the simulated system and the implemented system stem from one of three sources:

1. Differences in functionality:
    - User 1 was given the opportunity to create extremely simple macro definitions based on subsequences of events. For example, after three occurrences of "cancel flight *k*" followed immediately by "schedule flight *n*" where the two flights had the same origin and destination, the user was asked if there was a verb that captured this idea. This allowed her to define a "reschedule" macro. Initial experimentation with User 1 made it clear

that even a modest macro definition facility would require either very sophisticated inferences on the part of the system or a great deal of user interaction. Consequently, subsequent users were not afforded the same opportunity and CHAMP did not include the facility.

- Conjunction was allowed in the experiment and is encouraged by the stimuli. CHAMP, however, expects only one database action per utterance.
- Relative times (for example, "after 5 p.m.") were allowed in the experiment but are not provided for in the kernel grammar or the fixed Concept Hierarchy.
- For a few concepts, only a subset of the meanings allowed in the experiments were implemented. For example, a **projectname** in CHAMP always represents a group of people. In the experiments, a **projectname** was also a permissible *subject* for a meeting.
- During the simulation, changes to the database were made overnight rather than on-line. As a result, a user's utterance was never rejected because it was inconsistent with an entry introduced into the database by a previous sentence in the session. CHAMP performs changes on-line and therefore detects such inconsistencies.

Differences in functionality between the simulation and the implementation are compensated for by Rules 2, 3, and 5 in Figure 9-1, below.

2. Differences in inference capabilities:
   - During the simulation, *constructive inference* was used by the hidden-operator to build new forms and to eliminate redundant phrases (see Figure 3-2 for the rules followed by the hidden-operator). Constructive inference was considered to be possible if all the parts of a legitimate concept were present such that no more that two deviation points had accrued in the constituents. Constructive inference was intended as a way to overcome the arbitrary two-deviation limit when enough information was present in the utterance to satisfy a sentential-level caseframe (and, thus, a database action). In the simulation, forms derived by constructive inference were created by joining the tokens and generalized categories present in the confirmed interpretation. Thus, "The 3 o'clock meeting will be held in 5409" creates a **roomform** with the **roomwd** deleted as well as a **changeform** constructed from **m-d-ggform** + "will be held" + **m-locationform**. Without constructive inference, some portions of the users' grammars are simply not learnable (see Section 8.5). Constructive inference is discussed further in Chapter 11.
   - Inferences allowed during the Resolution phase of the simulation were more powerful than those implemented. For example, reasoning about time was more complex during the experiment than the sorts of inferences of which CHAMP is capable. In addition, some instances of inference used during the experiment are explicitly prevented by recovery-time constraints. It is not possible, for example, to infer the type of a deleted object from the other constituents in the sentence because steps with **object** *bindvars* are protected by a **no-del** constraint.

Inference differences are compensated for by Rules 4 and 7 in Figure 9-1.

3. Differences arising from lack of specificity in the model:

   - During the simulation, if there was a segmentation of tokens that made the sentence understandable, that segmentation was used. CHAMP, on the other hand, is constrained in its interpretation by the Single Segment Assumption which enforces the view that contiguous unknowns perform a single functional role within the utterance. Consequently, the Single Segment Assumption may prevent CHAMP from finding the correct segmentation and explanation of the sentence. For further discussion of the problem, see Section 5.1.2.

   - The simulation required only that some decomposition of the utterance into constituents be within two deviations of the grammar. In contrast, the Maximal Subsequence Heuristic forces CHAMP's parser to EXPAND only the largest subsequence that corresponds to a constituent. This may prevent sentences that are within two deviations of the current grammar from being recognized. For further discussion of the problem, see Section 5.2.

   - The *lexical extension problem* occurs when a known word or phrase is used in a new way. During the simulation it was assumed that CHAMP would be able to treat the tokens as unknowns under these circumstances. CHAMP's current design is unable to ignore the definitions for the word or phrase in the lexicon, however. The system relies on extra-grammatical markers to enforce the interpretation of the tokens as unknowns. Chapter 10 contains a more detailed discussion of the problem.

   - Since the adaptation model makes no commitment to a particular form of generalization, derived forms were integrated into the grammar in the most useful way possible during the experiments. Achieving the best possible generalization of a new form represents an ideal circumstance. CHAMP's method of generalization, while both uniform and useful, is hardly ideal. Thus, the grammatical components derived by CHAMP during adaptation do not always correspond to the same derived components added to the grammar during the experiments (this is especially true when a set of components was derived through constructive inference). The differences are felt primarily as undergeneralization in CHAMP, requiring more learning episodes to capture the user's conditions on usage.

   Differences that stem from the underspecified nature of the model are compensated for, in part, by Rules 1 and 7 in Figure 9-1.

Figure 9-1 specifies the methods used to compensate for the differences between the simulated and the implemented systems. The primary method of compensation relies on preprocessing the data in the specific ways outlined in Rules 1 through 4. Preprocessing enables CHAMP to accept some sentences that were accepted during the experiment but that CHAMP could not otherwise understand. Rule 1 compensates for the lexical extension problem by introducing extra-grammatical markers that help force the correct behavior. The only justification offered for the rule is that it compensates for an open problem in this research. Rules 2 through 4 change the actual tokens typed by the user.

The justification for these rules is simple: had the relevant forms been rejected by the hidden-operator at their first appearance (as they would have been by CHAMP), the user would not have relied on them in future utterances.

The second method of compensation described in Figure 9-1 is forced rejection or acceptance. During evaluation of the test set, CHAMP is forced to reject a sentence that it would otherwise have accepted if the hidden-operator mistakenly rejected it (Rule 5). On the other hand, if CHAMP found a resolution-time error in a sentence that was accepted in the experiment, the system was forced to adapt anyway (Rule 6). As a last resort, a difference in behavior is prevented from propagating to future interactions by introducing the necessary derived components by hand (Rule 7).

In parentheses after the statement of each rule, Figure 9-1 shows the number of sentences in the test set affected by each type of interference. Only two values are noteworthy. The fact that 145 sentences were preprocessed to remove redundancy (Rule 4) is actually quite misleading: 127 of those sentences were from User 4. She used the day of the week redundantly when specifying the date in almost every utterance. Had the redundancy not been learned by constructive inference the first time it occurred, it is unlikely that she would have included it in the remaining 126 sentences. Indeed, this is exactly what happened in User 10's protocol the first time she included a day of the week along with the full date. CHAMP's interaction with User 10 during segmentation of the sentence let her know that the system considered **saturday** an unknown token. As a result, she never included a day of the week as part of the date again.

It is more difficult to dismiss the sixteen percent of the test set sentences that had to be marked to compensate for the lexical extension problem. As we noted above, a solution for this problem within our adaptation model remains an open aspect of this research. We discuss the difficulties involved in parsing sentences that contain lexical extensions in Chapter 10. Here we note only that the problem is significant and cannot be ignored.

The purpose of preprocessing the original experimental data and interfering with the normal understanding process in CHAMP is to recreate the conditions of each protocol as closely as possible given the differences between the simulation and the implementation. Although some sentences required more than one kind of interference, most sentences required none. We turn now to an examination of CHAMP's performance on the preprocessed test set.

1. If a known word is being used in a new way and the hidden-operator interpreted it in the new way, then force it to be treated as an unknown by enclosing it in angle brackets. The modified sentence is then treated in accordance with the Single Segment Assumption.(84/539 (16%))
   Example: "the University of Chicago" has one unknown token, **university**, and three tokens being used in new ways. The segment therefore becomes: <the> university <of> <chicago>. When contiguous unknown tokens are combined, segmentation produces **the university of chicago** as a single unknown phrase.

2. Turn conjunctions into separate sentences, distributing shared information. (31/539 (6%))
   Example: "cancel flight #1 on june 16 and schedule flight #2" becomes "cancel flight #1 on june 16" and "schedule flight #2 on june 16" (this counts as two sentences affected by conjunction).

3. If a failure is caused by unimplemented semantics, change the data by replacing the phrase referring to the unimplemented concept with one that refers to an implemented equivalent. (39/539 (7%))
   Example: the unimplemented relative time, "after 7 p.m." is replaced with the concrete time, "from 7 p.m. to 11:59 p.m."

4. If a sentence contains a redundant phrase that was compensated for using constructive inference, then remove each occurrence of the redundant phrase. (145/539 (27%))
   Example: "Monday, June 12" becomes "June 12."

5. If a sentence was accepted during the experiment but CHAMP detects a resolution-time error for the meaning, ignore or correct the error and accept the sentence. (16/539 (3%))
   Example: if the user failed to complete a subtask that removed an event from the calendar then CHAMP will detect the conflict when the user tries to schedule a new event in the occupied time slot.

6. If CHAMP accepts a sentence not accepted during the simulation, then force rejection and prevent adaptation. (5/539 (1%))
   Example: this occurs when the hidden-operator mistakenly rejected a sentence as too deviant.

7. If a sentence was accepted during the experiment which CHAMP cannot parse (even after applying the previous rules), then consider whether the adaptations resulting from acceptance are required in the future. If what was learned from the sentence is never used again or CHAMP can learn it the next time it appears, then do nothing. If, on the other hand, what was learned will be required later and CHAMP will not be able to learn it at the next occurrence either, then add to the grammar by hand the minimal number of grammatical components that capture the necessary adaptations. (8/539 (1%) resulting in a total of 13 components being added by hand for the 4 users).
   Example: this occurs when acceptance was by constructive inference or when the hidden-operator erroneously accepted.

Figure 9-1: The rules used to preprocess the data from the hidden-operator experiments and (the number of test set sentences affected by each rule).

9.1.2. Comparison of Performance: Simulation versus Implementation

A fundamental purpose of the original experiments was to demonstrate the self-bounded linguistic behavior of frequent users in an adaptive environment. Consequently, our primary measure of performance was a calculation of the number of new constructions per interpretable utterance for each user and session (Figure 3-3 in Chapter 3). We found strong evidence for self-bounded behavior in the decrease over time of both the number of new constructions and the number of uninterpretable sentences. Figures 9-2 through 9-5 show the original graphs constructed for each of the four users in the Adapt condition side-by-side with CHAMP's performance for the same measure.[43] The graphs for User 2 show the most similarity, the graphs for User 1, the least. We explain the causes for the apparent differences between the sets the graphs in the remainder of this section. Despite the differences, the general pattern remains: an overall decrease in the number of new constructions and rejected sentences over time, with a local increase after the introduction of the supplementary instructions to work quickly. In short, the implementation's behavior conforms closely enough to the model to capture the regularities present in the test set utterances.

It should be noted that the y-axis in the lefthand graph of each figure is labelled differently from its counterpart in Figure 3-3 where it read, "new constructions/interpretable sentences." For the hidden-operator graphs, the meaning of the new label ("new constructions/learning opportunity") is the same as the old: the number of interpretable sentences is exactly the number of opportunities for learning and exactly the number of sentences accepted. Under the conditions of this evaluation, however, CHAMP's opportunities for learning may be different from the number of sentences the system can interpret. In accordance with the rules in Figure 9-1, we do not consider as opportunities for learning those sentences from the test set that CHAMP understood but was forced to reject (Rule 6), but we do consider as opportunities for learning those sentences CHAMP was forced to accept (Rule 5) as well as those after which we added to the grammar by hand (Rule 7).[44] Note that it is possible to have more than one new construction per opportunity for learning if multiple deviations in the sentence resulted in the addition of multiple derived components to the grammar (see Figure 9-5).

---

[43]The values shown for CHAMP take competition into account. In other words, if six components were added to the grammar as three sets of competitors, only two components (one competitor) were counted as new constructions. Taking competition into account gives a more realistic picture of the asymptotic behavior of the system: as competitions are resolved only one set of components will remain in the grammar. The numbers of new constructions without considering competition are provided in Appendix C.

[44]None of the graphs includes learning at Deviation-level 0 (new instances of known classes, abbreviations, and stored spelling corrections) because we do not expect the same decrease in these phenomena from self-bounded behavior as we expect for deviations. The number of instances of learning at Deviation-level 0 for each user can be found in Appendix C.

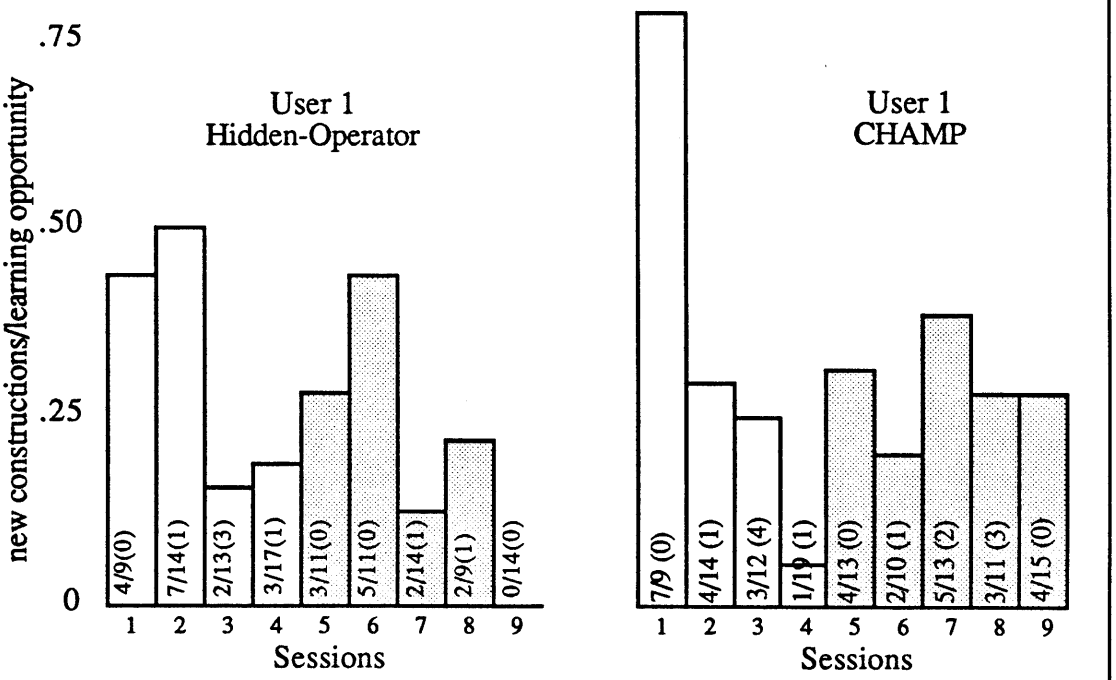

Figure 9-2: Capturing User 1's regularity under the model and in CHAMP (shaded sessions followed the supplementary instructions to work quickly, the number of unparsable utterances for each session is given in parentheses).

What it means to be "unparsable" must be defined carefully for CHAMP as well. The figures show the number of unparsable sentences for each session in parentheses after the fraction for new constructions. The number of unparsable sentences in the experiments is exactly the number of utterances rejected. The number of unparsable sentences for CHAMP, on the other hand, includes not only those sentences actually rejected by the system (some of which lead to adaptations anyway, via Rule 7) but also those rejected by force (Rule 6). Note that under this method of evaluating the implementation it is possible for the same sentence to be both unparsable (showing up in parentheses) and a contributor to the number of opportunities for learning. In the experiment, this was impossible.

Although we attempted to recreate the conditions of the protocol as closely as possible, Figures 9-2 through 9-5 show that CHAMP's performance was not exactly the same as the model's. The differences between the two graphs for each user stem from three sources: the number of rejected sentences, undergeneralization during adaptation, and the redistribution of learning episodes by Rule 7. We analyze the reasons for the increase in rejections in the next section; here we note only that CHAMP does, in general, reject more sentences than does the model. When the number of opportunities for learning decreases, the number of constructions per opportunity, reflected by the height of the bars, increases.

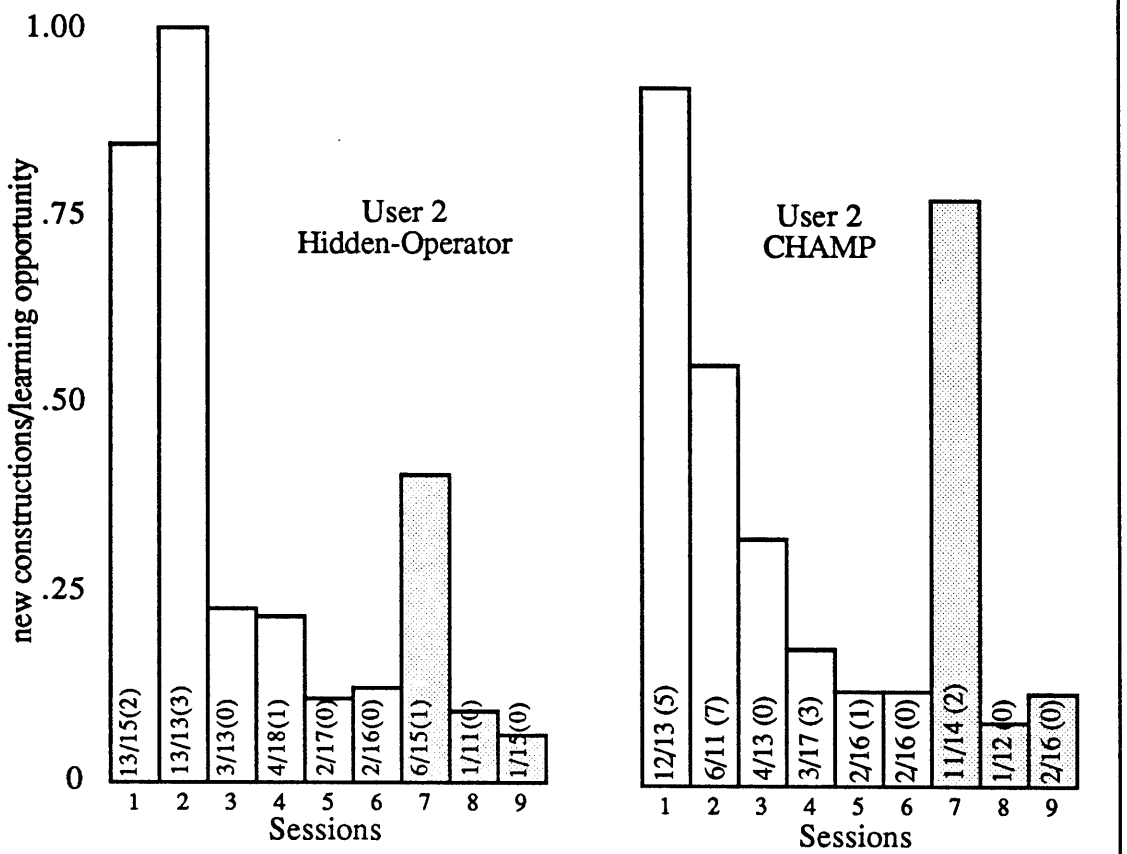

Figure 9-3: Capturing User 2's regularity under the model and in CHAMP (shaded sessions followed the supplementary instructions to work quickly, the number of unparsable utterances for each session is given in parentheses).

CHAMP undergeneralizes in comparison to the hidden-operator's behavior primarily because CHAMP lacks constructive inference. In essence, more episodes of learning by straightforward adaptation are required to cover all the cases covered by a single episode of learning via constructive inference and hidden-operator generalization. The lack of constructive inference accounts for many of the differences in the sessions after the supplementary instructions were given; CHAMP had undergeneralized some transpositions and deletions that were crucial as utterances became shorter.

The redistribution of learning episodes occurs because of the uneven compensatory role of Rule 7. Recall that if a component could not be learned by CHAMP but had been relied upon by the user in future utterances, Rule 7 adds the component to the grammar immediately after the sentence that contained the deviation. If the component could be learned at the next occurrence, however, no action is taken. As a result, some components learned during session *i* in the hidden-operator graph may not have been learned until sessions *j* and *k* in CHAMP's graph, depending upon when the next sentence requiring them occurred in the protocol.

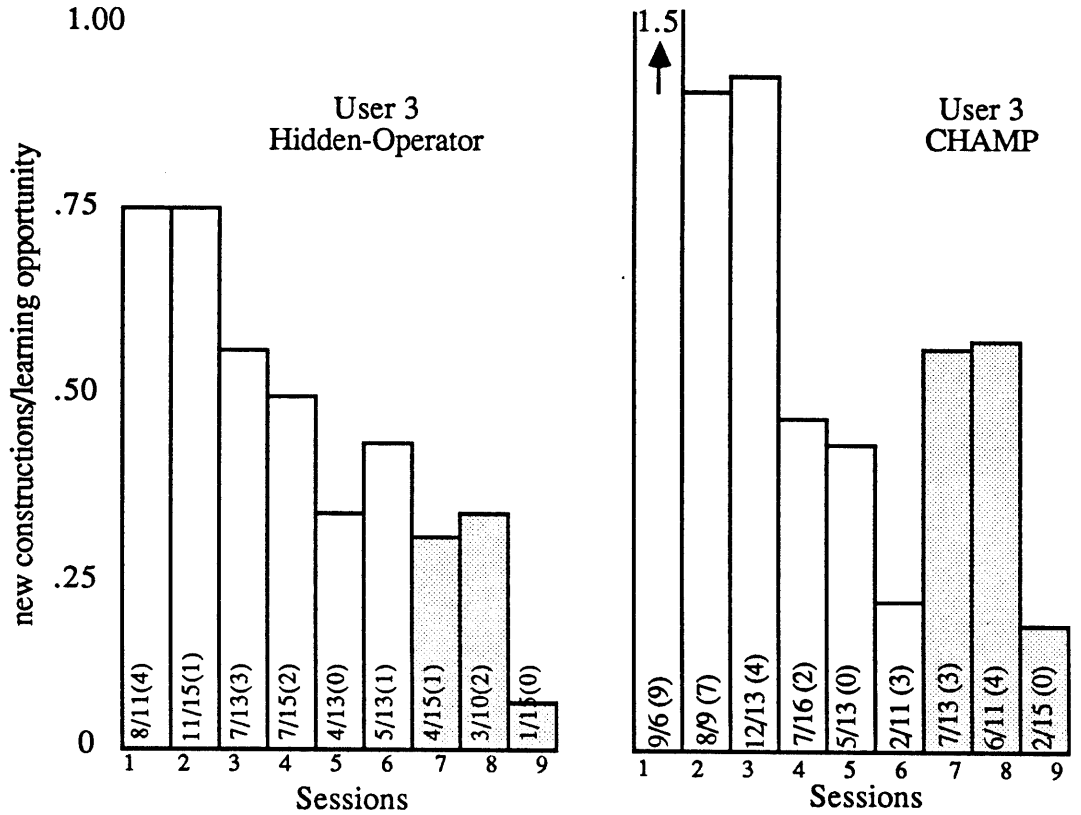

Figure 9-4: Capturing User 3's regularity under the model and in CHAMP (shaded sessions followed the supplementary instructions to work quickly, the number of unparsable utterances for each session is given in parentheses).

Despite these differences, the graphs for performance under CHAMP show the same general pattern as those under the simulation, reflecting the inherently self-bounded nature of each user's utterances and the ability of the implementation to capture her regularity.

9.1.3. Analysis of Differences in Acceptance/Rejection

Having noted that CHAMP accepts fewer sentences from the test set than did the simulation, it is useful to ask what causes the additional rejections. When we factor out hidden-operator errors (five erroneous rejections and seven erroneous acceptances), we find that 38/539 of the test set sentences were truly too deviant with respect to the grammar current at the time of use. Thus, a perfect implementation of the model would have rejected seven percent of the sentences. Factoring out forced rejections, CHAMP fails to accept 73/539 sentences, or about fourteen percent.

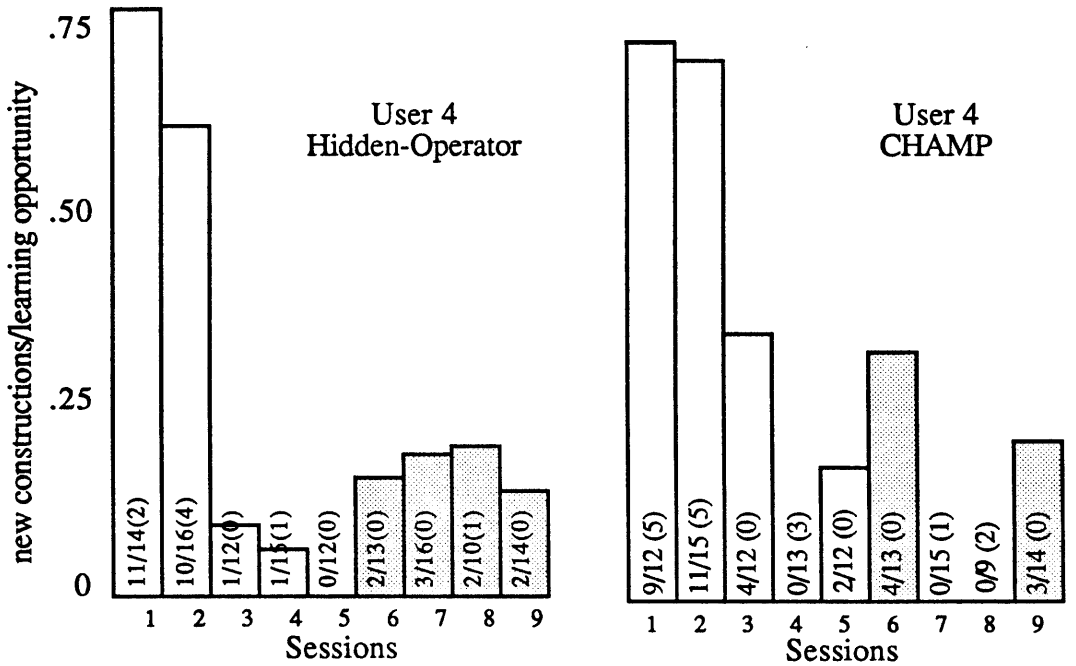

Figure 9-5: Capturing User 4's regularity under the model and in CHAMP (shaded sessions followed the supplementary instructions to work quickly, the number of unparsable utterances for each session is given in parentheses).

Reason for difference	% of test set
Constructive inference	16/539 (3%)
Single Segment Assumption	18/539 (>3%)
Recovery-time constraint	1/539 (<1%)
Total:	35/539 (7%)

Figure 9-6: A breakdown of the reasons why CHAMP rejects 35 sentences from the test set that were acceptable according to the adaptation model.

Figure 9-6 shows that most of the thirty-five rejections not predicted by the model come from two sources: the lack of constructive inference and the intrusion of the Single Segment Assumption. Note that for the four users in the Adapt condition of the original experiments only one sentence was rejected because of recovery-time constraints and no sentences were rejected because of the Maximal Subsequence Heuristic.

Although a three percent increase in rejections from the lack of constructive inference does not seem like a lot, the statistic does not reflect the true value because of Rule 7. Using Rule 7 we added by hand those grammatical components that would be required but still unlearnable later in the user's protocol. Thus the sixteen sentences rejected

because of constructive inference represent only a portion of the sentences truly affected—the portion that contained adaptations that could be learned at the next occurrence. Without constructive inference or Rule 7, CHAMP would have rejected at least eight additional utterances.

Constructive inference figured prominently in the simulation because it is a good mechanism for overcoming bad choices in the kernel design. One bad design choice in the kernel, for example, affected every user's ability to accomplish the class of subtasks that added a value to an empty slot in a database record. From a database update point of view these subtasks correspond to a **change** action in which the source value for the slot is not specified (for example, "Change the speaker of the AI seminar [from nil] to John"). Thus, it was expected that the kernel **changeforms** adequately indexed the action. Unfortunately, none of the users thought about those subtasks in that way. Instead, they seemed to think about the subtasks as "amendment" actions, distinct from both **schedule** actions and **change** actions (for example, "Add John as the speaker for the AI seminar."). In the experiment, this conceptual difference posed no serious problem because each user eventually employed a deviant **changeform** acceptable via constructive inference. In CHAMP, this would not have been possible, and users would have had to battle a conceptual mismatch as well as a two-deviation limit on error recovery. We examine the issues involved in extending CHAMP to include constructive inference in Chapter 11.

The Single Segment Assumption was also responsible for rejecting three percent of the sentences that were recognizable by the model. The effect is modest for this user group and task. While the experiments with new users do not result in a higher percentage, it is unclear whether the problem might be more pervasive given a different task or domain. We briefly examine ways in which to extend the system to compensate for single segment violations in Chapter 11.

9.1.4. Analysis of the Utility of Adaptation and Generalization

Although we were able to draw many useful conclusions from the experiments, the underspecification of the model made it impossible to give quantitative answers to a number of questions we would have liked to ask. It is difficult, for example, to derive the utility of adaptation in any sort of precise way from the results of the simulation when generalization by the hidden-operator provided some of the power of the learning mechanism. An implementation of the model, on the other hand, allows access to quantitative measures. The purpose of this section, then, is to provide a sense of the overall utility of our method of adaptation and generalization by answering the following questions for CHAMP's performance on the test set:

- What is the effectiveness of learning? How much actual improvement in understanding the user's language comes from adaptation?

- What is the cost of learning? What affect does adaptation and generalization have on the level of ambiguity in the grammar?
- How effective is competition as a method of controlling grammar growth?
- Given a uniform (non-human) method of adaptation and generalization, how truly idiosyncratic are the derived grammars?

To answer the first of these questions, we contrast for each user the number of her sentences accepted by the kernel and the number accepted by her final grammar—any increase is due to learning. Figure 9-7 shows that the increase is significant for each user. On the average, the kernel accepts only 16% of a user's utterances while her own derived grammar accepts 88%.

	User 1	User 2	User 3	User 4
G_{kernel}	18/127 (14%)	34/144 (24%)	9/138 (7%)	25/130 (19%)
G_{final}	115/127 (91%)	127/144 (88%)	112/138 (81%)	118/130 (91%)

Figure 9-7: The effectiveness of learning as measured by the differences in the number of utterances understood by the kernel and by the user's final grammar.

To carry the analysis further, we measure the utility of each learning episode by computing the average number of sentences brought into the grammar each time a deviant utterance is explained. The value is computed by subtracting the number of sentences accepted by the kernel from the number accepted by a user's grammar and dividing by the number of learning episodes. For Users 1, 2, 3, and 4, respectively, the values are: 3.3, 2.7, 2.3, and 3.7 sentences per episode. These values represent a kind of "bootstrapping constant" that reflects the way in which CHAMP's particular implementation of adaptation and generalization captures within-user consistency. An alternative implementation would probably produce very different values. Consider, for example, an implementation using conservative substitution adaptations (as described in Chapter 8). The tendency of the conservative approach to undergeneralize is likely to appear as a decrease in the utility of each learning episode: the user's final grammar might result in the same number of acceptances but at the cost of requiring more instances of adaptation. Thus, the values themselves are less important than the fact that our ability to compute them provides a metric for comparing design choices.

The second question posed at the beginning of the section concerns the cost of adaptation. As we bring more of the user's language into the grammar we increase the likelihood that we will understand her future utterances. But is the increase in

understanding, as measured by acceptances, negated by a larger increase in the cost of understanding, as measured by search? We know that the user's language is self-bounded, but it may still be quite ambiguous. The question, then, is not whether we can prevent the rise in search stemming from inherent ambiguity in the user's idiolect, but whether the system as a whole suffers disproportionately as ambiguity increases.

We measure the rise in ambiguity in an adapted grammar in two ways. First, holding the test sentences constant, we compare the average number of parse states considered by successive grammars during search at Deviation-level 0. As the level of ambiguity in the grammar increases through the adaptations of each session, so will the average amount of search required to accept at Deviation-level 0. Second, we examine the average number of explanations (roots) produced by the Parse/Recovery Cycle for each sentence accepted at Deviation-level 0. This value reflects a rise in ambiguity that is somewhat independent of the increase in search because of the way annotated parse trees share substructure. A rise in search need not give rise to additional explanations; additional root nodes may indicate only a modest increase in search. From the user's point of view increased search corresponds to decreased response time while a rise in the number of explanations corresponds to an increase in the number of interactions required during resolution.

Figures 9-8 through 9-11 display the measurements for the successive grammars for each user. The figures show, session by session, the growth in the average number of states and roots over all the user's sentences that are non-deviant with respect to the state of the grammar at the end of the session. Thus, the first row of Figure 9-8 shows that the kernel grammar, G(0), accepts eighteen sentences from User 1, examining nineteen parse states and producing one explanation per sentence, on the average. The second row of the same figure reflects changes in performance due to learning in User 1's first session. The derived components are responsible for bringing sixty-one sentences from that user's total protocol into Deviation-level 0 with no significant rise in search and only a small increase in the average number of explanations produced.

For three of the four users, the increase in the number of parse states examined is proportionally far less than the increase in the number of sentences understood. By the end of the ninth session, the search for User 1 has expanded by a factor of two but the number of her sentences that are now non-deviant has expanded by a factor of six. User 3's trade-off is even more favorable: twelve times as many sentences are accepted by the final grammar as by the kernel, at a cost of only two and a half times the search. User 4 gains almost five times as many sentences at slightly less than twice the search. User 2 has the most balanced case: a factor of 3.5 increase in accepted utterances and a factor of three increase in search.

If increase in search corresponds to increase in response time, what do these values tell us? In short, response times will get slower over all but will not grow exponentially as a

Grammar after session(i)	Number of non-deviant sentences	Average number of states to accept	Average number of roots produced
G(0)	18	19.3	1.0
$G_1(1)$	79	37.0	1.1
$G_1(2)$	86	37.4	1.3
$G_1(3)$	88	38.3	1.3
$G_1(4)$	90	37.7	1.3
$G_1(5)$	96	36.4	1.2
$G_1(6)$	98	36.2	1.3
$G_1(7)$	104	37.2	1.4
$G_1(8)$	111	36.8	1.5
$G_1(9)$	115	37.6	1.5

Figure 9-8: The change in the cost of parsing User 1's sentences at Deviation-level 0 as a function of grammar growth.

Grammar after session(i)	Number of non-deviant sentences	Average number of states to accept	Average number of roots produced
G(0)	34	21.9	1.0
$G_2(1)$	78	25.8	1.1
$G_2(2)$	91	26.5	1.1
$G_2(3)$	94	28.7	1.2
$G_2(4)$	103	57.6	1.4
$G_2(5)$	107	61.5	1.6
$G_2(6)$	114	59.4	1.6
$G_2(7)$	121	59.4	1.6
$G_2(8)$	123	59.8	1.6
$G_2(9)$	123	59.8	1.6

Figure 9-9: The change in the cost of parsing User 2's sentences at Deviation-level 0 as a function of grammar growth.

function of the increase in the language accepted. Since the grammar itself is bounded by the user's natural behavior, the increase in response time is bounded as well. Even with kinds of increases seen for these users, response times were usually under ten seconds.

The near-monotonic increase in the size of the search goes hand-in-hand with a near-monotonic increase in the average number of explanations produced for each accepted utterance. Where exactly is the ambiguity coming from? The users do introduce some lexical ambiguity into their idiosyncratic grammars, but lexical ambiguity contributes primarily to small, local increases in the size of the search space. As paths representing the correct constituents are JOINed, the Maximal Subsequence Heuristic eliminates useless alternative paths which, in turn, keeps the effects of lexical ambiguity from propagating very far up the Agenda.

Grammar after session(i)	Number of non-deviant sentences	Average number of states to accept	Average number of roots produced
G(0)	9	18.0	1.0
$G_3(1)$	21	30.3	1.1
$G_3(2)$	28	30.8	1.5
$G_3(3)$	71	34.6	1.4
$G_3(4)$	83	46.7	2.1
$G_3(5)$	92	44.6	2.1
$G_3(6)$	93	44.2	2.1
$G_3(7)$	103	43.2	1.9
$G_3(8)$	109	43.6	1.9
$G_3(9)$	112	43.3	2.0

Figure 9-10: The change in the cost of parsing User 3's sentences at Deviation-level 0 as a function of grammar growth.

Grammar after session(i)	Number of non-deviant sentences	Average number of states to accept	Average number of roots produced
G(0)	25	21.0	1.0
$G_4(1)$	62	28.0	1.1
$G_4(2)$	78	30.9	1.3
$G_4(3)$	95	37.3	1.3
$G_4(4)$	106	36.9	1.3
$G_4(5)$	112	38.8	1.3
$G_4(6)$	116	40.2	1.3
$G_4(7)$	116	40.2	1.3
$G_4(8)$	116	40.2	1.3
$G_4(9)$	118	40.7	1.5

Figure 9-11: The change in the cost of parsing User 4's sentences at Deviation-level 0 as a function of grammar growth.

Most of the increase in ambiguity comes from adaptation to deletion deviations. As the user's language becomes increasingly terse, the system builds forms in which the content words that correspond to critical differences between strategies are deleted. As a result, there is an increase in the number of constituents that are satisfied by each segment; the increase propagates to each level of the Agenda and, eventually, to the root nodes themselves.

Consider as a simple example what happens if the user drops both a marker and a head noun (a common occurrence). From that point on, every sentence indexing the derived components produces multiple roots: "Schedule 7 p.m. on June 4 with John" produces two effectively equivalent roots if neither a **meetingwd** nor an **hourmkr** is required by

the grammar. One tree contains "7 p.m." as a normal unmarked prenominal constituent in the **meetingform** strategy with the **meetingwd** deleted. The second tree explains the same time segment as a postnominal constituent using both of the components derived by deletions. If the marker for the date is also optional in the adapted grammar (another common occurrence), and we change the sentence slightly to, "Schedule 7 p.m. June 4 with John," then four roots result from the obvious combinatorics (unmarked prenominal versus marked postnominal for each modifier). If the user has derived forms for more than one type of **groupgathering**, each of which has the head noun deleted (a slightly less common occurrence), the same sentence will produce four roots for each **groupgathering** that can be recognized without its head noun. Similar problems arise when the user relies on derived forms without verbs.

The third question we posed at the beginning of the section concerns the effectiveness of competition as a method of controlling grammar growth. Competition allows the system to postpone choosing among alternative sets of adaptations that explain the same deviations. Thus, the issue of grammar growth is related to but not identical to increased ambiguity. Competitions arise from ambiguities already present in the grammar; the purpose of competition is to keep those ambiguities from proliferating exponentially via further adaptations.

What do we mean by the "effectiveness of competition?" In short, we want to know whether competition allows the system to eliminate all but one subset of explanations from a set of effectively equivalent adaptations. The reoccurrence of a competitor is necessary but not sufficient to resolve a competition; it is possible for all competitors to succeed when any one succeeds. If competition is a useful mechanism then it provides the system with a way of choosing among alternatives, favoring components that explain an idiosyncratic constituent in more than one context. We consider competition harmful, however, if most competitors are preserved even with repeated use; under these circumstances, it would have been less costly to have the system choose arbitrarily among the initial set of alternative adaptations.

Figure 9-12 displays the relevant competition values for users in the Adapt condition. With the exception of User 4, most of the competitions that had a chance to be resolved were resolved the next time a competitor was required to understand a sentence. For User 4, subsequent uses of competitors generally occurred in utterances that were structurally identical to those that gave rise to the competition initially. As a result, all explanations succeeded repeatedly and the competition remained unresolved.

Considering Figures 9-8 through 9-12 together, we conclude that although competition is fairly successful at stopping the exponential proliferation of existing ambiguities, some additional mechanism for removing ambiguity would be useful as well (see Chapter 11 for further discussion).

	Number resolved competitions	Number of competitions left unresolved	Number left unresolved with chance to resolve
User 1	0	1	0
User 2	1	5	1
User 3	7	10	2
User 4	1	4	4

Figure 9-12: Controlling grammar growth by preserving only the most useful explanations for deviations through competition.

The final question we posed at the beginning of the section concerns across-user variability. Recall from Chapter 3 that a strong argument can be made in favor of adaptation at the interface if users' grammars are truly idiosyncratic. With high variability in preferred expression across users, a single grammar capable of understanding all users actually penalizes each individual user during search for forms she does not use. Although the simulation resulted in little overlap among the users' final grammars, one could argue that the perceived idiosyncracy was an artifact of the hidden-operator's freedom when generalizing the derived form. CHAMP's uniform adaptation and generalization mechanism provides the opportunity to examine across-user variability in a more meaningful way.

We attain a quantitative measure for idiosyncracy by examining the acceptance rate for each user's utterances under her own and each of the other users' final grammars (Figure 9-13). If each final grammar accepts approximately the same language, then there should be no significant difference in the acceptance rates. In contrast, Figure 9-13 clearly demonstrates a wide range of variability: User 3's final grammar is able to parse 59% of User 1's sentences, but User 4's final grammar can parse only 14% of the utterances from User 3. Observe that in no instance did the grammar for another user come close to the performance of the user's own final grammar. Taken in conjunction with the high acceptance rates by each user's idiosyncratic grammar, we conclude that the method of adaptation and generalization realized in CHAMP is adequate to accommodate the very real variability in preferred expression that exists across users.

9.1.5. Adaptive versus Monolithic Performance

To what degree did the size of CHAMP's kernel grammar affect the analyses in the previous section? Did we stack the deck in favor of adaptation by choosing a small kernel? It could be argued that with a larger grammar, learning might not be necessary at all—whatever small mismatches exist between the user's natural language and the system's natural language might be easily overcome by user adaptation. Even across-user variability is unimportant if everyone can be understood. In short, wouldn't a monolithic, non-adaptive grammar have performed just as well?

Final Grammar for	Applied to All Sentences of User 1	User 2	User 3	User 4
User 1	**115/127 (91%)**	72/144 (50%)	40/138 (29%)	60/130 (46%)
User 2	59/127 (46%)	**127/144 (88%)**	15/138 (11%)	74/130 (57%)
User 3	75/127 (59%)	71/144 (49%)	**112/138 (81%)**	66/130 (51%)
User 4	48/127 (38%)	66/145 (46%)	19/138 (14%)	**118/130 (91%)**

Figure 9-13: Measuring across-user variability by computing the acceptance rate for each user's utterances under her own and each of the other users' final grammars.

The answer is no. To see why, let us build the best monolithic grammar possible for the four users we have been studying, best in the sense that it is guaranteed to understand every sentence understood by any of the users' individual grammars. We construct the grammar by adding to the kernel the union of the users' final grammars minus any redundant components. Figure 9-14 demonstrates what happens when we use the monolithic grammar to parse the test set.

User	Average States$_{user}$	Average States$_{mono}$	Average Roots$_{user}$	Average Roots$_{mono}$	Additional sentences
1	37.6	109.6	1.5	7.7	4
2	59.8	155.5	1.6	8.5	4
3	43.3	156.5	2.0	9.3	1
4	40.7	148.3	1.5	7.1	1

Figure 9-14: A comparison of the relative costs of parsing with a monolithic grammar versus parsing with the idiosyncratic grammars provided by adaptation. (the last column shows the number of additional sentences for each user that could be parsed by the monolithic grammar but not the user's final grammar).

In every instance the average number of search states examined by the monolithic

grammar is significantly greater than the number required by the user's idiosyncratic grammar. Of course, we do gain the ability to understand ten additional sentences in the test set (this occurs when adaptations learned for User *i* are able to parse a sentence that was rejected for User *j*). The trade-off hardly seems favorable, however. User 1 would find a three percent increase in the number of her sentences understood by the interface (4/127) at a cost of almost three times the search. User 2 also receives about a three percent increase (4/144) at a cost of about two and a half times the search. User 3 and User 4 suffer the most: each receives an increase of less than one percent in accepted sentences but must wait through 3.6 times the amount of search.

The trade-off seems even less favorable when we consider the difference in the number of roots produced under each type of system. The monolithic static grammar produces about five times as many roots on the average as a user's adapted grammar—five times as many incorrect or redundant explanations. Even for a system able to make the kinds of resolution-time inferences that CHAMP can, a significant portion of multi-root sentences cannot be resolved by inference and constraints from the databases alone. Thus, in addition to waiting through longer searches, the users are likely to be subjected to more interactions as well.

In Chapter 1 we argued from a theoretical point of view that a monolithic approach to interface design must engender some degree of mismatch between the language accepted by the system and the language employed by any particular user. The mismatch comes in two forms: language the user prefers but that cannot be understood by the interface and language the interface understands that is never employed by the user. In this section we have controlled for the first kind of mismatch by guaranteeing that the monolithic grammar could parse at least as many sentences in the test set as could CHAMP. Figure 9-14 shows the price paid by each user for the second kind of mismatch during every search over those portions of the system's language she will never use. We know that we cannot control for the second kind of mismatch because of the real variability of expression across the users in our sample. Although it may be possible to achieve the same acceptance rate with less ambiguity in a much more carefully crafted monolithic grammar, the across-user variability is not going to disappear.

In contrast to the monolithic system's performance, adaptation minimizes both kinds of mismatch without requiring inordinate skill as a grammar designer. By learning the user's preferred forms of expression and engendering only those mismatches required by its small kernel, an adaptive interface keeps the cost of understanding each user in proportion to the inherent ambiguity in her idiosyncratic language.

9.2. CHAMP's Performance in Real User Interactions

The performance analysis in Section 9.1 seems to argue strongly in favor of an adaptive interface design for frequent users. Still, one might argue that the sorts of interference we needed to employ to compare the implementation with the simulation, the use of the development set during system design, previous exposure to the entire test set during the experiments, and the small user sample all contributed to an exaggerated evaluation of the system's effectiveness.

To guard against this possibility, CHAMP was tested in on-line experiments with two new users (User 9 and User 10) performing the same task as the original users. The two new users were drawn from the same population as the first eight (female professional secretaries between twenty and sixty-five years of age who keep a calendar as part of their jobs but who have no prior experience with natural language interfaces). The experimental condition was the same as that described in Section 3.2 for the Adapt users, including initial instructions, supplementary instructions, pictorial stimuli, and a limit of three attempts per subtask. The kernel grammar used in the on-line experiments can be found in Appendix A.[45]

The role of the hidden-operator in the on-line experiments was reduced to adding extra-grammatical markers to compensate for the lexical extension problem before passing the user's sentence to CHAMP. Of the 385 sentences for the new users, thirty-one (8%) required hidden-operator intervention.

In the next three sections, we evaluate CHAMP's performance in the on-line experiments in terms of the same measures examined for the original users. A variety of other measures (including a response-time synopsis) can be found in Appendix C. The analysis shows the same patterns for spontaneous inputs as it did for the modified test set from the initial experiments. This validates our previous results and suggests that the modifications themselves played no significant part in the system's performance. Note also that the increase in sample size, albeit modest, did not introduce qualitatively different results.

---

[45]The kernel grammar for the experiments differed from the kernel grammar used to evaluate the original data in two ways. First, a set of kernel **changeforms** were added to compensate for the conceptual problem of amendments discussed in Section 9.1.3. Second, the kernel form recognizing marked calendar segments had its marker changed from required to optional. This made it more likely that user preferred forms for the **show** action would succeed (in the original experiments, user preferred **showforms** were usually learned by constructive inference). Many subtasks required the user to look at the calendar or airline schedule before actually performing the indicated action. Thus, making user **showforms** slightly closer to the kernel made it less likely that users would be prevented from carrying out multiple subtasks because of one unacceptable form.

9.2.1. Synopsis of User and System Performance

The self-bounded behavior of the two new users is evident in Figure 9-15. Each user shows the same overall decrease in the number of new constructions per interpretable sentence as the original users, as well as the same local increase and return to stabilization after the supplementary instructions are given.

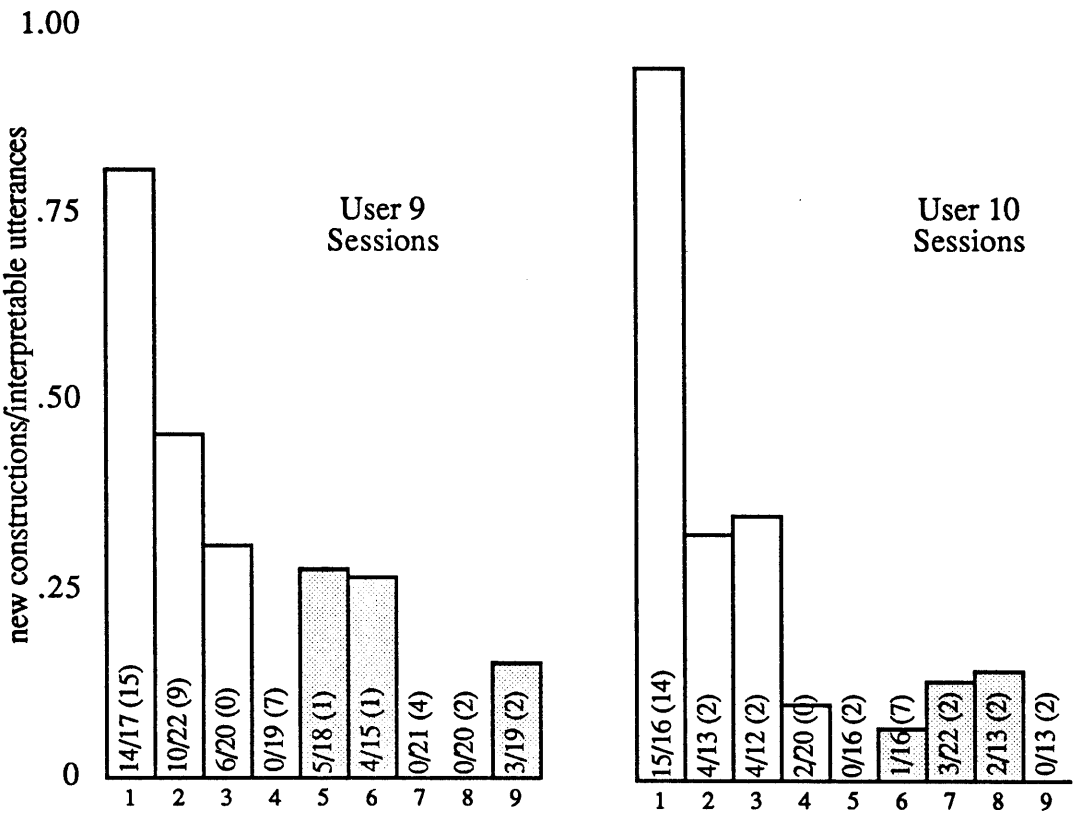

Figure 9-15: Self-limiting behavior in on-line experiments as measured by the number of new constructions/the number of interpretable sentences (shaded sessions follow the supplementary instructions to work quickly).

User 9's behavior was particularly interesting. Despite the fact that the instructions clearly stated that the system could not understand its own output, she insisted on using the system's messages as a template for her own commands (User 7 had shown a less extreme form of the same behavior). Figure 9-16 demonstrates one such episode. By the end of her third session, User 9 had essentially bootstrapped the system to where it could parse a terser form of its own already terse output.

User 10's natural grammar was quite different from the kernel, resulting in both a high rejection rate and high number of new constructions on the first day. Of the ten users

```
next> show schedule for June 10

Tracking...9...13...16...18...20...21...21...21...done.

JUNE 10    10:00am - 12:00pm MEETING in/at OFFICE with AISYS

Ok, done.

next> cancel 10:00am Meeting in/at Office with AISys June 10

Trying to understand IN/AT. Does it refer to a/an:
(0) DATE, (1) TIME-INTERVAL, (2) SINGLE-TIME, (3) PARTICIPANT,
(4) LOCATION, (5) SUBJECT, (6) GENERAL-TOPIC,
(7) none of these? [give number]: 7

Tracking...16...28...37...41...49...50...50...50...done.
Tracking...52...54...61...67...79...82...82...82...done.

Do you want:
   to delete: JUNE 10    10:00am - 12:00pm MEETING in/at OFFICE with AISYS?
[y or n] y

Creating LOCMKR from IN/AT via SUBST
Ok, done.
```

Figure 9-16: An example of User 9's tendency to formulate her utterances based on CHAMP's output ("Tracking" shows the node count at each Agenda-level and deviation-level. It was a device to keep the user's attention focused during search.)

studied, however, she was by far the most adaptive and rapidly settled into a comfortable mix of kernel forms and derived forms. In fact, at the end of her third session, User 10 commented that she found it more natural to use a form she believed the system would understand than to use an alternative form that she preferred but that she believed the system could not understand. Figure 9-15 demonstrates that CHAMP was doing a significant amount of learning despite User 10's belief that only she was adapting.

As was the case with the original users, both of the new users finished most of the subtasks in the experiment. On three occasions a new user was unable to effect the correct change to the calendar due to the system's incorrect inferences (twice for User 9, once for User 10). User 10 was also stopped once by the Maximal Subsequence Heuristic. She was the only user to have sentences rejected for this reason (three sentences corresponding to one subtask). The exact cause of the rejections is explained by Figure 5-13. Except for a slight increase after the supplementary instructions were given, each new user also showed a steady decrease in the number of utterances per task over time as her natural language and her adapted grammar converged.

9.2.2. Analysis of Rejections

Extrapolating from the analysis of rejected sentences for users in the original Adapt condition, we expect CHAMP to reject about twice as many sentences as the model considers unparsable. Figure 9-17 confirms the prediction. Eight percent of the new test set was either too deviant with respect to the grammar current at the time of use or corresponded to meanings that conflicted with information in the database at that time. CHAMP rejects an additional eight percent of the test set for implementation-dependent reasons.

Reason for rejection	% of test set
Deviance or resolution-time conflict	31/385 (8%)
Implementation	29/385 (8%)
Operator	4/385(1%)
User	10/385 (3%)
Total:	74/385 (20%)

Figure 9-17: Apportioning the blame for rejected sentences in the on-line experiments.

In addition, four sentences were rejected because the hidden-operator failed to notice occasions of lexical extension. The ten rejections attributed to the user represent cases where CHAMP produced the correct interpretation of the sentence typed, but the user rejected the interpretation during resolution. Usually this meant that the user had misunderstood the stimuli, but occasionally she had simply changed her mind about what to add to the calendar.

Figure 9-18 breaks down the implementation-dependent rejections by cause. The largest subgroup is formed by sentences in unreachable portions of the user's language space. These deviations cannot be learned because each one tries to introduce a subconstituent into a formclass that does not expect it. We saw an example of an unreachable deviation for User 10 in Section 8.5: "Change June 11 NY to Pittsburgh from flight 82 to flight 265" (U10S120). Since no kernel form expects a full **flight** object in the source case of a source-target pair, CHAMP cannot understand the segment "from flight 82." Further, CHAMP cannot learn to understand the segment because the system adapts only in the presence or absence of expected categories. Were the system extended to include constructive inference, however, each of the fourteen unlearnable deviations would be within reach.

Overall, the pattern of rejections for the new users conforms to the pattern we saw in the original test set. In a perfect implementation of the model, the average acceptance rate for user utterances is about 92%. Given the constraints on understanding imposed by the

Reason for rejection	% of test set
Unlearnable deviations	14/385 (4%)
Maximal Subsequence Heuristic	3/385 (<1%)
Single Segment Assumption	3/385 (<1%)
Recovery-time constraints	6/385 (2%)
Incorrect inferences	3/385 (<1%)
Total:	29/385 (8%)

Figure 9-18: A breakdown of the causes of implementation-dependent rejections.

design choices in CHAMP, acceptance falls to about 84%. A significant portion of the implementation-dependent rejections could be reversed by the addition of constructive inference to the system.

9.2.3. Analysis of the Utility of Adaptation and Generalization

Measurements other than rate of acceptance show the same patterns for the new users versus the original users as well. If we compare Figure 9-7 with Figure 9-19, for example, we find that the learning effects for Users 9 and 10 are comparable to the learning effects for Users 1, 2, 3 and 4: an average increase via adaptation of 78% for the new users and an average increase of 72% for the original users. Notice that CHAMP did quite a bit of learning for User 10 despite her belief that she was doing all the adapting.

	User 9	User 10
G_{kernel}	20/212 (9%)	19/173 (11%)
G_{final}	192/212 (91%)	148/173 (86%)

Figure 9-19: The effectiveness of learning as measured by the increase in the number of utterances understood for each new user via adaptations.

In Section 9.1.4 we defined a measure called the "bootstrapping constant" to reflect the utility of each learning episode. To compute the constant we divide the number of utterances brought into the grammar by adaptation by the number of learning episodes. The values for Users 1, 2, 3, and 4 were 3.3, 2.7, 2.3, and 3.7, respectively. The constants were somewhat higher for both Users 9 and 10: 5.7 and 5.6, respectively.

Another measure we examined for the original users was the rise in ambiguity in the

grammar over time due to adaptation. Figures 9-20 and 9-21, like Figures 9-8 through 9-11, show that as more of the user's language is brought into the grammar, more work is required to understand each utterance at Deviation-level 0. The trade-off is extremely favorable for User 10 (Figure 9-21): she gains almost eight times as many accepted sentences with virtually no increase in search. The trade-off is less favorable for User 9 (almost a factor of ten increase in the number of her sentences accepted at a cost of about six times the search) but is still within the general trade-off ratios seen among Users 1,2, 3, and 4.

Grammar after session(i)	Number of non-deviant sentences	Average number of states to accept	Average number of roots produced
G(0)	20	13.0	1.1
$G_9(1)$	99	29.6	1.1
$G_9(2)$	155	54.9	1.9
$G_9(3)$	171	65.9	2.3
$G_9(4)$	171	65.9	2.3
$G_9(5)$	183	68.8	2.3
$G_9(6)$	188	76.7	2.5
$G_9(7)$	189	77.2	2.5
$G_9(8)$	189	77.2	2.5
$G_9(9)$	192	80.9	2.6

Figure 9-20: The change in the cost of parsing User 9's sentences at Deviation-level 0 as a function of grammar growth.

Of all the users, User 9 shows the largest increases in both the average amount of search required to accept and the average number of explanations produced for each accepted sentence. The reason is simple: her idiosyncratic grammar included derived forms omitting almost every content word in the kernel. Specifically, by the end of session three she had successfully dropped most markers for postnominal cases, three of the four verbs, and two of the four **groupgathering** head nouns (not to mention a few other, less easily characterizable deletions). As a result, almost every sentence she typed in the last seven sessions that was not a request to see the calendar or airline schedule created at least two effectively equivalent meanings. Whatever markers she had not dropped by session three were eliminated after the supplementary instructions were given for session six. On the last day the simple sentence, "Dinner June 24 with Allen," created twelve roots at Deviation-level 0. A different decomposition of the kernel grammar into formclasses and forms would have had an enormous effect on the rise in ambiguity for this user without reducing the number of sentences accepted. The relationship between kernel design, adaptation, and ambiguity is discussed further in Chapter 10.

Evaluation of the protocols for the new users reveals nothing unexpected with respect to

Grammar after session(i)	Number of non-deviant sentences	Average number of states to accept	Average number of roots produced
G(0)	19	40.4	1.1
$G_{10}(1)$	121	37.7	1.6
$G_{10}(2)$	131	38.2	1.7
$G_{10}(3)$	135	40.3	1.7
$G_{10}(4)$	142	42.3	1.7
$G_{10}(5)$	142	42.3	1.7
$G_{10}(6)$	144	40.7	1.7
$G_{10}(7)$	147	42.8	1.7
$G_{10}(8)$	148	45.8	1.7
$G_{10}(9)$	148	45.8	1.7

Figure 9-21: The change in the cost of parsing User 10's sentences at Deviation-level 0 as a function of grammar growth.

CHAMP's competition mechanism. Figure 9-22 shows that most of the competitions that had a chance to be resolved were resolved the next time a competitor was required to understand a sentence.

	Number resolved competitions	Number of competitions left unresolved	Number left unresolved with chance to resolve
User 9	3	5	2
User 10	1	2	1

Figure 9-22: Competition results for Users 9 and 10.

The final measure described for Users 1, 2, 3, and 4 was across-user variability. We found a significant variation among the final grammars of the original users by examining the rate of acceptance for each user's sentences under her own final grammar and the final grammars of the others. Figure 9-23 shows the results for the same computation for Users 9 and 10.[46] The two grammars overlap by about half; as in Figure 9-13 the explanatory power of a user's individual grammar far outweighs that of another user.

[46]We do not measure the new sentences with the original users' grammars or the original sentences with the new users' grammars because of differences in the kernels under which each set of sentences was accepted. The differences in the kernel would show up as increased variability, biasing the results in adaptation's favor.

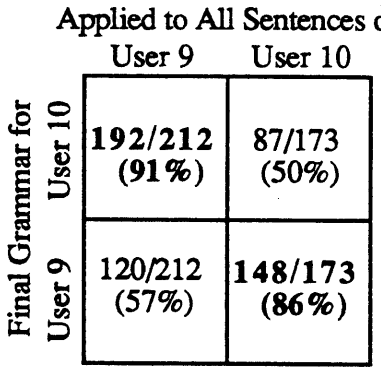

Figure 9-23: Measuring across-user variability for Users 9 and 10 by computing the acceptance rate for each user's utterances under her and the other user's final grammar.

9.2.4. Adaptive versus Monolithic Performance

Given the results for across-user variability, it is not surprising that a comparison of monolithic versus adapted performance for the test set from Users 9 and 10 reaffirms our previous conclusions. Figure 9-24 displays the increase in both search and the number of explanations produced under a monolithic grammar built from the union of the kernel and the final grammars of Users 9 and 10 with redundant components removed.

User	Average States$_{user}$	Average States$_{mono}$	Average Roots$_{user}$	Average Roots$_{mono}$	Additional sentences
9	80.6	96.7	2.6	2.6	0
10	45.8	68.2	1.7	3.4	0

Figure 9-24: A comparison of the relative costs of parsing with a monolithic grammar versus parsing with the adapted grammars for Users 9 and 10 (the last column shows the number of additional sentences for each user that could be parsed by the monolithic grammar but not the user's final grammar).

User 9 would notice little difference between interactions with a monolithic grammar and those with CHAMP. User 10, on other hand, would notice some difference, primarily in the number of interactions required to resolve the extra explanations produced by the monolithic interface. Most of the extra work required to accept User 10's sentences is contributed by User 9's grammar. In other words, in a monolithic design, User 10 must pay for User 9's idiosyncracy.

9.3. Summary

In total, CHAMP has been tested on 1042 utterances most of which represent unmodified spontaneous input by frequent users whose profession includes calendar scheduling as part of its duties. The full test set of 1042 sentences is given in Appendix B (with modifications indicated). We found no qualitative differences between CHAMP's performance for utterances in the original test set (gathered by simulation of the model) and the system's performance for spontaneous utterances from on-line interactions. The results of evaluating the interface can be summarized as follows:

1. CHAMP is able to understand about 84% of each user's utterances using error recovery, adaptation, and generalization to learn their idiosyncratic forms of expression.[47] This rate of acceptance was adequate for users to be able to accomplish the task.

2. CHAMP's acceptance rate is about eight percent lower than that predicted by the model for the same set of utterances. The difference is caused primarily by deviations that require the system to introduce a new subconstituent into a context that does not expect it. CHAMP's current understanding process can adapt only in the presence or absence of expected categories. Extending the system to include constructive inference would solve the problem.

3. Design choices such as the Single Segment Assumption, the Maximal Subsequence Heuristic, and the use of recovery-time constraints had only a small effect on CHAMP's acceptance rate. Given their power to constrain search, we consider these mechanisms to have been good solutions to some of the problems that arose in trying to realize the model computationally. The lexical extension problem, on the other hand, remains an open issue that cannot be ignored.

4. CHAMP's mixed liberal-conservative adaptation mechanism combines with generalization through the Formclass Hierarchy to learn the user's language effectively. The cost of learning is an increase in ambiguity in the current grammar. The increase seen in CHAMP reflects, in part, the inherent ambiguity in the user's idiolect. The remainder of the added ambiguity, however, comes from the decomposition of the initial experimental grammar into the particular set of formclasses and forms chosen for the kernel. Although competition keeps the effects of ambiguity from proliferating, and response times for most utterances were in a reasonable range, a better understanding of the relationship between kernel design, adaptation technique, and the rise in ambiguity would be worthwhile.

5. Given the chance to use their own natural language, users will manifest

---

[47]The average value of 84% reflects asymptotic behavior of the system over nine sessions. For Users 5 and 7 (see Appendix C), the acceptance rates are much lower because the number of sessions was low (three and five, respectively). The asymptotic value is a more reasonable measure of performance given the users' naturally self-bounded linguistic behavior.

significant variability in their preferred forms of expression for the same task. An adaptive interface like CHAMP accommodates this variability more efficiently than a monolithic, static design.

By the end of Chapter 3, we had proven all but one of our theoretical hypotheses:

H3: The Fixed Kernel Hypothesis: The kernel grammar for an adaptive parser need contain only a small set of system forms—in general only one form of reference for each meaningful domain action and object. Any user can be understood by then extending the grammar as a direct result of experience with the individual, using the paradigm of deviation detection and recovery in a least-deviant-first search.

Since we have shown that one adaptive interface design, CHAMP, is adequate to capture within-user consistency and accommodate across-user variability, we now conclude that the Fixed Kernel Hypothesis is proven as well.

Chapter 10
Critical Issues

Despite the evaluation's clear demonstration of the usefulness of adaptation at the interface, both theoretical and practical problems remain. The discussion in this chapter is offered as a kind of friendly warning to others interested in adaptive language acquisition for spontaneously generated input. The issues raised here are probably inherent in learning from real user interactions; regardless of the model you choose or the implementation you design, these problems are not likely to go away. The best one can do under such circumstances is to try to understand the tradeoffs involved in each issue and find the appropriate balance within one's particular framework.

10.1. The Effect of Kernel Design on Learning

The tenor of the discussion in Chapter 8 suggests that the choice of adaptation mechanism is the only variable affecting generalization. In reality, the structure of the kernel grammar is crucial as well.

The structural bias in CHAMP's kernel is toward undergeneralization. Consider the case of transposition over the kernel **changeforms**. Each of ACT3, ACT4, and ACT5 is a **changeform** which may contain an introductory adverbial phrase specifying the time or date of the **change** action (see Figure 10-1). Despite sharing steps 701, 702, and 703, the three strategies nevertheless use different steps to seek the source and target objects of the change. Because the three forms contain critical differences, an utterance that succeeds via one form cannot succeed via the others. Thus to learn to transpose one of the introductory adverbial phrases under every **change** context requires a separate deviant utterance—a separate learning episode—for each form. Many of the learning episodes in later sessions of the users' protocols reflect this particular source of undergeneralization.

Critical differences exist across formclasses too. Strategies in **addforms** and those in **deleteforms** contain critical differences with respect to each other and with respect to the members of **changeforms** as well. Still, each **addform** and each **deleteform** may be introduced by the same adverbial phrases as ACT3, ACT4, and ACT5. Again, each critically different form will require its own positive example to learn what could be considered a shared adaptation.

ACT3
strategy (701 702 703 706 709 711 799)
rnode (706 709 711)
unodes ((701 702 703))
mnodes ((701 702))

ACT4
strategy (701 702 703 706 710 799)
rnode (706 710)
unodes ((701 702 703))
mnodes ((701 702))

ACT5
strategy (701 702 703 706 709 723 724 799)
rnode (706 709 723 724)
unodes ((701 702 703))
mnodes ((701 702))

701: class m-hourforms
702: class m-intervalfroms
703: class m-dateforms
706: class changewd
709: class m-d-ggforms
710: class m-d-slotforms
711: class stpforms
723: class targetmkr
724: class m-i-ggforms
799: class eosmkr

ACT345
strategy (701 702 703 706 709 710 711 723 724 799)
rnode (706 709 710 723 724)
unodes ((701 702 703))
mnodes ((701 702) (709 710) (711 710) (723 710) (724 710) (723 711) (724 711))

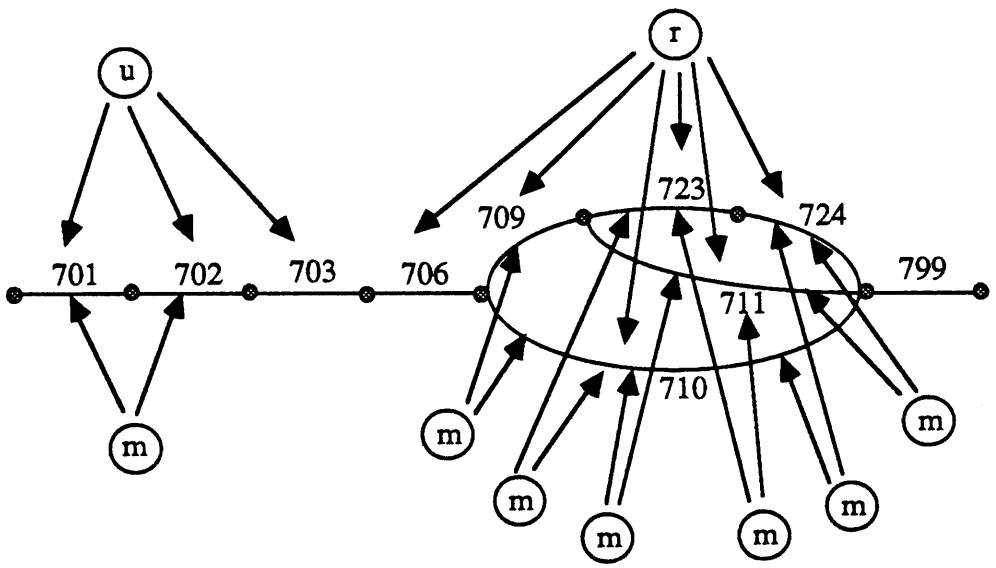

Figure 10-1: Three equivalent representations of the kernel changeforms ACT3, ACT4, and ACT5: as they appear in CHAMP, merged into a single caseframe, and as a shared, annotated graph structure. The graph is constructed by joining links representing strategy steps with unlabelled nodes. Variations on a straight path through the graph are signalled by control nodes ("u"= unordered, "r"= required, "m"= mutually-exclusive).

There are at least two ways to eliminate the bias toward undergeneralization in CHAMP's kernel. First, we could add a mechanism to adaptation that explicitly

compensates for the bias. Such a mechanism might, for example, look for all forms sharing the deviant step and then try to perform the same adaptation for each form. This approach would leave intact the search control provided by critical differences.

Alternatively, we could redesign the kernel so that all forms in a formclass are merged into a single form using mutual-exclusivity relations.[48] ACT345 in Figure 10-1 demonstrates how this approach would affect ACT3, ACT4, and ACT5 within the caseframe representation. ACT345 trades the search control provided by critical differences for the inclusion of more of the grammar in the adaptation context when the strategy succeeds. The bottom portion of the figure shows the representational shift that is the logical conclusion to the suggested redesign: the Formclass Hierarchy as a set of embeddable, shared, annotated graph structures, reminiscent of ATNs [64]. In this view, graph links correspond to strategy steps, while the parsing strategy itself corresponds to following a path through the graph subject to the controls represented by the arrows. Control arrows emanating from a node labelled "r" point to required links on the path just as the *rnode* field captures the required steps in the *strategy* list. Similarly, control arrows emanating from a "u" node point to links that may be unordered and those emanating from an "m" node point to mutually-exclusive links.

By bringing more of the grammar into the adaptation context—either explicitly at adaptation-time or through restructuring the kernel—both solutions clearly run the risk of overgeneralizing. In fact, either mechanism may derive components that increase search but serve no useful function in the grammar. Consider the case of a transposed step shared by ACT3, ACT4, and ACT5. In CHAMP's kernel, the utterance that succeeds via the transposition satisfies only one of the critically different forms. The order of the constituents in the utterance determines the new order of the strategy steps in the derived form. If we want to generalize across the formclass during adaptation, however, we must create new orderings for the other forms as well. Without an utterance containing the critically different constituents we have no way to dictate the ordering of the deviant step with respect to the steps seeking those unsatisfied constituents. Our only choice is to rely on some general method of reasoning (analogy, for example) to determine the orderings. Unless the reasoning method is fairly clever, however, the derived components it creates for the forms that did not actually succeed may never correspond to a structure produced by the user. In essence, we will have created exactly the kind of mismatch between the system's grammar and the user's grammar that we were trying to avoid by moving from a monolithic to an adaptive interface design.

The example teaches a simple lesson: any representation of the grammar biases what is

---

[48]The idea can be carried further: merging all the forms that share a step, for example, or even all the forms at the same Agenda-level.

learned from a single event. In building CHAMP, we made a conscious choice not to try to compensate for the kind of undergeneralization created by the kernel design. In general, the grammar design in any system that has fixed grammatical categories in fixed relations to each other will involve a similar choice.

10.2. The Effect of Kernel Design on Search

In an ideal implementation, the rise in ambiguity in an adapted grammar over time would reflect only the ambiguity inherent in the user's idiosyncratic language. In Chapter 9 we saw that CHAMP's performance is not ideal: the decomposition of the kernel grammar into a particular set of forms and formclasses contributed significantly to the rise in ambiguity in most of the users' adapted grammars.

To review the example presented in Chapter 9, we assume that previous interactions with the user have created both a derived form deleting the head noun in a **meetingform** (GG1') and a derived form deleting the hour marker in an **m-hourform** (HR0'). Under these circumstances the sentence "Schedule 7 p.m. on June 4 with John" produces two effectively equivalent roots. One tree contains "7 p.m." as a normal unmarked prenominal constituent in GG1' using the kernel **u-hourform**, HR1. The second tree explains the same time segment as a postnominal constituent in GG1' using the derived **m-hourform**, HR0'. If the marker for the date is also optional in the adapted grammar, and we change the sentence slightly to, "Schedule 7 p.m. June 4 with John," then four roots result: unmarked prenominal versus marked postnominal for each modifier. If the user has derived forms for more than one type of **groupgathering**, each of which has the head noun deleted, the same sentence produces four roots for each **groupgathering** that can be recognized without its head noun. The ambiguity that results from these simple deletions occurs because the combined effect of the adaptations eliminates the critical differences between formclasses. When neither the head noun nor the postnominal marker is required, an unmarked time segment legitimately serve two roles in the grammar without deviation: as a **u-hourform** and as an **m-hourform**.

The examples demonstrate how an interaction between adaptation and the kernel representational choices for noun phrases and prenominal and postnominal modifiers leads to ambiguity. Note that the combinatoric effect of even a small number of such deletions can be quite expensive at higher deviation-levels. Even if a grammar does not contain the additional deleted date marker, for example, a search that expands to Deviation-level 1 is likely to include a path that follows that interpretation; at each deviation-level we search as if the grammar contains derived components that compensate for the hypothesized deviation.

Although we have shown only one set of examples, there were other kernel choices that

led to similar interactions. While it is difficult to assign the blame to any one choice, there is no question that part of the rise in ambiguity seen in the users' grammars stems from these interactions rather than the ambiguity inherent in the users' idiolects.

In CHAMP, combinations of adaptations can interact to eliminate critical differences among forms and formclasses. The lesson is that an aspect of the knowledge representation that significantly controls search under some circumstances can be transformed by the adaptation mechanism into a source of increased search under other circumstances. In general it is important to make certain that the usefulness of the components we derive is proportional to their cost, but it may be that any seemingly efficient grammar organization can be made less efficient by the effects of adaptation.

10.3. The Effect of Search Constraints on System Predictability

CHAMP contains features other than critical differences that also help to control search: the Maximal Subsequence Heuristic, the Single Segment Assumption, and the various constraints that are applied at the different stages of processing (segmentation-time, bind-time, and so on). The performance evaluation in Chapter 9 showed that the number of sentences rejected by these features is fairly low. Given this result we might be tempted to conclude that the features are unqualifiedly advantageous. In reality, each feature contributes to rigid boundaries on understanding and learnability that the user may find perplexing.

Let's consider a few examples:

- In the session during which it is first encountered U7S33 ("change ending time from 4:00 to 4:30 for user studies lab with jill on june 11") is parsable at Deviation-level 1. Yet the same sentence is outside User 7's final grammar. How can this happen? The discrepancy is caused by an interaction between the Maximal Subsequence Heuristic and an adaptation during session five in which **for** is learned as an insertion. The effect of the interaction when User 7's final grammar is run on U7S33 is to make "4:30 for user studies lab with jill on june 11" maximal during each Parse/Recovery Cycle, with "4:30" interpreted as a prenominal time modifier and "for" ignored as an insertion. The maximal sequence masks the embedded and correct interpretation of "4:30" as the target end time. Had she typed a sentence like U7S33 after session five, User 7 might have thought it odd that a once acceptable structure was now rejected.

- How can "June 15 3 p.m." be a segment parsable at Deviation-level 0, when "June 7 3 p.m." is both unparsable and unlearnable with respect to every adapted grammar? The answer is: through an interaction between the Maximal Subsequence Heuristic and the bind-time constraint on **hour**. In the unlearnable segment "June 7 3 p.m." **3** has an interpretation as a value for **minutes** and **7** passes **hour**'s bind-time predicate. As a result, "7 3 p.m." becomes the maximal subsegment, masking the embedded subsegment "3 p.m." and ultimately causing the parse to fail. The problem does not arise for the segment "June 15 3 p.m."

because **15** does not satisfy **hour**'s bind-time predicate and thus never coalesces with the interpretation of **3** as minutes. Although she was the only user to encounter this apparent inconsistency, User 10 found the difference in the interface's responses for the relevant utterances confusing.

- When each of "Drew," "McDermott," and "speaking" is an unknown token, the segment "with Drew McDermott speaking about Non-Monotonic Logics" produces an interaction with the user in which she is asked whether or not "Drew McDermott speaking" is a member of one of the system's extendable classes. The segment "with Drew McDermott about Non-Monotonic Logics," on the other hand, produces an interaction referring to "Drew McDermott" alone. The difference between the interactions is caused by the Single Segment Assumption. Although users quickly learn to respond "no" in the first case, and "yes" in the second, they are nonetheless frustrated by the unpredictability of which type of event will occur for a given utterance.
- Any solution to the lexical extension problem is likely to exascerbate the effects of the Single Segment Assumption. Consider the phrase "on the topic of Non-Monotonic Logics." To compensate for new uses of known lexemes in CHAMP's kernel, this phrase must be marked by the hidden-operator prior to segmentation as follows: <on> <the> topic <of> Non <-> Monotonic Logics. Since the unbracketed tokens are truly unknown and the bracketed tokens are treated as unknowns, the entire phrase is presented to the user as a candidate new instance of an extendable class.
- Because of a **no-trans** recovery-time constraint, "cancel class 15-731" is acceptable, but "cancel the 15-731 class" is not. Similarly, the American date form "<month> <day> <year>" is parsable but the common European date form "<day> <month> <year>" is neither parsable nor learnable. In both examples, the recovery-time constraint is necessary to prevent more circumstances under which the Maximal Subsequence Heuristic masks the correct interpretation. To the user, however, there is no easily inferred explanation of why some transpositions are acceptable but others are not.

These examples demonstrate how a system may have to incorporate a tradeoff in at least two variables that affect the user's ability to accomplish the task efficiently: response time and the ease with which the user can predict the system's behavior. In CHAMP, the additional search control mechanisms probably help more than they hurt, although the user may never realize it.

10.4. The Lexical Extension Problem

When a word or phrase has one or more lexical definitions in the current grammar, none of which reflects the word's meaning in the current sentence, the system is faced with an instance of the *lexical extension problem.* In CHAMP, as in most expectation-driven parsers, when a word has definitions in the lexicon, those definitions are automatically indexed by the presence of the word in the input. Without some mechanism for ignoring the definitions that are inapplicable to the current situation, the knowledge the system thinks it has will cause the global parse to fail.

Examples of the lexical extension problem occur throughout the user data (see Appendix B). During evaluation we found that an average of fourteen percent of the utterances from all users contained instances of lexical extension. From a practical point of view, to reject fourteen percent of a user's utterances in addition to the average of sixteen percent already rejected by the implementation seems unreasonable. From a theoretical point of view, note that lexical extension can be thought of as the precursor of lexical ambiguity. Since lexical ambiguity is commonplace in natural language, we conclude that the lexical extension problem is not one we can ignore.

It is technically possible to solve the lexical extension problem within our adaptive model by brute force. When the parse fails globally, we could simply treat each token in turn as an unknown and reparse. If there is no case in which considering a single token as an unknown leads to global success, then we could try pairs, triples, and so on. In other words, we hypothesize single words as instances of lexical extension before we hypothesize increasingly large phrases as unknowns. Of course, the brute force approach is clearly intractable. In addition, it is doomed to produce vacuous results; most sentences are parsable (but meaningless) if we ignore large enough subsegments.

Unfortunately, there does not appear to be a simple solution to the lexical extension problem that is also computationally tractable. Some heuristics were suggested by an analysis of the experimental data but ultimately they proved untenable. About two-thirds of the instances of lexical extension in the user data corresponded to new uses of tokens from closed linguistic categories (articles, prepositions, punctuation). This suggests replacing the less ambiguous categories in the grammar with more ambiguous ones as a partial solution. For example, we might replace the relatively restricted class **datemkr** with the more general class **preposition**. The change permits any preposition to introduce the date; we no longer have to worry about unanticipated, idiosyncratic use of prepositions being unparsable in those circumstances. This solution is unsatisfactory in CHAMP because it is extremely costly at non-zero deviation-levels. If every preposition can mark every case, then most marked subsegments will succeed as most types of constituents given enough deviation points.

Another pattern that emerged from the data was that most instances of lexical extension occurred with respect to words in the kernel grammar; there were relatively few occurrences of users introducing lexical ambiguity in their idiosyncratic vocabulary. The pattern suggests modifying the brute force approach by relaxing kernel lexemes first (this would include members of closed linguistic categories). This solution also proved unsatisfactory in CHAMP because of the system's limit of two deviations. Most sentences that contained instances of lexical extension also contained other deviations. To increase the Maximum Deviation-level would have introduced significantly longer response times for all rejected sentences as well as longer response times and poorer explanations for many accepted ones.

It is interesting to note that little is said in the literature about the lexical extension problem. Most of the systems discussed in Section 2.5 (prior research in adaptation) make the explicit assumption that the lexicon contains either the applicable definition for a word or no definition at all. The real user input collected during our experiments shows how unreasonable that assumption is. Miller noticed the problem in the utterances for his users [41], but had no solution for it. Zernik's research on learning idioms [68] is the only work that seems to address the lexical extension problem head on. His system, RINA, tries to solve the problem by indexing the wrong definitions and then reasoning about the semantic inconsistences those definitions produce.

In essence, RINA performs a kind of deviation detection and error recovery over its semantic knowledge. In this respect Zernik's paradigm seems to fit in well with our own. Unfortunately, RINA incorporates the assumption that all deviations stem from semantic errors, never from syntactic ones. CHAMP assumes the opposite: any deviation can be explained syntactically but the semantics of the domain are fixed. Thus, to use Zernik's approach to solve the lexical extension problem within our adaptive model would seem to entail an exponential blow-up during search. Each time an expectation is violated we would have to consider both syntactic and semantic causes. In addition, there is the issue for CHAMP of the complexity of determining which semantic concepts are the inconsistent ones. By the time the global parse has failed at Deviation-level 2, there may be more than a thousand nodes on the Agenda. Allowed two deviation points, most subsegments will index more than one concept. How do we decide which concepts are legitimately indexed by deviant forms and which are erroneously indexed by instances of lexical extension? Are there useful heuristics to help make the determination?

Zernik's work does not address the inherent complexity of the problem because the examples he studied rarely indexed more than one suspect concept. Thus, despite providing a direction for future work, Zernik's research does not guarantee a tractable solution. The challenge to solve the lexical extension problem in real user data remains.

10.5. Summary

Each of the issues raised in this chapter represents a tradeoff along some dimension of what Watt, in 1968, called the "habitability" of the interface [61] and what we, in modern parlance, call its "user-friendliness." The choice of kernel representation involves balancing possible increases in response time from additional learning episodes against possible increases in response time from additional search. Kernel design also involves balancing response time during search in a tightly-constrained grammar against response time during search as adaptation negates those constraints. The choice of search control mechanisms involves juggling response time, rejection rate, and predictability in the behavior of the interface. In building and using lexical knowledge, there is a tradeoff

involved in the power of additional knowledge to increase understanding and the power of that same knowledge to confound a system that always assumes its knowledge is correct.

CHAMP represents a single point in the multidimensional space of tradeoffs just described. The placement and evaluation of other interfaces in that space should provide a characterization that suggests particular directions for advancing the quality and performance of future natural language interfaces.

Chapter 11
Conclusions and Future Directions

The thesis of this dissertation is that adaptive parsing is a desirable and expedient aspect of natural language interface design when users act in a linguistically idiosyncratic and self-bounded manner. In this chapter we review the main results that demonstrate this thesis and discuss future directions for research.

11.1. Main Results and Contributions

- *Idiosyncratic, self-bounded linguistic behavior under conditions of frequent use.* Through both hidden-operator and on-line experiments, we demonstrated that with frequent interactions a user limits herself to a restricted subset of forms that is significantly smaller than the full subset of natural language appropriate to the task, and quite different from the restricted subsets chosen by others.
    - Self-bounded behavior was demonstrated by constructing a performance graph for each user in an adaptive experimental condition. Each graph showed decreases over time in both the number of new grammatical constructions per utterance and the number of rejected utterances per session (Figures 3-3, 3-4, and 9-15).
    - Idiosyncracy was demonstrated by measuring overlap in the final, adapted grammars of users whose experimental conditions shared the same kernel grammar. To measure overlap, each user's sentences were parsed under her own final grammar and under the final grammar of each of the other users in her condition. The results showed both significant across-user variability and within-user consistency: acceptance rates by other users' final grammars ranged between eleven percent and fifty-nine percent, while acceptance rates by users' own final grammars ranged between eighty-one percent and ninety-one percent (Figures 9-13 and 9-23).
- *Adaptation, in theory.* The hidden-operator experiments also demonstrated that a model of adaptive language acquisition based on deviation detection, error recovery, and grammar augmentation is capable of learning different idiosyncratic grammars with a single, general mechanism.
    - Deviation detection and error recovery support a user's idiosyncratic style by increasing the likelihood that the user's natural form of expression will be accepted. Adaptation adds to that support by learning each explained deviation as an extension to the grammar; the ability to continually redefine

what is grammatical enables an adaptive interface to bootstrap itself toward a point where the system's grammar and the user's grammar converge. The success of adaptation as a learning method, in theory, was demonstrated by the convergence shown in the users' performance graphs. As simulated by the hidden-operator, the model was adequate to learn a number of idiosyncratic grammars, despite significant disparities between the kernel grammar and the preferred expressions of each user.

- The simulation also demonstrated that adaptation provides a more responsive environment than that provided by a static interface if the user's natural grammar differs significantly from the system's and the user herself adapts poorly. Given the same stimuli and a similar degree of initial deviation in her language with respect to the kernel, a user in the non-adaptive condition was unable to show the same improvements in her task performance as did users in the adaptive conditions.

- Adaptation supports the transition of an individual from naive to expert user by allowing her to develop a concise, idiosyncratic command language. This was demonstrated when users were given the supplementary instructions to work as quickly as possible. In response to the instructions, users tended to employ terser, less English-like forms. The same mechanism of deviation detection, error recovery and adaptation that had provided initial convergence was able to incorporate the new forms into the grammar and reestablish convergence with the user's new preferred language.

- *Adaptation, in practice.* CHAMP's performance for the utterances from the hidden-operator experiments and in on-line interactions with real users demonstrated the computational viability of the model. Evaluation of the system's performance for a test set of 1042 sentences from eight different users showed each of the following results:

  - CHAMP's performance was comparable to that of the model with respect to capturing within-user consistency. The performance graphs for Users 9 and 10 (Figure 9-15) clearly show the expected decrease over time of new constructions per utterance and rejections per session. With respect to Users 1, 2, 3, and 4, differences between the performance graphs under the simulation and the implementation were due largely to CHAMP's higher rejection rates and the methods used to compensate for discrepancies between the two conditions.

  - In addition to capturing within-user consistency, CHAMP was able to accommodate significant across-user variability. The idiosyncracy observed informally after the hidden-operator experiments was validated and quantified by the implementation.

  - CHAMP rejected about twice as many utterances as were predicted by the model, both in the hidden-operator and on-line experiments. The model accepted about ninety-two percent of each user's utterances while CHAMP's average acceptance rate was eighty-four percent. Although this rate of acceptance is well within the range of seventy-five to ninety percent suggested by Waltz [60] as necessary for natural language interfaces, the discrepancy between the model's rate and the implementation's rate indicates that there is room for improvement. An analysis of the reasons for

rejections suggests that a significant increase in acceptance rates is possible by extending the implementation to perform constructive inference.

- Acceptance of user utterances relied significantly on adaptation. This was demonstrated by a comparison of acceptance rates for each user's complete set of sentences under the kernel (from seven percent to twenty-four percent) and under the user's final grammar (from eighty-one percent to ninety-one percent).
- The adaptations themselves resulted in only a modest rise in ambiguity in each adapted grammar with CHAMP's competition mechanism helping to control the increase. Even with the rise in ambiguity, we demonstrated that performance under an idiosyncratic, adapted grammar is better than performance under a monolithic grammar with the same coverage. Adaptation produced significantly less search (thus, better response time) and fewer explanations (requiring less user interaction) than the monolithic approach.
- The system was able to perform effectively in the presence of spontaneous user input. No hidden-operator intervention was required in ninety-two percent of the utterances in the on-line experiments. Eight percent of the utterances did require intervention in the form of extra-grammatical markings to signal instances of lexical extension. Although no solution to the lexical extension problem is offered in this dissertation, some possible research paths were suggested in Chapter 10.
- Adaptation, as we have designed and implemented it, is a robust learning method; CHAMP was able to learn very different grammars, corresponding to very different linguistic styles, with a single general mechanism.

11.2. Future Directions

There are no easy solutions to the problems raised in Chapter 10, each demands extensive study. Here we present more immediate and tenable concerns: extensions to CHAMP and future directions for our particular style of adaptive interaction.

- As mentioned in Chapter 10, the Single Segment Assumption may be a source of frustration to the user when an unknown phrase contains an embedded instance of an extendable class. In the current version of CHAMP there is no way for the user to indicate that subsegments within the phrase serve distinct functional roles. It may be possible to eliminate the Single Segment Assumption by relying on user interaction to disambiguate segmentation. A relatively straightforward extension to the system would allow the user to designate the subsegment of an unknown phrase that corresponded to a new instance of a known category. In this way, the user could delineate "Non-monotonic Logics" as the true subject within the unknown phrase "on the topic of Non-monotonic Logics." The research issue is not whether such an extension is possible (it is), but whether users could employ it effectively.
- CHAMP's view that all deviations are created equal may be less useful than a view that considers some deviations more deviant than others. In the current implementation it takes one deviation point to compensate for a violated

expectation regardless of which expectation was violated, the cost of detecting the deviation, or the cost of explaining it. In addition, deviation is treated as strictly additive. We are no more confident of an explanation of deviations in separate constituents than we are of an explanation of the same number of deviations within a single constituent. Using simple additivity over equal deviation weights, the size of the search space and the quality of the explanations produced at higher deviation-levels forces CHAMP to restrict processing to its two-deviation limit. To break the two-deviation barrier, the system needs a more complex method for evaluating the effects of a deviation, based on either the cost of processing, the credibility of the resulting explanation, or both. An important question is whether we can increase the amount of deviance the system can tolerate, creating a more robust and flexible interface, without compromising performance.

- The adaptation mechanism implemented in CHAMP responds only to the presence or absence of *expected* constituents. As a result, we found that some portions of a user's idiosyncratic language were unreachable. Specifically, CHAMP cannot learn structures that can be explained only by the introduction of a subconstituent into a strategy not designed to expect it. The problem is clearly in the implementation, not the model; theoretically any set of constituents can be mapped into any other set of constituents using the four general recovery actions. The problem stems from the fact that insertion is not fully implemented in CHAMP—only meaningless tokens may be inserted into a constituent, not meaningful but unanticipated grammatical constituents.

  To overcome the problem we have suggested the use of a mechanism called *constructive inference*. As simulated by the hidden operator during the original experiments, constructive inference tries to build a legitimate database operation from the constituents that are present in the utterance. Constructive inference would be an extremely useful though non-trivial extension to CHAMP. To understand the difficulties involved, consider that when Deviation-level 2 is exhausted there may be more than a thousand constituents from which to try to create the intended action. We cannot afford to try all the combinations of constituents, nor can we assume that only the least-deviant ones or those covering the largest portions of the utterance are correct. In addition, even if we manage to construct one or more legitimate database actions, how do we then construct the new strategies to which they correspond? Since the inserted constituents carry meaning, they cannot be ignored in the way insertions are currently ignored in CHAMP. Yet as we demonstrated in Section 8.2, an insertion usually corresponds to a family of hypothetical explanations any of which may represent the correct or most useful generalization of the structure of the utterance. How do we decide which explanation to include? Could the competition mechanism accommodate the additional burden or would the system simply collapse under the load? These are questions that can and should be explored, although they may require significant changes in the implementation.

- Trying to integrate Zernik's work into our model of adaptation is important not only as a possible way of solving the lexical extension problem, but also as a way of further relaxing the Fixed Domain Assumption. CHAMP's only exception to the demand of a fully and correctly specified domain is its support of extendable classes. Even with this concession, the assumption seems unrealistic. Is it not likely that users are idiosyncratic in the way they think about a task as well as in the way

they express the thought? Is it not also likely that frequent performance of a task leads to the development of stereotypical subtasks which the user may want to reify? In the airline domain, for example, a user might ask the system to

"Center the trip to Boston around 2 p.m."[49]

The actual meaning of the sentence can be paraphrased: schedule a trip arriving in Boston around noon and leaving at about 4 p.m. (allowing the user time to get from the airport to her meeting and back). The required concept, that of **centering** a trip, is not included in CHAMP's kernel and is unlikely to be included *a priori* in any scheduling interface. zThe constituents from which such an idiosyncratic concept could be constructed are present in the kernel, however. A system combining the ideas in Zernik's research with our model of adaptation might be able to learn semantic idiosyncracy in essentially the same way it learns syntactic idiosyncracy: as deviation detection and recovery in the expectations of semantic forms.

- As demonstrated in the experiments, adaptation offers a partial solution to the problem of supporting the transition of users from naive to expert behavior. Others have shown that as the naive user becomes increasingly expert in her interactions she may want the convenience and speed usually associated only with a system-supplied command language. An adaptive interface permits the transition to an idiosyncratic command language as part of its natural function. Thus, despite the general validity of the Regularity Hypothesis, we expect that over sufficiently long periods the user's preferred language will change. In essence, we expect to see regularity for a time followed by mild destabilization as she introduces terser forms, followed by restabilization until, eventually, an idiosyncratic command language develops.

  Over the course of nine experimental sessions, we saw this kind of destabilization and reconvergence only when we introduced the supplementary instructions to work quickly. While the artificial impetus did demonstrate that CHAMP could accommodate the transition, we did not address the problem of how to remove obsolete forms once the new adapted grammar has stabilized. This is a crucial issue for long-term adaptation. Since forms may be tied to domain subtasks having irregular periodicity, it seems unlikely that a simple measure of frequency or recency would be adequate to determine when a form has become obsolete. A more practical measure might involve an evaluation of the degree of search constraint versus the degree of ambiguity contributed by a form over time. As an added benefit, the same evaluation function could supplant the competition mechanism CHAMP currently uses, providing a less arbitrary method for learning from multiple examples.

- Our model of adaptive language acquisition based on deviation detection, error recovery, and grammar augmentation is capable of learning different idiosyncratic grammars with a single, general mechanism. CHAMP implements the model using a modified bottom-up parsing algorithm and a decomposition of its kernel grammar into a particular caseframe representation. As an initial adaptive interface design CHAMP stands as a challenge to proponents of other parsing paradigms and

[49]The example was suggested by someone who actually uses this phrase with his travel agent.

grammar representations. Could a least-deviant-first search be accomplished as efficiently using an ATN-like representation such as the one suggested in Chapter 10? Is it possible to define processes analogous to deviation detection and recovery in a unification-based parser? In general we must ask how systems based on other paradigms and representations compare to CHAMP both in terms of what is theoretically learnable and in terms of performance for the same set of test sentences along dimensions such as rejection rate and response time.

- In Chapter 2 we referred to CHAMP's four general recovery strategies as a linguistic weak method for problem-solving when the understanding process goes awry. We argued that a relatively knowledge-free approach made sense in an initial exploration of adaptive language acquisition. Yet many of the poor explanations CHAMP produces are created because the system's recovery strategies are so linguistically naive. It would be interesting to see what the effect on performance would be if we replaced CHAMP's recovery strategies with more linguistically sophisticated knowledge. Would such a change create a more robust interface with better response, or would it simply create more rigidity and unpredictability from the user's point of view? To take the issue a step further, we might embed a competence model (such as transformational grammar [10]) in the recovery process as a way of pinpointing, and perhaps quantifying, the discrepancies between such models and real performance.

- We demonstrated the generality of our method of adaptation both across domains and across users. Still, a full demonstration of the model's generality demands at least two additional tests: across tasks and across natural languages. The creation of a new kernel for a non-scheduling task as well as the creation of a new kernel for a different natural language (especially a non-Romance and non-Germanic language) would undoubtedly provide additional insight into the interactions among the design of the kernel grammar, search, and adaptation.

Appendix A
CHAMP's Kernel Grammar

This appendix contains CHAMP's kernel lexicon, Formclass Hierarchy, and Concept Hierarchy for both the calendar and travel domains. Starred entries were added or modified before the on-line experiments with Users 9 and 10 for reasons explained in Chapter 9. With the exception of starred entries, CHAMP's kernel lexicon is taken entirely from the grammar developed for the hidden-operator experiments which were described in Chapter 3. The Formclass Hierarchy and Concept Hierarchy were created by combining the information in the grammar and domain definitions devised for the hidden-operator experiments with experience gained from working with the *development set* of utterances (eighty-five utterances from Users 1 and 2 pertaining only to the calendar domain). During the hidden-operator experiments we relied on a small, informal, BNF-style grammar constructed to provide a fairly minimal coverage of the stimuli. A subset of the instances of extendable classes required by the stimuli were chosen at random to be included in the dictionary for the experiments.

A.1. The Kernel Lexicon

```
;;;          VERBS

(delete  #s(LEX instof deletewd domain b))
(cancel  #s(LEX instof deletewd domain b))

(add #s(LEX instof addwd domain b))
(schedule  #s(LEX instof addwd domain b))
(go #s(LEX instof gowd domain t))

(change #s(LEX instof changewd domain b))
*((will be from) #s(LEX instof auxverb+ivlmkr domain c))
*((will be at) #s(LEX instof auxverb+hourmkr domain c))
*((will be on) #s(LEX instof auxverb+datemkr domain c))
*((will be in) #s(LEX instof auxverb+locmkr domain c))
*((will be with) #s(LEX instof auxverb+prtmkr domain c))
*((will be about) #s(LEX instof auxverb+sbjmkr domain c))

(show #s(LEX instof showwd domain b))
```

```
((show me) #s(LEX instof showwd domain b))
(display #s(LEX instof showwd domain b))

;;;          HEAD NOUNS

(calendar #s(LEX instof calendarwd domain c))
(schedule #s(LEX instof calendarwd domain c))
(meeting  #s(LEX instof meetingwd domain c))
(appointment #s(LEX instof meetingwd domain c))
(class  #s(LEX instof classwd domain c))
(seminar #s(LEX instof seminarwd domain c))
(meal #s(LEX instof mealwd domain c))
(breakfast #s(LEX instof mealwd def breakfast domain c))
(breakfast #s(LEX instof intervalwd def (800 *) domain c))
(lunch #s(LEX instof mealwd def lunch domain c))
(lunch #s(LEX instof intervalwd def (1200 *) domain c))
(dinner #s(LEX instof mealwd def dinner domain c))
(dinner #s(LEX instof intervalwd def (1900 *) domain c))

((airline schedule) #s(LEX instof fschedwd domain t))
(flight #s(LEX instof flightwd domain t))
(trip #s(LEX instof tripwd domain t))

;;;          SLOT IDENTIFIERS

(location #s(LEX instof locslotwd domain c))
(speaker #s(LEX instof partslotwd domain c))
(company #s(LEX instof locslotwd domain c))
(company #s(LEX instof partslotwd domain c))
(subject #s(LEX instof subjslotwd domain c))
(type #s(LEX instof gentopslotwd domain c))

(source #s(LEX instof originslotwd domain t))
(destination #s(LEX instof destslotwd domain t))
(arrival #s(LEX instof originslotwd domain t))
(arrival #s(LEX instof startslotwd domain t))
(departure #s(LEX instof destslotwd domain t))
(departure #s(LEX instof endslotwd domain t))

((start time) #s(LEX instof startslotwd domain b))
((beginning time)  #s(LEX instof startslotwd domain b))
(beginning #s(LEX instof startslotwd domain b))
(start #s(LEX instof startslotwd domain b))
((end time) #s(LEX instof endslotwd domain b))
((ending time)  #s(LEX instof endslotwd domain b))
(ending #s(LEX instof endslotwd domain b))
(end #s(LEX instof endslotwd domain b))
(finish #s(LEX instof endslotwd domain b))
(time #s(LEX instof timeslotwd domain b))
(time #s(LEX instof endslotwd domain b))
```

```
(time #s(LEX instof startslotwd domain b))
(day #s(LEX instof dateslotwd domain b))

;;;          LOCATION WORDS

((New York) #s(LEX instof cityname def NY domain t))
(NY #s(LEX instof cityname domain t))
(Pittsburgh #s(LEX instof cityname def Pgh domain t))
(Pgh #s(LEX instof cityname domain t))
(Chicago #s(LEX instof cityname def Chi domain t))
(Chi #s(LEX instof cityname domain t))

((Carnegie Mellon University) #s(LEX instof schoolname def CMU domain b))
(CMU #s(LEX instof schoolname domain b))

((Station Square) #s(LEX instof buildingname domain c))
((Station Square) #s(LEX instof tbuildingname domain t))
(airport #s(LEX instof tbuildingname domain t))
((the airport) #s(LEX instof tbuildingname domain t def airport))

(aisys #s(LEX instof businessname domain b))
((vc incorporated) #s(LEX instof businessname domain b))

(room #s(LEX instof roomwd domain c))
(office #s(LEX instof roomname domain c))
((the office) #s(LEX instof roomname domain c def office))

;;;          TIME AND DATE WORDS

(June #s(LEX instof monthwd def 6 domain b))

(noon #s(LEX instof hourname def 1200 domain b))

((o %apost clock) #s(LEX instof clockwd domain b))

(am #s(LEX instof daywd domain b))
((a %period m %period) #s(LEX instof daywd domain b))
(pm #s(LEX instof nightwd domain b))
((p %period m %period) #s(LEX instof nightwd domain b))

;;;          PEOPLE WORDS

(dad #s(LEX instof familyname def father domain c))
(father #s(LEX instof familyname domain c))

(John #s(LEX instof studentname domain c))
(Anderson #s(LEX instof profname domain c))
(Newell #s(LEX instof profname domain c))

(vc #s(LEX instof groupname domain c))
```

```
;;;          TOPIC/SUBJECT WORDS

((Natural Language Interfaces) #s(LEX instof gentopicname domain c))
(cogsci #s(LEX instof gentopicname domain c))
(ai #s(LEX instof gentopicname domain c))

(speech #s(LEX instof projectname domain c))
((speech research) #s(LEX instof subjname domain c))
(financial #s(LEX instof subjname domain c))
(project #s(LEX instof projwd domain c))

;;;          MARKERS

(about #s(LEX instof sbjmkr domain c))

(at #s(LEX instof hrmkr domain c))
(at #s(LEX instof dephrmkr domain t))
(at #s(LEX instof arrhrmkr domain t))
(at #s(LEX instof locmkr domain c))
((arriving at) #s(LEX instof arrhrmkr domain t))
((departing at) #s(LEX instof dephrmkr domain t))

(between #s(LEX instof ivlmkr domain b))
(and #s(LEX instof ehrmkr domain b))

(for #s(LEX instof tmdtmkr domain c))
(for #s(LEX instof depdtmkr domain t))
(for #s(LEX instof tmhrmkr domain c))
(for #s(LEX instof dephrmkr domain t))
(for #s(LEX instof tmivlmkr domain c))

(from #s(LEX instof ivlmkr domain b))
(from #s(LEX instof sourcemkr domain b))
(from #s(LEX instof originmkr domain t))
(from #s(LEX instof dephrmkr domain t))
((going from) #s(LEX instof originmkr domain t))

(in #s(LEX instof locmkr domain c))
((instead of) #s(LEX instof sourcemkr2 domain c))

(of #s(LEX instof mnthmkr domain b))
(of #s(LEX instof mdobjmkr domain b))

(on #s(LEX instof dtmkr domain c))
(on #s(LEX instof depdtmkr domain t))
((arriving on) #s(LEX instof depdtmkr domain t))
((departing on) #s(LEX instof depdtmkr domain t))

(to #s(LEX instof ehrmkr domain b))
(to #s(LEX instof arrhrmkr domain t))
```

```
(to #s(LEX instof targetmkr domain b))
(to #s(LEX instof destmkr domain t))
(to #s(LEX instof dlocmkr domain t))
((going to) #s(LEX instof destmkr domain t))

(with #s(LEX instof prtmkr domain c))

(the #s(LEX instof defmkr domain b))

(a #s(LEX instof indefmkr domain b))
(an #s(LEX instof indefmkr domain b))

;;;          PUNCTUATION

(%colon #s(LEX instof colonsymb domain b))
(%comma #s(LEX instof commasymb domain b))
(%period #s(LEX instof eosmkr domain b))
(%questionmark #s(LEX instof eosmkr domain b))

(%hyphen #s(LEX instof ehrmkr domain b))
(%hyphen #s(LEX instof dlocmkr domain t))
(%hyphen #s(LEX instof hyphensymb domain c))

(%pound #s(LEX instof poundsymb domain t))
```

A.2. The Formclass Hierarchy

```
;;;              MISCELLANEOUS

(gentopicforms #s(CL isa domainroot domain c cept gentopic alevel 0))
(ifgentop #s(ST isa gentopicforms strategy (15) mode (15)))
(15 #s(LN class gentopicname))

(classnumforms
 #s(CL isa domainroot domain c cept classnumber alevel 0 ig (ig-clnum res)))
(clnum1 #s(ST isa classnumforms strategy (12 13 14) mode (12 13 14)))
(12 #s(LN class number bindvar level))
(13 #s(LN class hyphensymb))
(14 #s(LN class number bindvar id))

(u-subjectforms #s(CL isa domainroot domain c cept subject alevel 1))
(subj1 #s(ST isa u-subjectforms strategy (125) mode (125)))
(ifsubj #s(ST isa u-subjectforms strategy (129) mode (129)))
(125 #s(LN class tokenseq))
(129 #s(LN class subjname))

(m-subjectforms #s(CL isa domainroot domain c cept subject alevel 2))
(subj0 #s(ST isa m-subjectforms strategy (213 214) mode (213 214)))
```

```
(213 #s(LN class sbjmkr bindvar marker))
(214 #s(LN class u-subjectforms bindvar (subject dsubj)))

;;;              PEOPLE

(u-participantforms #s(CL isa domainroot domain c cept participants alevel 1))
(u-groupforms #s(CL isa u-participantforms domain c cept group alevel 1))
(u-personforms #s(CL isa u-participantforms domain c cept person alevel 1))

(ifgroup1 #s(ST isa u-groupforms strategy (114) rmode (114)))
(ifgroup2 #s(ST isa u-groupforms strategy (115) rmode (115)))
(ifgroup3 #s(ST isa u-groupforms strategy (116 117) rmode (116)))
(ifgroup4 #s(ST isa u-groupforms strategy (150) rmode (150)))
(projectname #s(CL isa domainroot domain c cept project alevel 1))
(114 #s(LN class businessname bindvar (instance majorloc)))
(115 #s(LN class schoolname bindvar (instance majorloc)))
(116 #s(LN class projectname))
(117 #s(LN class projwd bindvar postmarker))
(150 #s(LN class groupname bindvar instance))

(ifperson1 #s(ST isa u-personforms strategy (118) rmode (118)))
(ifperson2 #s(ST isa u-personforms strategy (119) rmode (119)))
(ifperson3 #s(ST isa u-personforms strategy (120) rmode (120)))
(118 #s(LN class studentname bindvar instance))
(119 #s(LN class profname bindvar instance))
(120 #s(LN class familyname bindvar instance))

(m-participantforms #s(CL isa domainroot domain c cept participants alevel 2))
(part0 #s(ST isa m-participantforms strategy (207 208) rmode (207 208)))
(207 #s(LN class prtmkr bindvar marker))
(208 #s(LN class u-participantforms bindvar (participant dparts)))

;;;              PLACES

(u-locationforms
 #s(CL isa domainroot domain c cept location alevel 1 ig (ig-loc res)))
(ifloc2 #s(ST isa u-locationforms strategy (114) rmode (114)))
(ifloc3 #s(ST isa u-locationforms strategy (115) rmode (115)))
(ifloc4 #s(ST isa u-locationforms strategy (122 123) rmode (122 123)))
(ifloc5 #s(ST isa u-locationforms strategy (128) rmode (128)))

(buildingforms
 #s(CL isa u-locationforms domain c cept location alevel 0 ig (ig-building res)))
(loc7 #s(ST isa buildingforms strategy (133) rmode (133)))
(133 #s(LN class buildingname bindvar (instance majorloc)))

(roomforms
 #s(CL isa u-locationforms domain c cept location alevel 0 ig (ig-room res)))
(loc6 #s(ST isa roomforms strategy (10 11) rmode (10 11)))
(10 #s(LN class roomwd bindvar marker))
```

```
(11 #s(LN class number bindvar (roomnumber location minorloc)))

(122 #s(LN class buildingforms bindvar majorloc))
(123 #s(LN class roomforms bindvar minorloc))
(128 #s(LN class roomname bindvar (instance minorloc)))

(u-triplocforms
 #s(CL isa domainroot domain t cept triploc alevel 1 ig (ig-triploc res)))
(ifloc01 #s(ST isa u-triplocforms strategy (121) mode (121)))
(ifloc02 #s(ST isa u-triplocforms strategy (114) mode (114)))
(ifloc03 #s(ST isa u-triplocforms strategy (134) mode (134)))
(ifloc04 #s(ST isa u-triplocforms strategy (115) mode (115)))

(121 #s(LN class cityname bindvar (instance majorloc)))
(134 #s(LN class tbuildingname bindvar (instance majorloc)))

(citypairforms #s(CL isa domainroot domain t cept citypair alevel 2))
(loc00 #s(ST isa citypairforms strategy (226 227 228) mode (226 227 228)))
(226 #s(LN class cityname bindvar (instance dorig origin)))
(227 #s(LN class dlocmkr bindvar marker))
(228 #s(LN class cityname bindvar (instance ddest destination)))

(m-locationforms #s(CL isa domainroot domain c cept location alevel 2))
(loc0 #s(ST isa m-locationforms strategy (211 212) mode (211 212)))
(m-originlocforms #s(CL isa domainroot domain t cept triploc alevel 2))
(loc01 #s(ST isa m-originlocforms strategy (222 224) mode (222 224)))
(m-destlocforms #s(CL isa domainroot domain t cept triploc alevel 2))
(loc02 #s(ST isa m-destlocforms strategy (223 225) mode (223 225)))

(211 #s(LN class locmkr bindvar marker))
(212 #s(LN class u-locationforms bindvar (location dloc)))
(222 #s(LN class originmkr bindvar marker))
(223 #s(LN class destmkr bindvar marker))
(224 #s(LN class u-triplocforms bindvar (origin dorig oloc)))
(225 #s(LN class u-triplocforms bindvar (destination ddest destloc)))

;;;                HOURFORMS

(u-hourforms
 #s(CL isa domainroot domain b cept hour alevel 0 ig (ig-hour exp)))
(hr1 #s(ST isa u-hourforms
  strategy (4 7 8 16)
  mode (4)
  mnodes ((8 16))))
(hr2 #s(ST isa u-hourforms
  strategy (4 5 6 7 8 16)
  mode (4 5 6)
  mnodes ((8 16))))

(4 #s(LN class number bindvar hr))
```

```
(5 #s(LN class colonsymb))
(6 #s(LN class number bindvar minutes))
(7 #s(LN class clockwd))
(8 #s(LN class daywd bindvar dnbit))
(16 #s(LN class nightwd bindvar dnbit))
(17 #s(LN class hourname bindvar instance))

(m-hourforms #s(CL isa domainroot domain c cept hour alevel 1))
(hr0 #s(ST isa m-hourforms strategy (111 113) mode (111 113)))
(m-dephrforms #s(CL isa domainroot domain t cept hour alevel 1))
(dhr0 #s(ST isa m-dephrforms strategy (130 113) mode (130 113)))
(m-arrhrforms #s(CL isa domainroot domain t cept hour alevel 1))
(ahr0 #s(ST isa m-arrhrforms strategy (131 132) mode (131 132)))
(tm-hourforms #s(CL isa domainroot domain c cept hour alevel 1))
(hr00 #s(ST isa tm-hourforms strategy (112 113) mode (112 113)))

(111 #s(LN class hrmkr bindvar marker))
(130 #s(LN class dephrmkr bindvar marker))
(131 #s(LN class arrhrmkr bindvar marker))
(112 #s(LN class tmhrmkr bindvar marker))
(113 #s(LN class u-hourforms bindvar (hour dshour ddep departure)))
(132 #s(LN class u-hourforms bindvar (hour darr arrival)))

;;;              INTERVALFORMS

(u-intervalforms
 #s(CL isa domainroot domain b cept interval alevel 1 ig (ig-int exp)))
(int1 #s(ST isa u-intervalforms strategy (107 108 109) mode (107 108 109)))
(int2 #s(ST isa u-intervalforms strategy (110) mode (110)))

(107 #s(LN class u-hourforms bindvar starthour))
(108 #s(LN class ehrmkr bindvar emarker))
(109 #s(LN class u-hourforms bindvar endhour))
(110 #s(LN class intervalwd bindvar instance))

(m-intervalforms #s(CL isa domainroot domain b cept interval alevel 2))
(int0 #s(ST isa m-intervalforms strategy (204 206) mode (204 206)))
(tm-intervalforms #s(CL isa domainroot domain c cept interval alevel 2))
(int00 #s(ST isa tm-intervalforms strategy (205 206) mode (205 206)))

(204 #s(LN class ivlmkr bindvar marker))
(205 #s(LN class tmivlmkr bindvar marker))
(206 #s(LN class u-intervalforms bindvar (interval dinterval)))

;;;              DATEFORMS

(u-dateforms
 #s(CL isa domainroot domain b cept date alevel 1 ig (ig-date res)))
(date1 #s(ST isa u-dateforms strategy (101 102 103 105) mode (101 102)))
(date2 #s(ST isa u-dateforms strategy (102 104 101 105) mode (101 102 104)))
```

```
(ifdate #s(ST isa u-dateforms strategy (127) mode (127)))

(101 #s(LN class monthwd bindvar month))
(102 #s(LN class number bindvar day))
(103 #s(LN class commasymb))
(104 #s(LN class mnthmkr))
(105 #s(LN class number bindvar year))
(127 #s(LN class datename bindvar instance))

(m-dateforms #s(CL isa domainroot domain c cept date alevel 2))
(date0 #s(ST isa m-dateforms strategy (201 203) mode (201 203)))
(m-depdtforms #s(CL isa domainroot domain t cept date alevel 2))
(depdate0 #s(ST isa m-depdtforms strategy (220 203) mode (220 203)))
(tm-dateforms #s(CL isa domainroot domain c cept date alevel 2))
(date00 #s(ST isa tm-dateforms strategy (202 203) mode (202 203)))

(201 #s(LN class dtmkr bindvar marker))
(220 #s(LN class depdtmkr bindvar marker))
(202 #s(LN class tmdtmkr bindvar marker))
(203 #s(LN class u-dateforms bindvar (date ddate)))

;;;              CALSEGFORMS & AIRSEGFORMS

(calsegforms
 #s(CL isa domainroot domain c cept calseg alevel 3 ig (ig-calseg res)))
(calseg1 #s(ST isa calsegforms
  strategy (301 302 303 330 331 314 315 332 333 316)
  mode (330)
  unodes ((301 302 303) (331 332 333 314 315 316))
  mnodes ((303 316 331) (301 302 314 315 332 333) (331 332 333))))
(airsegforms
 #s(CL isa domainroot domain t cept airseg alevel 3 ig (ig-flight res)))
(airseg1 #s(ST isa airsegforms
  strategy (302 303 361 362 358 356 357 354 355)
  mode (362)
  unodes ((302 303 361) (358 356 357 354 355))
  mnodes ((302 356) (302 357) (303 358) (361 354) (361 355))))

(330 #s(LN class calendarwd bindvar head))
(331 #s(LN class tm-dateforms bindvar post))
(332 #s(LN class tm-intervalforms bindvar post))
(333 #s(LN class tm-hourforms bindvar post))
(361 #s(LN class citypairforms bindvar pre))
(362 #s(LN class fschedwd bindvar head))

;;;              SRCFORMS & TGTFORMS

(srcforms #s(CL isa domainroot domain c))
(asrcforms #s(CL isa domainroot domain t))
(tgtforms #s(CL isa domainroot domain c))
```

(atgtforms #s(CL isa domainroot domain t))

(intsrcforms #s(CL isa srcforms domain c cept source alevel 3))
(inttgtforms #s(CL isa tgtforms domain c cept target alevel 3))
(intsrc1 #s(ST isa intsrcforms strategy (321 301) mode (321 301)))
(inttgt1 #s(ST isa inttgtforms strategy (322 323) mode (322 323)))
(hrsrc1forms #s(CL isa srcforms domain c cept source alevel 3))
(hrtgt1forms #s(CL isa tgtforms domain c cept target alevel 3))
(hrsrc2forms #s(CL isa srcforms domain c cept source alevel 3))
(hrtgt2forms #s(CL isa tgtforms domain c cept target alevel 3))
(hrsrc1 #s(ST isa hrsrc1forms strategy (321 302) mode (321 302)))
(hrtgt1 #s(ST isa hrtgt1forms strategy (322 324) mode (322 324)))
(hrsrc2 #s(ST isa hrsrc2forms strategy (321 363) mode (321 363)))
(hrtgt2 #s(ST isa hrtgt2forms strategy (322 364) mode (322 364)))
(datesrcforms #s(CL isa srcforms domain c cept source alevel 3))
(datetgtforms #s(CL isa tgtforms domain c cept target alevel 3))
(datesrc1 #s(ST isa datesrcforms strategy (321 303) mode (321 303)))
(datetgt1 #s(ST isa datetgtforms strategy (322 325) mode (322 325)))
(locsrcforms #s(CL isa srcforms domain c cept source alevel 3))
(loctgtforms #s(CL isa tgtforms domain c cept target alevel 3))
(locsrc1 #s(ST isa locsrcforms strategy (321 304) mode (321 304)))
(loctgt1 #s(ST isa loctgtforms strategy (322 326) mode (322 326)))
(partsrcforms #s(CL isa srcforms domain c cept source alevel 3))
(parttgtforms #s(CL isa tgtforms domain c cept target alevel 3))
(partsrc1 #s(ST isa partsrcforms strategy (321 305) mode (321 305)))
(parttgt1 #s(ST isa parttgtforms strategy (322 327) mode (322 327)))
(subjsrcforms #s(CL isa srcforms domain c cept source alevel 3))
(subjtgtforms #s(CL isa tgtforms domain c cept target alevel 3))
(subjsrc1 #s(ST isa subjsrcforms strategy (321 307) mode (321 307)))
(subjtgt1 #s(ST isa subjtgtforms strategy (322 328) mode (322 328)))
(gentopsrcforms #s(CL isa srcforms domain c cept source alevel 3))
(gentoptgtforms #s(CL isa tgtforms domain c cept target alevel 3))
(gentopsrc1 #s(ST isa gentopsrcforms strategy (321 308) mode (321 308)))
(gentoptgt1 #s(ST isa gentoptgtforms strategy (322 329) mode (322 329)))

(aintsrcforms #s(CL isa asrcforms domain t cept source alevel 3))
(ainttgtforms #s(CL isa atgtforms domain t cept target alevel 3))
(intsrc01 #s(ST isa aintsrcforms strategy (321 301) mode (321 301)))
(inttgt01 #s(ST isa ainttgtforms strategy (322 323) mode (322 323)))
(ahr1srcforms #s(CL isa asrcforms domain t cept source alevel 3))
(ahr1tgtforms #s(CL isa atgtforms domain t cept target alevel 3))
(ahr2srcforms #s(CL isa asrcforms domain t cept source alevel 3))
(ahr2tgtforms #s(CL isa atgtforms domain t cept target alevel 3))
(hrsrc01 #s(ST isa ahr1srcforms strategy (321 302) mode (321 302)))
(hrtgt01 #s(ST isa ahr1tgtforms strategy (322 324) mode (322 324)))
(hrsrc01 #s(ST isa ahr2srcforms strategy (321 363) mode (321 363)))
(hrtgt02 #s(ST isa ahr2tgtforms strategy (322 364) mode (322 364)))
(adatesrcforms #s(CL isa asrcforms domain t cept source alevel 3))
(adatetgtforms #s(CL isa atgtforms domain t cept target alevel 3))

```
(datesrc01 #s(ST isa adatesrcforms strategy (321 303) mode (321 303)))
(datetgt01 #s(ST isa adatetgtforms strategy (322 325) mode (322 325)))
(aloc1srcforms #s(CL isa asrcforms domain t cept source alevel 3))
(aloc1tgtforms #s(CL isa atgtforms domain t cept target alevel 3))
(aloc2srcforms #s(CL isa asrcforms domain t cept source alevel 3))
(aloc2tgtforms #s(CL isa atgtforms domain t cept target alevel 3))
(locsrc01 #s(ST isa aloc1srcforms strategy (321 349) mode (321 349)))
(loctgt01 #s(ST isa aloc1tgtforms strategy (322 360) mode (322 360)))
(locsrc02 #s(ST isa aloc2srcforms strategy (321 365) mode (321 365)))
(loctgt02 #s(ST isa aloc2tgtforms strategy (322 366) mode (322 366)))

(321 #s(LN class sourcemkr bindvar smarker))
(322 #s(LN class targetmkr bindvar tmarker))
(323 #s(LN class u-intervalforms bindvar (target dinterval)))
(324 #s(LN class u-hourforms bindvar (target dshour ddep)))
(325 #s(LN class u-dateforms bindvar (target ddate)))
(326 #s(LN class u-locationforms bindvar (target dloc)))
(327 #s(LN class u-participantforms bindvar (target dparts)))
(328 #s(LN class u-subjectforms bindvar (target dsubj)))
(329 #s(LN class gentopicforms bindvar (target dgentop)))
(360 #s(LN class u-triplocforms bindvar (target dorig)))
(363 #s(LN class u-hourforms bindvar (source dehour darr)))
(364 #s(LN class u-hourforms bindvar (target dehour darr)))
(365 #s(LN class u-triplocforms bindvar (source ddest)))
(366 #s(LN class u-triplocforms bindvar (target ddest)))

;;;          GROUPGATHERING & TRAVEL OBJECT FORMS

(mrggforms #s(CL isa domainroot domain c))
(meetingforms
 #s(CL isa mrggforms domain c cept meeting alevel 3 ig (ig-mrgg res)))
(gg1 #s(ST isa meetingforms
  strategy (301 302 303 304 305 307 309 314 315 316 317 318 319)
  mode (309)
  unodes ((301 302 303 304 305 307) (314 315 316 317 318 319))
  mnodes ((301 302 314 315) (303 316) (304 317) (305 319) (307 318))
  snode (left 314 315 316 317 318 319)))
(seminarforms
 #s(CL isa mrggforms domain c cept seminar alevel 3 ig (ig-mrgg res)))
(gg2 #s(ST isa seminarforms
  strategy (301 302 303 304 305 308 310 314 315 316 317 318 319)
  mode (310)
  unodes ((301 302 303 304 305) (314 315 316 317 318 319))
  mnodes ((301 302 314 315) (303 316) (304 317) (305 319))
  snode (left 314 315 316 317 318 319)))
(classforms
 #s(CL isa mrggforms domain c cept class alevel 3 ig (ig-mrgg res)))
(gg3 #s(ST isa classforms
  strategy (301 302 303 304 308 311 312 314 315 316 317 318)
  mode (311)
```

```
  unodes ((301 302 303 304) (314 315 316 317))
  mnodes ((301 302 314 315) (303 316) (304 317) (308 312))
  snode (left 314 315 316 317 318)))
(uggforms #s(CL isa domainroot domain c))
(mealforms #s(CL isa uggforms domain c cept meal alevel 3 ig (ig-ugg res)))
(gg4 #s(ST isa mealforms
  strategy (301 302 303 304 305 307 313 314 315 316 317 318 319)
  rnode (313)
  unodes ((301 302 303 304 305 307) (314 315 316 317 318 319))
  mnodes ((301 302 314 315) (303 316) (304 317) (305 319) (307 318))
  snode (left 314 315 316 317 318 319)))
(mrtripforms #s(CL isa domainroot domain t))
(fl1forms
 #s(CL isa mrtripforms domain t cept flight alevel 3 ig (ig-flight res)))
(fl1 #s(ST isa fl1forms
  strategy (301 302 303 349 350 354 355 314 356 357 358)
  rnode (350)
  unodes ((301 302 303 349) (354 355 314 356 357 358))
  mnodes ((301 302 356 314) (301 302 357 314) (303 358) (349 354) (349 355))
  snode (left 354 355 314 356 357 358)))
(tripforms
 #s(CL isa mrtripforms domain t cept trip alevel 3 ig (ig-trip res)))
(trip1 #s(ST isa tripforms
  strategy (301 302 303 349 351 354 355 314 356 357 358)
  rnode (351)
  unodes ((301 302 303 349) (354 355 314 356 357 358))
  mnodes ((301 302 356 314) (301 302 357 314) (314 356) (314 357)
        (303 358) (349 354) (349 355))
  snode (left 354 355 314 356 357 358)))
(utripforms #s(CL isa domainroot domain t))
(fl2forms
 #s(CL isa utripforms domain t cept flight alevel 3 ig (ig-flight res)))
(fl2 #s(ST isa fl2forms
  strategy (359 350 352 353 354 355 314 356 357 358)
  rnode (350 353)
  unodes ((354 355 314 356 357 358))
  mnodes ((314 356) (314 357))
  snode (left 354 355 314 356 357 358)))

(301 #s(LN class u-intervalforms bindvar (source dinterval pre)))
(302 #s(LN class u-hourforms bindvar (source dshour ddep pre)))
(303 #s(LN class u-dateforms bindvar (source ddate pre)))
(304 #s(LN class u-locationforms bindvar (source dloc pre)))
(305 #s(LN class u-participantforms bindvar (source dparts participant pre)))
(307 #s(LN class u-subjectforms bindvar (source dsubj pre)))
(308 #s(LN class gentopicforms bindvar (source dgentop pre)))
(309 #s(LN class meetingwd bindvar head))
(310 #s(LN class seminarwd bindvar head))
(311 #s(LN class classwd bindvar head))
(312 #s(LN class classnumforms bindvar (dgentop post)))
```

```
(313 #s(LN class mealwd bindvar (mealtype head)))
(314 #s(LN class m-intervalforms bindvar post))
(315 #s(LN class m-hourforms bindvar post))
(316 #s(LN class m-dateforms bindvar post))
(317 #s(LN class m-locationforms bindvar post))
(318 #s(LN class m-subjectforms bindvar post))
(319 #s(LN class m-participantforms bindvar (participant post)))
(349 #s(LN class u-triplocforms bindvar (source dorig oloc pre)))
(350 #s(LN class flightwd bindvar head))
(351 #s(LN class tripwd bindvar head))
(352 #s(LN class poundsymb bindvar (flightmkr post)))
(353 #s(LN class number bindvar (flightnum dfltnum post)))
(354 #s(LN class m-originlocforms bindvar (oloc post)))
(355 #s(LN class m-destlocforms bindvar (destloc post)))
(356 #s(LN class m-dephrforms bindvar (departure post)))
(357 #s(LN class m-arrhrforms bindvar (arrival post)))
(358 #s(LN class m-depdtforms bindvar post))
(359 #s(LN class citypairforms bindvar (source pre)))

;;;             STPFORMS

(stpforms #s(CL isa domainroot domain c cept stp alevel 4 ig (ig-stp exp)))
(stp1 #s(ST isa stpforms
 strategy (406 407)
 mode (407)
 unodes ((406 407))
 snode (right 407)))
(astpforms #s(CL isa domainroot domain t cept stp alevel 4 ig (ig-stp exp)))
(astp1 #s(ST isa astpforms
 strategy (411 412)
 mode (412)
 unodes ((411 412))
 snode (right 412)))

(406 #s(LN class srcforms bindvar source))
(407 #s(LN class tgtforms bindvar target))
(411 #s(LN class asrcforms bindvar source))
(412 #s(LN class atgtforms bindvar target))

;;;             MARKED-OBJECTS1

(m-d-ggforms #s(CL isa domainroot domain c cept object alevel 4))
(dgg1 #s(ST isa m-d-ggforms strategy (401 403) mode (401 403) snode (all)))
(dgg2 #s(ST isa m-d-ggforms strategy (401 404) mode (404) snode (all)))
(m-i-ggforms #s(CL isa domainroot domain c cept object alevel 4))
(igg1 #s(ST isa m-i-ggforms strategy (402 403) mode (402 403) snode (all)))
(igg2 #s(ST isa m-i-ggforms strategy (402 404) mode (404) snode (all)))
(m-calsegforms #s(CL isa domainroot domain c cept object alevel 4))
*(mcalseg1 #s(ST isa m-calsegforms strategy (401 405) mode (405) snode (all)))
(m-d-tripforms #s(CL isa domainroot domain t cept object alevel 4))
```

```
(dtrip1
 #s(ST isa m-d-tripforms strategy (401 408) rnode (401 408) snode (all)))
(dtrip2 #s(ST isa m-d-tripforms strategy (409) rnode (409) snode (all)))
(m-i-tripforms #s(CL isa domainroot domain t cept object alevel 4))
(itrip1
 #s(ST isa m-i-tripforms strategy (402 408) rnode (402 408) snode (all)))
(itrip2 #s(ST isa m-i-tripforms strategy (409) rnode (409) snode (all)))
(m-airsegforms #s(CL isa domainroot domain t cept object alevel 4))
*(mairseg1 #s(ST isa m-airsegforms strategy (401 410) rnode (410) snode (all)))

(401 #s(LN class defmkr bindvar marker))
(402 #s(LN class indefmkr bindvar marker))
(403 #s(LN class mrggforms bindvar (object dggtype)))
(404 #s(LN class uggforms bindvar (object dggtype)))
(405 #s(LN class calsegforms bindvar (object dggtype)))
(408 #s(LN class mrtripforms bindvar (object dtriptype)))
(409 #s(LN class utripforms bindvar (object dtriptype)))
(410 #s(LN class airsegforms bindvar (object dtriptype)))

;;;              SLOTFORMS

(slotforms #s(CL isa domainroot domain c))
(shrslotforms #s(CL isa slotforms domain c cept slot alevel 5))
(shrslot1 #s(ST isa shrslotforms
  strategy (518 519 503 515) rnode (518 519 503 515) mnodes ((518 519))))
(shrslot2 #s(ST isa shrslotforms
  strategy (503 511 502 515) rnode (503 511 502 515)))
(ehrslotforms #s(CL isa slotforms domain c cept slot alevel 5))
(ehrslot1 #s(ST isa ehrslotforms
  strategy (518 519 504 515) rnode (518 519 504 515) mnodes ((518 519))))
(ehrslot2 #s(ST isa ehrslotforms
  strategy (504 511 502 515) rnode (504 511 502 515)))
(intslotforms #s(CL isa slotforms domain c cept slot alevel 5))
(intslot1 #s(ST isa intslotforms
  strategy (518 519 505 515) rnode (518 519 505 515) mnodes ((518 519))))
(intslot2 #s(ST isa intslotforms
  strategy (505 511 502 515) rnode (505 511 502 515)))
(locslotforms #s(CL isa slotforms domain c cept slot alevel 5))
(locslot1 #s(ST isa locslotforms
  strategy (518 519 506 515) rnode (518 519 506 515) mnodes ((518 519))))
(locslot2 #s(ST isa locslotforms
  strategy (506 511 502 515) rnode (506 511 502 515)))
(subjslotforms #s(CL isa slotforms domain c cept slot alevel 5))
(subjslot1 #s(ST isa subjslotforms
  strategy (518 519 507 515) rnode (518 519 507 515) mnodes ((518 519))))
(subjslot2 #s(ST isa subjslotforms
  strategy (507 511 502 515) rnode (507 511 502 515)))
(partslotforms #s(CL isa slotforms domain c cept slot alevel 5))
(partslot1 #s(ST isa partslotforms
  strategy (518 519 508 515) rnode (518 519 508 515) mnodes ((518 519))))
```

```
(partslot2 #s(ST isa partslotforms
  strategy (508 511 502 515) mode (508 511 502 515)))
(dateslotforms #s(CL isa slotforms domain c cept slot alevel 5))
(dateslot1 #s(ST isa dateslotforms
  strategy (518 519 509 515) mode (518 519 509 515) mnodes ((518 519))))
(dateslot2 #s(ST isa dateslotforms
  strategy (509 511 502 515) mode (509 511 502 515)))
(gentopslotforms #s(CL isa slotforms domain c cept slot alevel 5))
(gentopslot1 #s(ST isa gentopslotforms
  strategy (518 519 510 515) mode (518 519 510 515) mnodes ((518 519))))
(gentopslot2 #s(ST isa gentopslotforms
  strategy (510 511 502 515) mode (510 511 502 515)))

(aslotforms #s(CL isa domainroot domain t))
(ashrslotforms #s(CL isa aslotforms domain t cept slot alevel 5))
(shrslot01 #s(ST isa ashrslotforms
  strategy (520 521 503 516) mode (520 521 503 516) mnodes ((520 521))))
(shrslot02 #s(ST isa ashrslotforms
  strategy (503 511 512 516) mode (503 511 512 516)))
(aehrslotforms #s(CL isa aslotforms domain t cept slot alevel 5))
(ehrslot01 #s(ST isa aehrslotforms
  strategy (520 521 504 516) mode (520 521 504 516) mnodes ((520 521))))
(ehrslot02 #s(ST isa aehrslotforms
  strategy (504 511 512 516) mode (504 511 512 516)))
(aintslotforms #s(CL isa aslotforms domain t cept slot alevel 5))
(intslot01 #s(ST isa aintslotforms
  strategy (520 521 505 516) mode (520 521 505 516) mnodes ((520 521))))
(intslot02 #s(ST isa aintslotforms
  strategy (505 511 512 516) mode (505 511 512 516)))
(adateslotforms #s(CL isa aslotforms domain t cept slot alevel 5))
(dateslot01 #s(ST isa adateslotforms
  strategy (520 521 509 516) mode (520 521 509 516) mnodes ((520 521))))
(dateslot02 #s(ST isa adateslotforms
  strategy (509 511 512 516) mode (509 511 512 516)))
(aorigslotforms #s(CL isa aslotforms domain t cept slot alevel 5))
(origslot01 #s(ST isa aorigslotforms
  strategy (520 521 513 516) mode (520 521 513 516) mnodes ((520 521))))
(origslot02 #s(ST isa aorigslotforms
  strategy (513 511 512 516) mode (513 511 512 516)))
(adestslotforms #s(CL isa aslotforms domain t cept slot alevel 5))
(destslot01 #s(ST isa adestslotforms
  strategy (520 521 514 516) mode (520 521 514 516) mnodes ((520 521))))
(destslot02 #s(ST isa adestslotforms
  strategy (514 511 512 516) mode (514 511 512 516)))

(502 #s(LN class m-d-ggforms bindvar obj))
(503 #s(LN class startslotwd bindvar slottype))
(504 #s(LN class endslotwd bindvar slottype))
(505 #s(LN class timeslotwd bindvar slottype))
(506 #s(LN class locslotwd bindvar slottype))
```

```
(507 #s(LN class subjslotwd bindvar slottype))
(508 #s(LN class partslotwd bindvar slottype))
(509 #s(LN class dateslotwd bindvar slottype))
(510 #s(LN class gentopslotwd bindvar slottype))
(511 #s(LN class mdobjmkr bindvar marker))
(512 #s(LN class m-d-tripforms bindvar obj))
(513 #s(LN class originslotwd bindvar slottype))
(514 #s(LN class destslotwd bindvar slottype))
(515 #s(LN class stpforms bindvar stp))
(516 #s(LN class astpforms bindvar stp))
(518 #s(LN class mrggforms bindvar (obj dggtype)))
(519 #s(LN class uggforms bindvar (obj dggtype)))
(520 #s(LN class mrtripforms bindvar (obj dtriptype)))
(521 #s(LN class utripforms bindvar (obj dtriptype)))

;;;              MARKED-OBJECTS2

(m-d-slotforms #s(CL isa domainroot domain c cept object alevel 6))
(dslot1 #s(ST isa m-d-slotforms strategy (601 602) rnode (601 602)))
(m-d-aslotforms #s(CL isa domainroot domain t cept object alevel 6))
(daslot1 #s(ST isa m-d-aslotforms strategy (601 603) rnode (601 603)))

(601 #s(LN class defmkr bindvar marker))
(602 #s(LN class slotforms bindvar object))
(603 #s(LN class aslotforms bindvar object))

;;;              ACTIONFORMS

(actionforms
 #s(CL isa domainroot domain b cept action alevel 8 ig (ig-act res)))

(addforms #s(CL isa actionforms domain c cept add alevel 7))
(act1 #s(ST isa addforms
  strategy (701 702 703 704 708 799)
  rnode (704 708) unodes ((701 702 703)) mnodes ((701 702))))
(taddforms #s(CL isa actionforms domain t cept add alevel 7))
(act01 #s(ST isa taddforms
  strategy (713 702 714 704 718 799)
  rnode (704 718) unodes ((713 702 714)) mnodes ((713 702))))
(act02 #s(ST isa taddforms
  strategy (713 702 714 715 716 717 713 702 714 799)
  rnode (715 717) unodes ((716 717 713 702 714)) mnodes ((713 702))))

(deleteforms #s(CL isa actionforms domain c cept delete alevel 7))
(act2 #s(ST isa deleteforms
  strategy (701 702 703 705 709 799)
  rnode (705 709) unodes ((701 702 703))  mnodes ((701 702))))
(tdeleteforms #s(CL isa actionforms domain t cept delete alevel 7))
(act03 #s(ST isa tdeleteforms
  strategy (713 702 714 705 719 799)
```

```
   rnode (705 719) unodes ((713 702 714)) mnodes ((713 702))))

(changeforms #s(CL isa actionforms domain c cept change alevel 7))
(act3 #s(ST isa changeforms
  strategy (701 702 703 706 709 711 799)
  rnode (706 709 711) unodes ((701 702 703)) mnodes ((701 702))))
(act4 #s(ST isa changeforms
  strategy (701 702 703 706 710 799)
  rnode (706 710) unodes ((701 702 703)) mnodes ((701 702))))
(act5 #s(ST isa changeforms
  strategy (701 702 703 706 709 723 724 799)
  rnode (706 709 723 724) unodes ((701 702 703)) mnodes ((701 702))))
*(act6a #s(ST isa changeforms
  strategy (703 709 726 732 739 740 799)
  rnode (709 726 732)))
*(act6b #s(ST isa changeforms
  strategy (703 709 727 733 739 741 799)
  rnode (709 727 733)))
*(act6c #s(ST isa changeforms
  strategy (703 709 727 734 739 742 799)
  rnode (709 727 734)))
*(act6d #s(ST isa changeforms
  strategy (701 702 709 728 735 739 743 799)
  rnode (709 728 735) unodes ((701 702)) mnodes ((701 702))))
*(act6e #s(ST isa changeforms
  strategy (701 702 703 709 729 736 739 744 799)
  rnode (709 729 736) unodes ((701 702 703)) mnodes ((701 702))))
*(act6f #s(ST isa changeforms
  strategy (701 702 703 709 730 737 739 745 799)
  rnode (709 730 737) unodes ((701 702 703)) mnodes ((701 702))))
*(act6g #s(ST isa changeforms
  strategy (701 702 703 709 731 738 739 746 799)
  rnode (709 731 738) unodes ((701 702 703)) mnodes ((701 702))))
(tchangeforms #s(CL isa actionforms domain t cept change alevel 7))
(act04 #s(ST isa tchangeforms
  strategy (713 702 714 706 719 721 799)
  rnode (706 719 721) unodes ((713 702 714)) mnodes ((713 702))))
(act05 #s(ST isa tchangeforms
  strategy (713 702 714 706 720 799)
  rnode (706 720) unodes ((701 702 703)) mnodes ((701 702))))
(act06 #s(ST isa tchangeforms
  strategy (713 702 714 706 719 723 725 799)
  rnode (706 719 723 725) unodes ((713 702 714)) mnodes ((713 702))))

(showforms #s(CL isa actionforms domain c cept show alevel 7))
(act7 #s(ST isa showforms
  strategy (701 702 703 707 712 799)
  rnode (707 712) unodes ((701 702 703)) mnodes ((701 702))))
(tshowforms #s(CL isa actionforms domain t cept show alevel 7))
(act07 #s(ST isa tshowforms
```

```
  strategy (713 702 714 707 722 799)
  mode (707 722) unodes ((713 702 714)) mnodes ((713 702))))

(701 #s(LN class m-hourforms))
(702 #s(LN class m-intervalforms))
(703 #s(LN class m-dateforms))
(704 #s(LN class addwd))
(705 #s(LN class deletewd))
(706 #s(LN class changewd))
(707 #s(LN class showwd))
(708 #s(LN class m-i-ggforms bindvar obj))
(709 #s(LN class m-d-ggforms bindvar obj))
(710 #s(LN class m-d-slotforms bindvar obj))
(711 #s(LN class stpforms bindvar stp))
(712 #s(LN class m-calsegforms))
(713 #s(LN class m-dephrforms))
(714 #s(LN class m-depdtforms))
(715 #s(LN class gowd))
(716 #s(LN class m-originlocforms))
(717 #s(LN class m-destlocforms))
(718 #s(LN class m-i-tripforms bindvar obj))
(719 #s(LN class m-d-tripforms bindvar obj))
(720 #s(LN class m-d-aslotforms bindvar obj))
(721 #s(LN class astpforms bindvar stp))
(722 #s(LN class m-airsegforms))
(723 #s(LN class targetmkr bindvar tmarker))
(724 #s(LN class m-i-ggforms bindvar target))
(725 #s(LN class m-i-tripforms bindvar target))
(726 #s(LN class auxverb+ivlmkr bindvar tmarker))
(727 #s(LN class auxverb+hourmkr bindvar tmarker))
(728 #s(LN class auxverb+datemkr bindvar tmarker))
(729 #s(LN class auxverb+locmkr bindvar tmarker))
(730 #s(LN class auxverb+prtmkr bindvar tmarker))
(731 #s(LN class auxverb+sbjmkr bindvar tmarker))
(732 #s(LN class u-intervalforms bindvar (target dinterval)))
(733 #s(LN class u-hourforms bindvar (target dshour)))
(734 #s(LN class u-hourforms bindvar (target dehour)))
(735 #s(LN class u-dateforms bindvar (target ddate)))
(736 #s(LN class u-locationforms bindvar (target dloc)))
(737 #s(LN class u-participantforms bindvar (target dparts)))
(738 #s(LN class u-subjectforms bindvar (target dsubj)))
(739 #s(LN class sourcemkr2 bindvar smarker))
(740 #s(LN class u-intervalforms bindvar (source dinterval)))
(741 #s(LN class u-hourforms bindvar (source dshour)))
(742 #s(LN class u-hourforms bindvar (source dehour)))
(743 #s(LN class u-dateforms bindvar (source ddate)))
(744 #s(LN class u-locationforms bindvar (source dloc)))
(745 #s(LN class u-participantforms bindvar (source dparts)))
(746 #s(LN class u-subjectforms bindvar (source dsubj)))
(799 #s(LN class eosmkr))
```

A.3. The Concept Hierarchy

```
(b action #s(CD))

(b add
  #s(CD isa action
        seg ((pointers addwd gowd)
             (opponents deletewd showwd tmhrmkr tmivlmkr tmdtmkr calendarwd
                  fschedwd changewd locslotwd subjslotwd partslotwd dateslotwd
                  gentopslotwd originslotwd destslotwd sourcemkr targetmkr
                  mdobjmkr startslotwd endslotwd timeslotwd defmkr)
             (turnoff deleteforms tdeleteforms showforms m-calsegforms calsegforms
                  tm-dateforms tm-hourforms tm-intervalforms tshowforms
                  m-airsegforms airsegforms changeforms m-d-slotforms stpforms
                  slotforms srcforms tgtforms tchangeforms m-d-aslotforms astpforms
                  aslotforms asrcforms atgtforms m-d-ggforms m-d-tripforms))
        rec ( (no-del obj))))
(b delete
  #s(CD isa action
        seg ((pointers deletewd)
             (opponents addwd gowd showwd tmhrmkr tmivlmkr tmdtmkr calendarwd
                  fschedwd changewd locslotwd subjslotwd partslotwd dateslotwd
                  gentopslotwd originslotwd destslotwd sourcemkr targetmkr
                  mdobjmkr startslotwd endslotwd timeslotwd indefmkr)
             (turnoff addforms taddforms showforms m-calsegforms calsegforms
                  tm-dateforms tm-hourforms tm-intervalforms tshowforms
                  m-airsegforms airsegforms changeforms m-d-slotforms stpforms
                  slotforms srcforms tgtforms tchangeforms m-d-aslotforms
                  astpforms aslotforms asrcforms atgtforms m-i-ggforms))))
*(b change
  #s(CD isa action
        seg ((pointers changewd locslotwd partslotwd subjslotwd gentopslotwd
                  originslotwd destslotwd startslotwd endslotwd mdobjmkr
                  timeslotwd dateslotwd sourcemkr targetmkr sourcemkr2
                  auxverb+ivlmkr auxverb+hourmkr auxverb+datemkr
                  auxverb+locmkr auxverb+prtmkr auxverb+sjbmkr)
             (opponents addwd gowd deletewd showwd tmhrmkr tmivlmkr
                  tmdtmkr calendarwd fschedwd)
             (turnoff addforms taddforms deleteforms tdeleteforms showforms
                  tshowforms m-calsegforms calsegforms tm-dateforms
                  tm-hourforms tm-intervalforms m-airsegforms airsegforms))
        rec ( (no-del obj stp target) (no-tm (smarker source) (tmarker target)))))
(b show
  #s(CD isa action
        seg ((pointers showwd)
             (opponents addwd gowd deletewd changewd locslotwd subjslotwd
                  partslotwd dateslotwd gentopslotwd originslotwd destslotwd
                  sourcemkr targetmkr mdobjmkr startslotwd endslotwd timeslotwd
                  indefmkr)
```

```
            (turnoff addforms taddforms deleteforms tdeleteforms changeforms
                m-d-slotforms stpforms slotforms srcforms tgtforms tchangeforms
                m-d-aslotforms astpforms aslotforms asrcforms atgtforms
                m-i-ggforms m-d-ggforms m-i-tripforms m-d-tripforms))))

(b slot
  #s(CD exp (((value stp) (class slottype))
             (lambda (dstp dslot)
              (case dstp
               (dinterval (eq dslot 'timeslotwd))
               (dshour (eq dslot 'startslotwd))
               (dehour (eq dslot 'endslotwd))
               (ddate (eq dslot 'dateslotwd))
               (dloc (eq dslot 'locslotwd))
               (dparts (eq dslot 'partslotwd))
               (dsubj (eq dslot 'subjslotwd))
               (dgentop (eq dslot 'gentopslotwd))
               (dorig (eq dslot 'originslot))
               (ddest (eq dslot 'destslot)))))
        rec ( (no-del obj slottype stp) (no-trn (marker obj)))))

(b object
  #s(CD rec ( (no-del object)
             (no-trn (marker object) (pre head) (head post) (pre post)))))

(b stp
  #s(CD exp (((dbvar source) (dbvar target))
             (lambda (bvsrc bvtgt)
              (or (eq bvsrc bvtgt)
                  (and (eq 'dinterval bvsrc) (find bvtgt '(dehour dshour)))
                  (and (eq 'dinterval bvtgt) (find bvsrc '(dehour dshour))))))
        rec ( (no-del target))))

(b source #s(CD rec ((no-del source) (no-trn (smarker source)))))
(b target #s(CD rec ((no-del target) (no-trn (tmarker target)))))

(t nongg
  #s(CD isa object
        seg ((pointers flightwd poundsymb tripwd tbuildingname)
             (opponents meetingwd seminarwd classwd mealwd buildingname sbjmkr
                 tokenseq subjname projectname groupname projwd studentname
                 profname familyname prtmkr roomwd roomname locmkr showwd
                 gentopicname calendarwd fschedwd tmhrmkr tmivlmkr tmdtmkr)
             (turnoff mrggforms uggforms gentopicforms classnumforms
                 u-subjectforms m-subjectforms u-participantforms m-participantforms
                 u-locationforms m-locationforms showforms m-calsegforms
                 calsegforms tm-dateforms tm-intervalforms tm-hourforms tshowforms
                 m-airsegforms airsegforms))))
(t trip
  #s(CD isa object
```

```
        exp (((value departure) (value arrival))
             (lambda (ds de)
               (let ((shr (xhr (if (listp ds) (car ds) ds)))
                     (smin (xmin (if (listp ds) (car ds) ds)))
                     (ehr (xhr (if (listp de) (car de) de)))
                     (emin (xmin (if (listp de) (car de) de))))
                 (or (< shr ehr)
                     (and (= shr ehr) (< smin emin))
                     (and (> shr ehr) (or (listp ds) (listp de)))))))
        rec ( (no-trn (pre head) (head post) (pre post)))))

(t air #s(CD isa trip))

(t flight
  #s(CD isa trip
        exp (((ptype oloc)) (lambda (l) (eq l 'cityname))
             ((ptype destloc)) (lambda (l) (eq l 'cityname)))
        rec ( (no-del flightnum) (no-sub flightnum) (no-trn (flightmkr flightnum)))))

(c gg
  #s(CD isa object
        seg ((pointers meetingwd classwd seminarwd mealwd buildingname roomwd
                  roomname familyname studentname profname groupname
                  gentopicname projectname subjname projwd sbjmkr tokenseq locmkr)
             (opponents tripwd flightwd poundsymb dephrmkr arrhrmkr depdtmkr
                  originmkr destmkr cityname showwd calendarwd fschedwd
                  tmhrmkr tmivlmkr tmdtmkr)
             (turnoff mrtripforms tripforms utripforms m-dephrforms m-arrhrforms
                  m-depdtforms u-triplocforms m-originlocforms m-destlocforms
                  fl1forms fl2forms citypairforms showforms m-calsegforms
                  calsegforms tm-dateforms tm-intervalforms tm-hourforms tshowforms
                  m-airsegforms airsegforms))))

(c meeting
  #s(CD isa object
        def ((loc ( * office)))))
(c seminar
  #s(CD isa object
        exp (((ptype participant)) (lambda (p) (eq p 'u-personforms)))
         def ((loc ( * 5409)))))
(c class
  #s(CD isa object
        def ((loc ( * 5409)))))
(c meal
  #s(CD isa object
        def ((loc (* *)))))
(c breakfast
  #s(CD isa meal
        def ((starthr 800))))
(c lunch
```

```
  #s(CD isa meal
        def ((starthr 1200))))
(c dinner
  #s(CD isa meal
        def ((starthr 1900))))

(c calseg
  #s(CD isa object
        seg ((pointers tmhrmkr tmdtmkr tmivlmkr calendarwd)
             (opponents fschedwd meetingwd seminarwd classwd mealwd tokenseq
                  sbjmkr subjname projectname groupname projwd studentname
                  profname familyname prtmkr roomwd roomname locmkr
                  gentopicname cityname tripwd flightwd poundsymb dephrmkr
                  arrhrmkr depdtmkr originmkr destmkr addwd gowd deletewd
                  locslotwd subjslotwd partslotwd dateslotwd gentopslotwd
                  originslotwd destslotwd changewd sourcemkr targetmkr mdobjmkr
                  startslotwd endslotwd timeslotwd businessname schoolname
                  buildingname tbuildingname)
             (turnoff addforms taddforms deleteforms tdeleteforms changeforms
                  m-d-slotforms stpforms slotforms srcforms tgtforms tchangeforms
                  m-d-aslotforms astpforms aslotforms asrcforms atgtforms
                  m-i-ggforms m-i-tripforms m-d-ggforms m-d-tripforms mrggforms
                  uggforms mrtripforms tripforms utripforms gentopicforms
                  classnumforms u-subjectforms m-subjectforms u-participantforms
                  m-participantforms u-locationforms m-locationforms m-dephrforms
                  m-arrhrforms m-depdtforms u-triplocforms m-originlocforms
                  m-destlocforms meetingforms seminarforms classforms mealforms
                  fl1forms fl2forms m-airsegforms airsegforms citypairforms))))
(t airseg
  #s(CD isa object
        seg ((pointers fschedwd)
             (opponents calendarwd tmhrmkr tmivlmkr tmdtmkr meetingwd seminarwd
                  classwd mealwd sbjmkr tokenseq subjname projectname groupname
                  projwd studentname profname familyname prtmkr roomwd
                  roomname locmkr gentopicname cityname tripwd flightwd
                  poundsymb addwd gowd deletewd locslotwd subjslotwd partslotwd
                  dateslotwd gentopslotwd originslotwd destslotwd changewd
                  sourcemkr targetmkr mdobjmkr startslotwd endslotwd timeslotwd
                  businessname schoolname buildingname tbuildingname)
             (turnoff addforms taddforms deleteforms tdeleteforms changeforms
                  m-d-slotforms stpforms slotforms srcforms tgtforms tchangeforms
                  m-d-aslotforms astpforms aslotforms asrcforms atgtforms
                  m-i-ggforms m-i-tripforms m-d-ggforms m-d-tripforms mrggforms
                  uggforms mrtripforms tripforms utripforms gentopicforms
                  classnumforms u-subjectforms m-subjectforms u-participantforms
                  m-participantforms u-locationforms m-locationforms meetingforms
                  seminarforms classforms mealforms fl1forms fl2forms))))

(t flightnum
  #s(CD bind (find !value '(103 71 18 22 82 265 115 11 16 86 192
```

```
                    54 458 250 616))))

(b date
  #s(CD rec ( (no-del date day)
              (no-sub day year)
              (no-trn (marker date) (month day) (month year) (day year)))
        ext  (date ((userdate datename)))))
(b day #s(CD bind (< 0 !value 31)))
(b year #s(CD bind (< 1985 !value 1989)))

(b interval
  #s(CD exp (((value starthour) (value endhour))
              (lambda (ds de)
                (let ((shr (xhr (if (listp ds) (car ds) ds)))
                      (smin (xmin (if (listp ds) (car ds) ds)))
                      (ehr (xhr (if (listp de) (car de) de)))
                      (emin (xmin (if (listp de) (car de) de))))
                  (or (< shr ehr)
                      (and (= shr ehr) (< smin emin))
                      (and (> shr ehr) (or (listp ds) (listp de)))))))
        rec ( (no-del interval starthour endhour)
              (no-trn (marker interval) (starthour endhour)
                      (emarker starthour) (emarker endhour)))
        ext (time-interval ((userint intervalwd)))))
(b hour
  #s(CD rec ( (no-del hour hr minutes)
              (no-sub hr minutes)
              (no-trn (marker hour) (hr minutes) (hr dnbit)))
        ext (single-time ((usertime hourname)))))
(b hr #s(CD bind (<= 1 !value 12)))
(b minutes #s(CD bind (<= 0 !value 59)))

(c participants
  #s(CD rec ( (no-del participants) (no-trn (marker participants)))
        ext (participant ((student studentname) (professor profname)
                          (family-member familyname) (group groupname)
                          (business businessname) (project projectname)))))

(c person #s(CD))
(c group #s(CD rec ((no-del instance) (no-trn (instance postmarker)))))

(t citypair
  #s(CD rec ((no-trn (marker origin) (marker destination) (origin destination)))))

(t triploc
  #s(CD rec ( (no-del origin destination)
              (no-trn (marker origin) (marker destination)))
        ext (location ((city cityname) (business businessname)
                       (school schoolname) (building buildingname)))))
```

```
(c location
  #s(CD rec ( (no-del location majorloc minorloc)
              (no-sub roomnumber)
              (no-trn (marker location)))
        ext (location ((business businessname) (school schoolname)
                       (building buildingname) (room roomname)))))
(c roomnumber #s(CD bind (or (= !value 430) (> !value 4000))))

(c subject
  #s(CD rec ((no-del subject) (no-trn (marker subject)))
        ext (subject subjname)))

(c classnumber #s(CD rec ((no-del id) (no-trn (level id)))))
(c level #s(CD bind (<= 15 !value 18)))
(c id #s(CD bind (<= 700 !value 800)))

(c gentopic #s(CD ext (general-topic gentopicname)))
```

Appendix B
Test Set Sentences

This appendix contains the test set of sentences considered during evaluation of CHAMP. The utterances were produced spontaneously in response to pictorial stimuli by users in either the hidden-operator or on-line experiments discussed in Chapters 3 and 9, respectively. Users 1, 2, 3, and 4 participated in the Adapt condition of the hidden-operator experiments. Users 5 and 7 participated in the Adapt/Echo condition (input from Users 6 and 8 is not included in the test set because they participated in the No-Adapt condition of the hidden-operator experiments). Users 9 and 10 saw the same stimuli as the others (with dates adjusted for events to occur in 1989) but interacted directly with CHAMP.

The sentences are presented as they appear in the user's log file except where they have been marked to indicate the changes that were made to accommodate evaluation (see Section 9.1.1 for a full discussion). The references to "Rules," below, relate the added markings to the rules for compensating for differences between the simulation and the implementation that were given in Figure 9-1.

Marking	Explanation
<text>	angle brackets surround text to compensate for instances of the lexical extension problem (Rule 1). They force CHAMP to treat the tokens as unknown.
>text<	inverted angle brackets surround text that was inserted during testing to force CHAMP to accept. The added information generally keeps the system from finding a resolution-time error not found during the simulation (Rule 5). In a few cases, text was added to avoid having to make radical changes to the LISP reader (for example, "June20", a legitimate symbol name in LISP, is changed to "June> <20").
[text]	square brackets surround text that was removed before testing. The deleted text represents redundancy that could be ignored via constructive inference during the simulation (Rule 4).
[text]->text<	square brackets and inverted angle brackets joined by a hyphen indicate that the square bracketed text was replaced with the angle bracketed text prior to testing. The replacement may compensate for unimplemented semantics (Rule 3) or a resolution-time error that was undetected during the simulation (Rule 5).

|| double vertical bars at the end of a sentence indicate that the following sentence was joined to the current one by conjunction. The sentences are separated via Rule 2 with information distributed across sentences enclosed in inverted angle brackets.

B.1. Utterances from User 1

Session 1

1. change >June 10< speech project meeting from 10 to 11
2. change >June 10< speech project meeting from 10-11 to 10-11:30
3. cancel John's speech research meeting on June 9
4. schedule AISyS meeting at 7 P.M. on June 11
5. schedule lunch with Andy from Noon until 1:30 P.M. on June 12
6. change June 12 meeting at AISys to 2 P.M.
7. cancel June 14th AISys meeting
8. change June 15 flight from NY to Pittsburgh [to an earlier time]->to a flight from NY to pittsburgh at 4 p.m.<
9. move Anderson seminar on June 10 to room 7220

Session 2

1. change CogSci seminar on June 10 from Anderson to VanLehn
2. schedule Jill Larkin in Porter Hall Room 430 on June 10 from 3 to 4 P.M.
3. cancel 1 P.M. class on June 16th
4. cancel the 1:30 class <the same day>
5. schedule <trip> from Pittsburgh to Chicago on June 12th at 9 P.M.
6. cancel reservations <on> Flight <No.> 54 on June 13
7. show me the June 13th schedule
8. schedule a flight from Chicago to NY on June 13th at 11 P.M.
9. move June 13th lunch from Station Square to <VC. Incorporated>
10. On June 11 schedle the LISP tutorial class from 10 to 11, the PRODIGY meeting with Craig from 2 to 3, a meeting with Jill in <the User Studies Lab> at 4, and dinner with Allen at 6
11. show me the schedule for June 11th
12. schedule a meeting about PRODIGY with Craig from 2 to 3 on June 11
13. schedule a meeting in the User Studies Lab with Jill from 3 to 4 on June 11
14. schedule dinner with Allen at 6 P.M. on June 11
15. schedule a Natural Language Interfaces Seminar at 9:30 A.M. in Room 5409 on June 12 [; the seminar is] about Story Understanding <and> Generation

Session 3

1. cancel June 12th meeting at AISyS
2. Note that June 11th LISP tutorial is in Room 8220

3. The June 11th LISP tutorial is in Room 8220
4. change meeting time from 3-4 to 3-4:30 on June 11
5. schedule a meeting with John about Speech Research on June 12 from 2 to 3
6. schedule a taxi to the airport on June 12 at 8:30 P.M.
7. cancel June 11 meeting at 3 P.M.
8. move PRODIGY meeting on June 11 one hour later
9. move PRODIGY meeting on June 11 to 3-4
10. schedule department meeting from 2 to 3 on June 11
11. cancel dinner on June 11
12. schedule a meeting with Jill in the User Studies Lab on June 12th from 3 to 4:30
13. schedule a meeting with Roger from <the University of Chicago> on June 13th from 10 to Noon
14. Schedule a meeting on June 13th from 10 to noon [with University of Chicago's Roger]->at the University of Chicago with Roger<
15. schedule meeting on June 14th from 10 to 1 with Mike at Columbia
16. cancel June 12th meeting at AISyS

Session 4

1. display the schedule for June 12
2. schedule a meeting with John on June 12 from 8:30 to 9:30
3. change 2 to 3 meeting with John about speech research on June 12 to a meeting with Jaime
4. show me the <flight schedule> for June 13th
5. reschedule flight 103 on June 13th for 10 P.M.
6. cancel flight 103 on June 13th ||
7. schedule a new flight at 10 p.m. the same day
8. show me the airline schedule from Chicago to New York on June 13th
9. cancel reservations on flight 103 ||
10. schedule reservations on flight 71 on june 13
11. move lunch with Andy on June 12th from Noon to 12:30 P.M.
12. move lunch with VC on June 13th from 1 P.M. to 4 P.M.
13. show me the schedule for [Friday,] June 13th
14. schedule dinner with Roger from 6 P.M. to 8 P.M. on June 13th
15. schedule meeting at AISyS about finances on June 16 from 10 to 11:30
16. schedule meeting with Craig on June 16th from Noon to 1 P.M.
17. show me the June 16th schedule
18. move 3 P.M. meeting on June 12th to June 16th
19. show me the June 12th schedule
20. schedule dinner with Marsha from 6 P.M. to 8:30 P.M. on June 12th

Session 5

1. show June 14 schedule
2. show flight schedule from MY to Pittsburgh on June 14
3. cancel flight 115 on june 15 ||
4. schedule flight 115 on june 14
5. move june 14 meeting with mike from 1 to 11:30
6. schedule lunch meeting with Bob on June 14 at Noon
7. schedule June 15 meeting with Jaime from 11 to 12
8. cancel 10 A.M. meeting on June 13
9. schedule LISP tutorial class from 9 to 10 on June 16
10. move 10 a.m. meeting on June 16 from AISyS to CMU
11. schedule 10-11 a.m. meeting with andy on june 17 in room 7220 ||
12. schedule 11-12 a.m. speech project meeting on june 17 in room 7220
13. change June 17 Newell seminar to Anderson

Session 6

1. move June 16 LISP class to June 17
2. cancel 10 a.m. appointment on June 17
3. move [lunch] June 16 lunch [ahead one half hour]->to 12:30-1:30<
4. schedule AI seminar on June 16 from 2 to 3 with Drew McDermott speaking about "Non-Monotonic Logics"
5. move June 16 meeting from User Studies Lab to office
6. move 11 a.m. June [16]->17< meeting to room 8220
7. show me the flight schedule from Pittsburgh to NY on June 18
8. schedule flight 250
9. show return schedule the same day
10. show <schedule> from NY to Pittsburgh the same day
11. schedule flight 115

Session 7

1. change June 18 to lunch with Richard
2. move meeting with Mike on June 18 [back to hours]->to 9-10<
3. show Pittsburgh to NY flight schedule for June 18
4. reschedule <airline reservations> on June 18 to flight 616
5. schedule 11 A.M. meeting on June 19 at AISyS
6. schedule noon meeting about finances on June 19
7. note that speaker at 9:30 June 19 meeting is Jaime
8. schedule Jaime as the speaker at the Natural Language Interfaces Seminar on June 19 at 9:30
9. cancel 10 A.M. June 20th meeting
10. show flight schedule from Pgh to NY on June 20
11. show flights from Pgh to Chicago on the same day

12. schedule flight 86
13. show [return flights]->flights from chicago to pgh< on June 21
14. schedule flight 11
15. move class 15-731 to room 7220 on June 23

Session 8

1. change the 9:30 a.m. meeting on june 19 from jaime to jill
2. return to cmu from aisys at 2:30 p.m. on june 20
3. schedule lisp tutorial class from 3-4 on june 20
4. show schedule from Chicago to Pgh on June 21
5. reschedule to flight 16
6. schedule PRODIGY meeting with Craig from 10-11 on June 23
7. schedule lunch with Marsha at Noon on the 23rd
8. on the 24th meet with Andy from 10 to 11 ||
9. >on the 24th< meet at Station Square with VC about finances from 2 to 3
10. on the 26th cancel 9 :30 a.m. meeting ||
11. >on the 26th< schedule meeting from 4 to 5:30 with Mitchell about "explanation-based reasoning" for AI seminar
12. on the 26th cancel 9:30 a.m. meeting ||
13. >on the 26th< schedule ai seminar from 4 to 5:30 with Mitchell speaking about "explanation-based reasoning"

Session 9

1. change June 19 Natural Language Interfaces Seminar to AI Seminar
2. cancel 3 P.M. class on June 20
3. move 1:30 P.M. class on June 23 to room 5409
4. note that June 24 meeting at 10 A.M. is in Room 7220
5. show June 25 schedule
6. schedule lunch with Marsha at Noon
7. show flights from Pgh to NY [after 6 p.m.]->from 6 p.m. to 11:59 p.m.< on June 27
8. schedule flight 458
9. schedule meeting with Mike from 10 to 12 on June 28
10. show flights from NY to Pgh [after noon]->from noon to 11:59 p.m.< on June 28
11. schedule flight 115
12. schedule dinner with Allen from 6:30 to 7:30 P.M. the same day
13. schedule meeting with Jill Larkin from 10-11 on June 26 in the User Studies Lab
14. change speaker <at> June 26 AI seminar to Craig ||
15. >change< subject of same seminar to PRODIGY

B.2. Utterances from User 2

Session 1

1. change meetings end from 11:00 a.m. to 11:30 a.m. on June 9, 1986
2. cancel Speech Research meeting with John on June 9, 1986 at 9:00 a.m.
3. schedule AISys meeting at 7:00 p.m. on June 11, 1986
4. schedule lunch with Andy on June 12, 1986 from 12:00 [noon] till 1:00 p.m.
5. schedule lunch with Andy on June 12, 1986 from 12:00 [noon] till 1:30 p.m.
6. cancel 12:00 meeting at AISys reschedule it for 2:00 June 12, 1986
7. cancel 12:00 meeting on June 12, 1986
8. schedule 2:00 meeting at AISys on June 12, 1986
9. cancel AISys Meeting on June 14, 1986 at 10:00 a.m.
10. cancel flight #82 from New York leaving at 8:00 p.m. on June 15, 1986 arriving in Pittsburgh 9:00 p.m.
11. I need to know the flight schedule for airlines leaving New York and arriving in Pittsburgh on [Sunday] June 15, 1986 before 5:00 p.m.
12. <book 1 person for> Flight #115 leaving New York and arriving in Pittsburgh at 5:10
13. schedule dinner with his father at <the> Greentree Marriot at 6:30 p.m.
14. change the location of the meeting on June 10, 1986 at 12:00 [noon] ending at 1:30 from room 5409 to 7220
15. need to meet with Allen on June 12, 1986 between 9:00 a.m. and 12:00 [noon] is there anything scheduled?
16. schedule meeting with Allen on June 12, 1986 <beginning at> 10:00 and <ending at> 11:00 a.m.
17. schedule meeting with Allen on June 12, 1986 <beginning at> 11:00 a.m. and <ending at> 12:00 [noon]

Session 2

1. seminar on June 10, 1986 was to be lead by Mr. Anderson, he will no longer be there. It will be headed by VanLehn instead
2. CogSci Seminar on June 10, 1986 will be with VanLehn not Anderson
3. schedule meeting with Jill Larkin on June 10, 1986 at Porter Hall room 430 from 3:00 p.m. to 4:00 p.m.
4. cancel class 15-731 on [Monday,] June 16, 1986 from 1:30 to 3:00
5. <I want to see flight schedule> for June 12, 1986 from Pittsburgh to Chicago [around 9:00 p.m.]->at 9:00 p.m.<
6. schedule flight #192 for 8:10 p.m. <departure from> Pittsburgh arriving in Chicago at 8:34 p.m. on June 12, 1986
7. cancel flight #54 leaving Pittsburgh 7:00 p.m. arriving in New York at 8:00 p.m. on June 13, 1986 ||
8. show me the <flight schedule> for [that evening]->June 13, 1986<
9. is this the only flight that evening

10. are there any other flights
11. schedule flight #71 leaving Chicago at 8:05 p.m. and arriving in New York at 11:05 on June 13, 1986
12. change the location of the lunch with VC <, a company> on June 13, 1986 beginning at 12:00 [noon] ending at 1:00 p.m. to [location] VC Incorporated
13. schedule Lisp Tutorial Class <for> June 11, 1986 beginning at 10:00 a.m. and ending at 11:00 a.m.
14. schedule meeting with Craig <,> Prodigy on June 11, 1986 beginning at 2:00 p.m. and ending at 3:00 p.m.
15. schedule meeting with Jill in <the Lab about User Studies> on June 11, 1986 beginning at 3:00 p.m. and ending at 4:00 p.m.
16. schedule dinner with Allen <who is a professor> at 6:00 p.m. on June 11, 1986
17. the subject of the Natural Language Interfaces Seminar on June 12, 1986 will be Story Understanding <and> Generation

Session 3

1. cancel trip to AISyS at 2:00 p.m. on June 12, 1986
2. schedule LISP Tutorial Class in Room 8220 beginning at 10:00 a.m. and ending at 11:00 a.m.
3. change ending time <for> meeting with Jill in <the User Studies Lab> on June 11, 1986 from 4:00 p.m. to 4:30 p.m.
4. schedule meeting with John about Speech Research on June 12, 1986 beginning at 2:00 p.m. and ending at 3:00 p.m.
5. schedule transit <time> from CMU to airport at 8:30 p.m.
6. show me the calendar for June 12, 1986
7. cancel meeting with Jill in the User Studies Lab beginning at 3:00 p.m. and ending at 4:30
8. change the time of the meeting with Craig on June 11, 1986 from 2:00 p.m. ending at 3:00 p.m. to 3:00 p.m. ending at 4:00 p.m.
9. schedule Department meeting for June 11, 1986 beginning at 2:00 p.m. and ending at 3:00 p.m.
10. cancel dinner with Allen on June 11, 1986 at 6:00
11. schedule meeting with Jill in the User Studies Lab on June 12, 1986 beginning at 3:00 p.m. and ending at 4:30 p.m.
12. schedule meeting with Roger at <the University of Chicago> on June 13, 1986 beginning at 10:00 ending at 12:00
13. schedule meeting with Mike at Columbia beginning at 10:00 a.m. and ending at 1:00 p.m.

Session 4

1. show me the calendar for June 12, 1986
2. schedule meeting with John on June 12, 1986 8:00 to 9:00 p.m.
3. cancel meeting witlh John on June 12, 1986
4. schedule meeting with John on June 12, 1986 8:00 a.m. to 9:00 a.m.

5. schedule lunch with Andy 12:00 to 1:30 p.m. on June 12, 1986
6. cancel meeting with John about Speech Research on June 12, 1986 ||
7. schedule meeting this <time> with Jaime
8. show me the flight schedule <for flights> leaving Chicago [before 11:00 p.m.]->from 6 a.m. to 11 p.m.< on June 13, 1986
9. show me the calendar for June 13, 1986
10. schedule flight #71 leaving Chicago at 8:05 and arriving in New York at 11:05 on June 13, 1986
11. lunch with Andy on June 12, 1986 12:30 to 1:30 instead of 12:00 to 1:00
12. schedule lunch with Andy on June 12, 1986 for 12:30 to 1:30 instead of 12:00 to 1:30
13. <schedule> ending time of lunch with VC on Jue 13, 1986 from 1:00 p.m. to 4:00 p.m.
14. schedule dinner with Roger on June 13, 1986 from 6:00 p.m. to 7:00 p.m.
15. schedule AISys meeting (Financial) on June 16, 1986 from 10:00 a.m. to 11:30 a.m.
16. schedule AISyS meeting <to> discuss finance on June 16, 1986 from 10:00 a.m. to 11:30 a.m.
17. schedule meeting with Craig on June 16, 1986 from 12:00 to 1:00 p.m.
18. change the meeting with Jill from June 12, 1986 to [Monday] June 16, 1986
19. show me the calendar for June 12, 1986
20. schedule dinner with Marsha on June 12, 1986 from 6:00 p.m. to 7:30 p.m.

Session 5

1. cancel flight plans for [Sunday] June 15, 1986
2. show me the flight schedule for June 14, 1986
3. show me the schedule [before 4:00 p.m.]->from 6 a.m. to 4 p.m.<
4. show me the flight schedule for flights leaving New York arriving in Pittsburgh
5. show me the flight schedule for flights leaving New York arriving in Pittsburgh [before 4:00 p.m.]->from 6 a.m. to 4 p.m.< on [Saturday] June 14, 1986
6. schedule flight #115 on June 14, 1986
7. change the ending time of the meeting on June 14, 1986 with Mike from 1:00 p.m. to 11:30 a.m.
8. schedule lunch with Bob on June 14, 1986 at 12:00
9. schedule meeting with Jaime on June 15, 1986 from 11:00 a.m. to 12:00
10. show me the calendar for June 15, 1986
11. cancel meeting with Roger on June 13, 1986
12. schedule LISP Tutorial Class for [Monday,] June 16, 1986 from 9:00 a.m. to 10:00a.m.
13. change location of Financial Meeting on June 16, 1986 from AISyS to CMU
14. show me the calendar for June 16, 1986
15. schedule meeting with Andy in Room 7220 on June 17, 1986 from 10:00 a.m. to 11:00 a.m.

16. schedule Speech project in Room 7220 on June 17, 1986 from 11:00 a.m. to 12:00
17. change Newell to Anderson for the CogSci Seminar on June 17, 1986

Session 6

1. move the June 16, 1986 LISP Tutorial Class to June 17, 1986
2. change lunch with Craig on June 16, 1986 to 12:30 p.m. to 1:30 p.m.
3. cancel meeting with Andy in Room 7220 on June 17, 1986
4. schedule AI Seminar about "Non-Monotonic Logics" with Drew McDermott on June 16, 1986 from 2:00 p.m. to 3:00 p.m.
5. change location of meeting with Jill on June 16, 1986 to office
6. change location of meeting [about]->with< Speech Project on June 17, 1986 to Room 8220
7. show me the calendar for June 25, 1986
8. show me the flight schedule [for after 7:30 a.m.]->from 7:30 a.m. to 11:59 p.m.< on June 25, 1986
9. show me the flight schedule [for after 7:00 a.m.]->from 7:00 a.m. to 11:59 p.m.< on June 25, 1986
10. schedule flight #616 on June 25, 1986
11. schedule lunch with Mike at Columbia from 11:00 a.m. to 12:30 p.m. on June 25, 1986
12. cancel lunch with Mike on June 25, 1986
13. schedule meeting with Mike at Columbia at 11:00 a.m. on June 25, 1986
14. schedule lunch with Bob on June 25, 1986 at 12:30 p.m.
15. show me the flight schedule for June 25, 1986 New York to Pittsburgh
16. schedule flight #265 on June 25, 1986

Session 7

1. change location of June 17, 1986 meeting from 8220 to 7220
2. need earlier flight on June 18, 1986 from Pittsburgh to New York
3. show me the flight schedule for June 18, 1986 from Pittsburgh to New York
4. schedule flight #616 instead of #250 on June 18, 1986 leaving Pittsburgh and arriving in New York
5. change <meeting time> with Mike on June 18 to 9:00 a.m. to 10:00 a.m.
6. change lunch with Bob on June 18 to lunch with Richard
7. leave CMU at 11:00 a.m. on June 19 and go to AISys
8. schedule Financial meeting <for> 12:00 <,> June 19
9. show me the calendar for June 19
10. Seminar at 9:30 will be lead by Jaime
11. cancel AI Seminar June 20
12. show me the flight schedule for June 20 Pittsburgh to Chicago [after 8:30 p.m.]->from 8:30 p.m. to 11:59 p.m.<
13. schedule flight #86 on June 20

14. show me the flight schedule for June 21 Chicago to Pittsburgh [after 7:00 p.m.]->from 7:00 p.m.to 11:59 p.m.<
15. schedule flight #16 on June 21
16. location of Class 15-731 on June 23 will be Room 7220

Session 8

1. seminar on June 19 will be with Jill not Jaime
2. go from AISys to CMU at 2:30 p.m. on June 20
3. schedule LISP tutorial class for June 20 from 3:00 p.m . to 4:00 p.m.
4. schedule Prodigy with Craig on June 23 from 10:00 a.m. to 11:00 a.m.
5. schedule lunch with Marsha on June 23 at 12:00
6. schedule meeting with Andy, June 24 from 10:00 a.m. to 11:00 a.m.
7. schedule meeting with VC [(finances)]->about finances< at Station Square on June 24 from 2:00 p.m. to 3:00 p.m.
8. cancel seminar on June 26 from 9:30 to 10:30
9. show me the flight schedule for June 21 Chicago to Pittsburgh
10. schedule flight #16 on June 21 ||
11. cancel flight #11
12. schedule AI Seminar with Mitchell about Explanation <-> Based Reasoning on June 26 from 4:00 p.m. to 5:30 p.m.

Session 9

1. change Natural Language Interfaces Seminar to AI Seminar on June 19
2. cancel LISP tutorial class on June 20
3. change Class 15-731 location to Room 5409 on June 23
4. show me the schedule for June 25
5. schedule lunch with Marsha from 12:00 to 1:00 p.m. on June 25
6. meeting with Andy on June 24 <will be in> Room 7220
7. change location of class 15-731 to 5409
8. show me the flight schedule for June 27 Pittsburgh to New York
9. schedule flight #458 for June 27
10. schedule meeting with Mike on June 28 10:00 a.m. to 12:00
11. show me the flight schedule for June 28 New York to Pittsburgh
12. schedule flight #115 for June 28
13. schedule dinner with Allen at 6:30 on June 28
14. schedule User Studies Lab with Jill Larkin 10:00 a.m. to 11:00 a.m. on June 26
15. change speaker of AI Seminar to Craig on june 26 ||
16. change subject of ai seminar to Prodigy on June 26

B.3. Utterances from User 3

Session 1

1. Speech Project Meeting scheduled for June 9, 1986 will be from 10:00 to 11:30
2. <There has been a slight time change for> the Speech Project Meeting on June [9]->10<, 1986 <. It will be held from> 10:00 to 11:30.
3. Cancellation of John's speech on June 9, 1986
4. Cancel John's Speech Research Meeting at 9:00 on June 9, 1986.
5. ALSyS Meeting to be held June 11, 1986 at 7:00 p.m.
6. Lunch with Andy <,> June 12, 1986 from 12:00 p.m. to 1:30 p.m.
7. CMU AISys being held June 12, 1986 at 2:00 p.m.
8. Please change departure time from 12:00 to 2:00 for trip from CMU to AISyS
9. cancel AISyS 10:00 Meeting on June 14, 1986
10. Dinner with Dad [in pittsburgh] June 14, 1986 at 6:00.
11. I would like to request flight information to Pittsburgh from New York
12. Please schedule Flight #115 to Pittsburgh
13. CogSci Seminar will be held on [Tuesday,] June 10, 1986 from 12:30 to 1:30 in room 7220;Speaker will be Anderson.
14. Please keep calendar free for [Thursday,] June 12, 1986 from 9:00 a.m. to 12:00 p.m.;1 hour is needed with Professor Allen.
15. <one hour is needed> between 9:00 a.m. and 10:00 a.m. on [Thursday,] June 12, 1986 <to schedule> an appointment with Professor Allen.

Session 2

1. Please change speaker <for> CogSci Seminar on June [9]->10<, 1986 to Van Lehn.
2. Schedule meeting with Jill Larkin on June 10, 1986 at Porter Hall <,> Room 430 from 3:00 to 4:00.
3. Please cancel Class 15-731 on June 16, 1986 from 1:30 to 3:00.
4. What events are scheduled for June 12, 1986?
5. Flight schedule information from Pittsburgh to Chicago on June 12, 1986.
6. Schedule Flight #86 leaving Pittsburgh at 9:30pm
7. cancel flight #54 departing Pittsburgh on June 13, 1986.
8. Please state flight schedule from Chicago to New York on June 13, 1986.
9. Schedule Flight #103 departing Chicago at 11:00p.m.
10. Lunch with VC on June 13, 1986 has been changed from Station Square to <VC Inc>.
11. Please show me the calendar for June 11, 1986.
12. Schedule Lisp Tutorial Class from 10:00 to 11:00 on June 11, 1986.
13. Schedule Craig <,> Prodigy from 2:00 to 3:00 on June 11, 1986.
14. Schedule <the> User Studies Lab for Jill at 4:00 on June 11, 1986.
15. Dinner with Allen at 6:00pm on June 11, 1986.

16. Subject for Natural Language Interfaces Seminar on June 12, 1986 should be Story Understanding and Generation.

Session 3

1. <May I please see> calendar for June 12, 1986.
2. Please cancel trip to AISYS at 2:00 on June 12, 1986.
3. Lisp Tutorial class <will be held in> Room 8220 on June 11, 1986.
4. Please reschedule time for Jill in the User Studies Lab to be from 3:00 to 4:30.
5. Schedule time for John to do speech research from 2:00 to 3:00 on June 12, 1986
6. Schedule trip from CMU to Airport [by 8:30]->at 8:30<.
7. Please schedule a meeting with John <in reference to> Speech Research.
8. Cancel Time scheduled for Jill in The Users Studies Lab.
9. Cancel Jills meeting in the Users Studies Lab on June 11, 1986 at 3:00.
10. Please change time for Craigs Research project (Prodigy) to 3:00 on June 11th.
11. Reschedule Craigs meeting on June 11, 1986 <to be from> 3:00 to 4:00.
12. Scheduel a Department meeting from 2:00 to 3:00 on June 11, 1986.
13. cancel dinner with Allen on June 11, 1986.
14. Schedule <time for> Jill in the User Studies Lab from 3:00 to 4:30 on June 12th.
15. Schedule a meeting with Roger at <the University of Chicago> from 10:00 to 12:00 on June 13, 1986.
16. Schedule a meeting with Mike at Columbia on June 14, 1986 from 10:00 to 1:00.

Session 4

1. Can I see the calendar for June 12, 1986
2. Schedule a meeting with John from 5:00 to 6:00 on June 12, 1986.
3. Cancel meeting with John <,> Speech Research <,> on June 12th at 2:00.
4. reschedule a meeting with jaime at the same time
5. reschedule a meeting with Jaime at 2:00 on June 12th.
6. A early flight from chicago to new york is needed.
7. Please show me <flight> information for June 13th from Chicago to New York.
8. Cancel flight #103. ||
9. schedule Flight #71.
10. Change Andy's lunch on June 12th <to begin at> 12:30.
11. Change lunch with VC on June 13th to be from 12:00 to 4:00.
12. What is <on> the schedule for June 13th [after 12:00]->from 12:00 to 11:59<.
13. Schedule dinner with Roger at 6:00 on June 13th.
14. Schedule AISyS meeting on June 16th from 10:00 to 11:30.
15. Schedule meeting with Craig on June 16th from 12:00 to 1:00.
16. Reschedule Jills >June 12< meeting to June 16th from 3:00 to 4:30 in the Users Studies Lab.
17. What is on the calendar for June 12th [after 12:00]->from 12:00 to 11:59 p.m.<

18. schedule dinner with Marsha at 6:00 on June 12th.

Session 5

1. Cancel Flight #115 from New York to Pittsburgh on June 15th.
2. Show calendar for June 14, 1986.
3. Please list flight information for June 14, 1986 from New York to Pittsburgh.
4. Schedule flight #265 <departing from> New York at 6:10pm on June 14th.
5. Reschedule meeting with Mike at Columbia on June 14th to be from 10:00 to 11:30.
6. Schedule lunch with Bob on June 14th at 12:00.
7. Schedule <time> with Jaime on June 15th from 11:00 to 12:00.
8. Cancel meeting with Roger at the University of Chicago on June 13th.
9. Schedule Lisp Tutorial Class on June 16th from 9:00 to 10:00.
10. The Financial meeting on June 16th at AISyS has been changed to CMU.
11. Schedule meeting with Andy on June 17th in Room 7220 from 10:00 to 11:00.
12. Schedule room 7220 on June 17th from 11:00 to 12:00 <for> Speech Project.
13. Change speaker for CogSci Seminar on June 17th from Newell to Anderson

Session 6

1. Please reschedule June 16th LISP Tutorial Class to June 17th from 9:00 to 10:00.
2. Reschedule lunch with Craig on June 16th to be from 12:30 to 1:30.
3. Cancel meeting with Andy in Room 7220 on June 17th.
4. Schedule Al's Seminar entitled "Non-Monotonic Logics" with <speaker> Drew McDermott on June 16th from 2:00 to 3:00.
5. Reschedule Jill to be in the Office rather than the User Studies Lab on June 16th.
6. Reschedule Jills meeting in the Office rather than the User Studies Lab on June 16th from 3:00 to 4:00.
7. Speech Project will be held in Room 8220 on June 17th from 11:00 to 12:00.
8. Please list flight information from Pittsburgh to New York for June 25th.
9. What is already scheduled <on> the calendar for June 25th.
10. Schedule Flight #250 departing Pittsburgh at 8:00a.m.
11. Schedule a meeting with Mike at Columbia on June 25th at 11:00.
12. Schedule lunch with Bob at 12:30 on June 25th.
13. Please list Flight information from New York to Pittsburgh for June 25th.
14. Schedule Flight #115 departing New York at 4:00p.m.

Session 7

1. Speech Project on June 17th will be held in room 7220 instead <of> 8220.
2. <Are there any flights> leaving Pittsburgh [before 8:00]->from 6:00 a.m. to 8:00 p.m.< to New York on June 18th.
3. Reschedule Flight #616 departing Pittsburgh at 7:05 on June 18th.
4. Cancel Flight #250.

5. Schedule Flight #616 on June 18th.
6. Reschedule meeting with Mike on June 18th to be from 9:00 to 10:00.
7. Change lunch <to be with> Richard instead of Bob on June 18th.
8. Schedule a meeting at AISyS on June 19th at 11:00.
9. Schedule Financial meeting on June 19th at 12:00
10. Please list Jaime as the Speaker for the Natural Language Interfaces Seminar on June 19th.
11. Cancel AI Seminar on June 20th.
12. List Flight information <for> Pittsburgh to Chicago on June 20th.
13. schedule Flight #86 on June 20th.
14. List Flight information for Chicago to Pittsburgh on June 21st.
15. Schedule Flight #11 on June 21st.
16. Class 15-731 will be held in Room 7220 on June 23rd.

Session 8

1. Change Jaime to Jill for the Natural Language Interfaces Seminar on Juen 19th.
2. Schedule transportation time from AISys to CMU. Must arrive at CMU by 2:30.
3. Depart AISyS at 2:00 to be at CMU by 2:30.
4. Schedule LISP Tutorial class on June 20th from 3:00 to 4:00.
5. Please list flight information from Chicago to Pittsburgh on June 21st.
6. Cancel flight #11 for June 21st. ||
7. Reschedule Flight #16 for June 21st.
8. Schedule Craig on June 23rd from 10:00 to 11:00 <to work on his research project entitled> "Prodigy>"<.
9. Schedule lunch with Marsha at 12:00 on June 23, 1986.
10. Schedule meeting with Andy on June 24th from 10:00 to 11:00.
11. Schedule a meeting at Station Square with VC (Finances) on June 24th from 2:00 to 3:00.
12. Cancel Natural Language Interfaces Seminar on June 26th.
13. Schedule AI Seminar on June 26th <at> 4:00 to 5:30. ||
14. >AI seminar< Speaker will be Mitchell ||
15. the >AI seminar< subject is "Explanation-Based Reasoning".

Session 9

1. reschedule AI Seminar to begin at 9:00 instead of 9:30 on June 19th.
2. Cancel LISP Tutorial Class on June 20th.
3. Class 15-731 will be held in Room 5409 instead of 7220 on June 23rd.
4. Meeting with Andy will be held in Room 7220 on June 24th.
5. List Calendar for June 25th.
6. Schedule lunch with Marsha at 12:00 on June 25th.
7. List <Flight schedule> for Pittsburgh to New York on June 27th.

8. Schedule Flight #458 on June 27th.
9. Meeting with Mike on June 28th from 10:00 to 12:00.
10. List Flight schedule for New York to Pittsburgh on June 28th.
11. Schedule Flight #115 on June 28th.
12. Dinner with Allen on June 28th at 6:00.
13. Schedule meeting with Jill Larkin in the User Studies Lab on June 26th from 10:00 to 11:00.
14. Change Speaker for AI Seminar on June 26th from Mitchell to Craig.
15. Revise Subject for AI Seminar on June 26th from "Explanation-Based Reasoning" to "Prodigy".

B.4. Utterances from User 4

Session 1

1. change >June 10< speech project meeting from 10 to 11 to 11:30
2. cancel John's speech research meeting >on june 9<
3. sceedule 7pm meeting at aisys >on june 11<
4. schedule lunch <for> andy on [Thursday] June 12 from 12p.m. to 1:30
5. change comuting <time> from CMU to Aisys on [Thursday,] june 12 from 12p.m. to 2p.m.
6. cancel Aisys meeting on [Saturday,] June 14 at 10:00
7. cancel 8:05 flight leaving NY on [Saturday] June 14 arriving in Pittsburgh at 9:25. Book flight for arrival in pittsburgh at 6p.m. schedule dinner at dads fo 6p.m. [Saturday] June 14
8. cancel flight #82 on [Saturday] June 14. Schedule flight for arrival in Pittsburgh on [Sunday] June 15, before 6 p.m. Schedule dinner at dads at 6 pm on [Sunday] June 15
9. cancel flight #82 on [Sunday] June, 15
10. what flights are avaible on [Sunday] June 15 from NY arriving in Pittsburgh before 6 p.m.
11. schedule flight #115 on [Sunday] June 15
12. schedule dinner at <dad's> at 6 p.m. [Sunday] June 15
13. change room number of seminar from 5409 to 7220 on [Tuesday,] June 10 from 12:30 to 1:30
14. schedule meeting with allen from 9:30 to 10:30 on [Thursday] June 12
15. schedule meeting with allen from 10:15 to 11:15 on [Thursday,] June 12
16. schedule meeting with allen from 10:45 to 11:45 on [Thursday,] June 12

Session 2

1. change seminar on [Tuesday,] June 10 at 12:30 from Anderson to Van Lehn
2. is anything scheduled on [Tuesday] June 10 from 3 pm to 4 pm
3. schedule meeting with Jill Larkin in Porter Hall Room 430 on [Tuesday] June 10 From 3 pm to 4 pm

4. cancel class 15-731 on [Monday] June 16, from 1:30 to 3:00
5. what flights are avaible on [Thursday] June 12, From Pittsburgh to Chicago [leaving 9 p.m. of later]-><leaving from> 9 p.m. to 11:59 p.m.<
6. what flights are avaible on [Thursday] June 12, from Pittsburgh to Chicago [leaving after 9 p.m.]->leaving from 9 p.m. to 11:59 p.m.<
7. schedule Flight #86 on [Thursday] June 12 leaving Pittsburgh at 9:30 <arriving in> Chicago at 9:54 p.m.
8. cancel flight #54 leaving Pittsburgh at 6:55 <and> arriving in NY at 8:05 on [Friday] June 13
9. what is the latest <flight> available from Chicago to NY on [Friday,] June 13
10. schedule flight #103 on [Friday,] June 13 leaving Chicago at 11 p.m. arriving in NY at 2 a.m.
11. change lunch on [Friday,] June 13 at 12 p.m. from Station Square to VC Incorporated
12. what is the schedule for [Wednesday,] June 11
13. schedule Lisp Tutorial Class from 10 a.m. to 11 a.m. on [Wednesday,] June 11
14. schedule PRODIGY with Craig (a graduate student) on [Wednesday] June 11 from 2 p.m. to 3 p.m.
15. schedule Craig <,> Prodigy on [Wednesday,] June 11 from 2 p.m. to 3 p.m.
16. schedule <the> User Studies Lab for Jill on [Wednesday,] June 11 from 3 to 4
17. schedule the User Studies Lab [Room] <for> Jill From 3 p.m. to 4 p.m. on [Wednesday,] June 11
18. schedule dinner with Allen <(a professor)> at 6 p.m. on [Wednesday,] June 11
19. Add Story Understanding and Generation to title of Natural Language Interfaces Seminar on [Thursday] June 12 at 9:30 a.m.
20. add topic Story Understanding and Generation to Natural Language Interfaces Seminar on June [Thursday] 12 at 9:30

Session 3

1. cancel commuting time from CMU to Aisys on [Thursday,] June 12 at 2 p.m.
2. the Lisp Tutorial Class on [Wednesday,] June 11 from 10 a.m. to 11 a.m. will be <in> room 8220
3. the meeting with Jill in the User Studies Lab on [Wednesday,] June 11 will be <from> 3 p.m. to 4:30 p.m.
4. schedule speech Research meeting with John on [Thursday,] June 12 from 2 p.m. to 3 p.m.
5. schedule commuting time from CMU to airport at 8:30 p.m. on [Thursday,] June 12
6. cancel meeting with Jill in the User Studies Lab from 3 p.m. to 4:30 p.m. on [Wednesday,] June 11
7. the PRODIGY meeting with Craig on [Wednesday,] June 11 will be from 3p.m. to 4 p.m.
8. schedule Department Meeting from 2p.m. to 3p.m. on [Wednesday,] June 11
9. cancel 6p.m. dinner with Allen on [Wednesday,] June 11

10. schedule meeting with Jill in the User Studies Lab from 3p.m. to 4:30p.m. on [Thursday,] June 12
11. schedule meeting with <Roger from the University of Chicago> [in Chicago] from 10 a.m. to 12 p.m. on [Friday,] June 13
12. schedule meetin on [Saturday,] June 14 from 10 a.m. to 1 p.m. with Mike at Columbia [in New york]

Session 4

1. what is the schedule for [the morning of]->6 a.m. to noon< [Thursday,] June 12
2. schedule meeting with John on [Thursday,] June 12 form 8:30 to 9:30 a.m.
3. the meeting with john on [Thursday,] June 12 from 2 p.m. to 3 p.m. will be <with> Jamie
4. cancel flight #103 from Chicago to NY on [Friday,] June 13
5. What flights are available on [Friday,] June 13 at 11p.m. form Chicago to New York
6. schedule Flight #103 on [Friday,] June 13 from Chicago to New York
7. Lunch with Andy on [Thursday,] June 12 will be from 12:30p.m. to 1:30p.m.
8. lunch with VC on [Friday,] June 13 will be from 12p.m. to 4p.m.
9. what is the schedule for [the evening of]->7 p.m. to 11:59< [Friday,] June> <13
10. schedule dinner with Roger on [Friday,] June 13
11. schedule Aisys Financial meeting on [Monday,] June 16 from 10a.m. to 11:30 a.m.
12. schedule meeting with Craig on [Monday,] June 16 from 12p.m. to 1p.m.
13. theUser Studies Lab with Jill on [Thursday,] June 12 from 3p.m. to 4:30p.m. will be on [Monday,] June 16 from 3p.m. to 4:30 p.m.
14. The User Studies Lab Meeting with Jill of [Thursday,] June 12 from 3p.m. to 4:30p.m. will be on [Monday,] June 16 for 3p.m. to 4:30p.m.
15. what is the schedule for [the evening of]->7 p.m. to 11:59< [Thursday,] June 12
16. schedule dinner with Marsha on [Thursday,] June 12 from 6p.m. to 7:30p.m.

Session 5

1. cancel flight #115 on [Sunday,] June 15 from New York to Pittsburgh
2. what flights are available on [Saturday,] June 14 from New York to Pittsburgh [after 3p.m]->from 3p.m. to 11:59 p.m.<
3. Schedule flight #115 on [Saturday,] June 14
4. The meeting with Mike at Columbia on [Saturday,] June 14 will be from 10a.m. to 11:30a.m.
5. Schedule lunch with Bob on [Saturday,] June 14 from 12p.m. to 1p.m.
6. Schedule meeting with Jamie on [Sunday,] June 15 from 11a.m. to 12p.m.
7. Cancel meeting with Roger at <the University of Chicago> on [Friday,] June 13 from 10a.m. to 12p.m.
8. Schedule Lisp Tutorial Class on [Monday,] June 16 from 9a.m. to 10a.m.
9. The Financial Meeting on [Monday,] June 16 from 10a.m. to 11:30a.m. will be <at> CMU

10. Schedule meeting with Andy in Room 7220 on [Tuesday,] June 17 from 10a.m. to> <11a.m.
11. Schedule Speech Project Meeting in room 7220 on [Tuesday,] June 17 from 11a.m. to 12p.m.
12. The CogSci Seminar on [Tuesday,] June 17 from 12:30p.m. to 1:30p.m. in room 5409 will be <with> Anderson

Session 6

1. The Lisp Tutorial Class on [Monday,] June 16 from 9a.m. to 10a.m. will be <on> [Tuesday,] June 17 from 9a.m. to 10a.m.
2. Lunch with Craig on June [Monday,] 16 from 12p.m. to 1:30p.m. will be from 12:30p.m. to 1:30p.m.
3. cancel the meeting with Andy in Room 7220 on [Tuesday,] June 17 from 10a.m. to 11a.m.
4. Schedule Al Seminar with Drew McDermott on [Monday,] June 16 from 2p.m. to 3p.m. <the topic will be> "Non-Monotonic Logics"
5. The meeting with Jill on [Monday,] June 16 from 3p.m. to 4:30 p.m. will be in the office
6. The Speech Project Meeting on [Tuesday,] June [16]->17< from 11a.m. to 12p.m. will be room 8220
7. what flights are available on [Wednesday,] June 25 from Pittsburgh to New York [after &;30 a.m.]->from 7:30 a.m. to 11:59 p.m.<
8. schedule flight #250 on [Wednesday,] June 25
9. Schedule Meeting with Mike at <Columbia> University [in new york] from 11a.m. to 12:30p.m.
10. Schedule lunch with Bob on [Wednesday,] June 25 at 12:30p.m. [in New York]
11. What flights are available on [Wednesday,] June 25 from New York to Pittsburgh [before 6 p.m.]->from 6 a.m. to 6p.m.<
12. what flights are available on [Wednesday,] June 25 from New York to Pittsburgh at 6p.m.
13. Schedule flight #115 on June [Wednesday,] 25

Session 7

1. The meeting on [Tuesday,] June 17 from 11a.m. to 12p.m. will be in room 7220
2. cancel flight #250 on [Wednesday,] June 18
3. what flights are available on [Wednesday,] June 18 from Pittsburgh to New york [arriving befor 9 a.m.]->from 6 a.m.to 9 a.m.<
4. schedule flight #616 on [Wednesday,] June 18
5. The meeting with Mike on [Wednesday,] June 18 will be from 9a.m. to 10a.m.
6. The meeting on [Wednesday,] June 18 with Mike will be at Columbia University [in New York]
7. Lunch on [Wednesday,] June 18 will be with Richard
8. Schedule commuting time from CMU to Aisys at 11a.m. on [Thursday,] June 19
9. Schedule Financial Meeting at 12p.m. on [Thursday,] June 19

10. Jamie will be the speaker at the 9:30a.m. Seminar on [Thursday,] June 19
11. cancel the Ai seminar on [Friday,] June 20
12. what flights are available on [Friday,] June 20 from Pittsburgh to Chicago [at 9a.m. or later]->from 9a.m. to 11:59p.m.<
13. schedule flight #86 on [Friday,] June> <20
14. what flights are available on [Saturday,] June 21 from Chicago to Pittsburgh [before 7:30p.m.]->from 6 a.m. to 7:30p.m.<
15. schedule flight #11 on [Saturday,] June 21
16. The 1:30p.m. class on [Monday,] June 23 will be in room 7220

Session 8

1. the seminar in Room 5409 on [Thursday,] June 19 will be with Jill
2. schedule commuting time from CMU to Aisys from 2:30p.m. to 3p.m. on [Friday,] June 20
3. Schedule Lisp Tutorial Class from 3p.m. to 4p.m. on [Friday,] June 20
4. Cancel Fligth #11 on [Saturday,] June 21
5. schedule Prodigy, Craig on [Monday,] June 23 from 10a.m. to 11a.m.
6. schedule lunch with Marsha on [Monday,] June 23 at 12p.m.
7. schedule Meeting with Andy from 10a.m. to 11a.m. on [Tuesday,] June 24
8. Schedule meeting with VC (finances) at Station Square on [Tuesday,] June 24 from 2p.m. to 3p.m.
9. schedule VC finances meeting at Station Square from 2p.m. to 3p.m. on [Tuesday,] June> <24
10. cancel seminar in room 5409 on [Thursday,] June 26
11. Schedule AI Seminar on [Thursday,] June 26 from 4p.m. to 5:30p.m. <Speaker:> Mitchell <Subject:> Explanation Based Reasoning

Session 9

1. The seminar in room 5409 on [Thursaday,] June 19 will be an AI Seminar
2. cancel Lisp Tutorial class on [Friday,] June 20
3. class 15-731 on [Monday,] June 23 will be in room 5409
4. The meeting with Andy On [Tuesday,] June 24 will be in room 7220
5. Schedule Lunch with Marsha on June 25 at 12p.m.
6. what flights are available on [Friday,] June 27 from Pittsburgh to New York [after 6p.m.]->from 6p.m. to 11:59p.m.<
7. what flights are available on [Friday,] June 27 from Pittsburgh to New York at 6p.m.
8. schedule flight #458 on [Friday,] June 27
9. Schedule meeting with Mike [in New York] on [Saturday,] June 28 from 10a.m. to 12p.m.
10. what flights are available from New York to Pittsburgh on [Saturday,] June 28 [after 12p.m.]->from 12p.m. to 11:59 p.m.<
11. Schedule flight #115 on [Scturday,] June 28

12. Schedule dinner with Allen on [Saturday,] June 28 at 6p.m.
13. Schedule user studies lab meeting with Jill Larkin on [Thursday,] June 26 from 10a.m. to 11a.m.
14. The speaker for the AI seminar on [Thursday,] June 26 will be Craig

B.5. Utterances from User 5

Session 1

1. June 9th the Speech Project Meeting that was to be held from 10:00 to 11:00 will be changed to 10:00 to 11:30.
2. change the speech meeting project from 10:00 to 11:30 instead of 10:00 to 11:00
3. cancell meeting on june 9th dealing with speech research time was 9:00 to 10:00
4. cancel meeting on june 9 [meeting was to be held] from 9:00 to 10:00 [please cancel].
5. schedule meeting <for> [Wednesday,] June 11 at AISyS at 7:00 p.m.
6. Andy will be <at> lunch from 12:00 to 1:30 on [Thursday,] June 12.
7. change departing time from 12:00 to 2:00, on [thursday,] june 12th. will be leaving now from cmu to aisys at 2:00.
8. cancel aisys meeting at 10:00 [saturday,] june 14th.
9. on june 15 change flight #82 to be in Pittsburgh before 6:00. schedule dinner at 6:30.
10. On June 15th change flight times so that arrival in Pittsburgh is before 6:00.
11. on june 10th change the seminar room from 5409 to 7220.

Session 2

1. On June 9th Seminar to be held in Room 7220 by Anderson will now be held by VanLehn.
2. Schedule Jill Larkin in Porter Hall <,> Room 430 on June 10 from 3:00 to 4:00.
3. Cancel class 15-731 on [Monday] June 16th from 1:30 - 3:00.
4. show me the airline database.
5. show me the airline database for [Thursday,] June 12th.
6. Show me the airline database for [Thursday] June 12th <, flights leaving> Pittsburgh <and arriving in> Chicago.
7. Schedule flight #86 leaving Pittsburgh at 9:30 p.m. and arriving in Chicago at 9:54p.m. [leaving Pittsburgh arriving in Chicago] on June 12th
8. Cancel flight #54 leaving Pittsburgh <arriving in> New York from 7:00 to 8:00 >on june 13<.
9. Show me the airline database for June 13th, flight leaving chicago and arriving in New York.
10. Schedule flight #103 leaving Chicago at 11:00 p.m. and arriving in New York at 2 a.m.
11. Change the location of lunch on June 13th from 12:00 to 1:00 from Station Square to VC Incorporated.

12. Schedule Lisp Tutorial Class from 10:00 to 11:00 on June 11th.
13. Schedule Craig with the Prodigy Group on June 11th from 2:00 to 3:00.
14. Schedule a meeting <for> Craig on June 11th from 2:00 to 3:00 <for> Prodigy.

Session 3

1. Cancel 2:00 <appointment> from CMU to AISyS on [Thursday,] June 12th.
2. Schedule Lisp Tutorial Class in Room 8220 from 10:00 to 11:00 on June 11.
3. Extend Jill's appointment in <the> User Studies Lab to 3:00 to 4:30 on June 11.
4. Schedule Speech Research Meeting <for> John from 2:00 to 3:00 on June 12 .
5. leaving CMU at 8:30 p.m. [thursday,] June 12th, going to the airport.
6. Departing CMU at 8:30 p.m. on [thursday,] June 12th to the airport.
7. Cancel Jill's <time> in the User Studies Lab from 3:00 to 4:30 on June 11.
8. Reschedule Craig's Prodigy Meeting from 2:00 to 3:00 to 3:00 to 4:00.
9. Schedule Department Meeting from 2:00 to 3:00 on June 11th.
10. Cancel Allen's dinner reservations at 6:00 on June 11.
11. Schedule the User Studies Lab for Jill from 3:00 to 4:30 on June 12th.
12. Schedule appointment for Roger at <the University of Chicago> from 10:00 to 12:00 on June 13th.
13. Schedule meeting at Columbia for Professor Mike on June 14th from 10:00 to 1:00.

B.6. Utterances from User 7

Session 1

1. I would like to change the speach project meeting on June 10th to end at 11:30 instead of 11:00
2. The speech meeting on June 10th will begin at 10:00 and end at 11:30 instead of 11:00
3. Please reschedule the speech project meeting for June 10th, the new times are from 10:00 to 11:30
4. Cancel the Speach Research meeting with John on [Monday,] June 9
5. Schedule an AISys meeting on June 11 at 7:00pm
6. schedule lunch with Andy on June 12 from 12:00 [noon] until 1:30
7. Leaving from CMU to AISys at 2:00 instead of 12:00
8. Change the 12:00 to 2:00 on June 12 traveling from CMU to AISys
9. cancel AISys meeting on June 14
10. what flights are there from Pittsburgh to New York on June 15, arriving before 5:30pm
11. cancel airline reservations for June 15, flight #82
12. cancel flight #82 on June 15
13. list <flight> info <for> flights from New York to Pittsburgh on June 15 [arriving in Pittsburgh] [before 5:30 pm]->from 6 am to 5:30 pm<

14. make reservation for flight #115 lv. New York at 4:00pm ar. Pittsburgh 5:10pm on June 15
15. June 15 6:00pm dinner with Dad
16. CogSci seminar changed to room 7220
17. CogSci seminar on June 10 changed to room 7220
18. what is my schedule between 9:00 and 12:00 on JUne 12
19. 10:45 - 11:45 Allen

Session 2

1. change speaker <for> CogSci [meeting]->seminar< on June 10 from Anderson to VanLehn
2. schedule meeting Jill Larker <,> Porter Hall <,> Rm <.> 430 3:00 - 4:00 June 10
3. cancel class 15-731, June 16
4. change lunch on June 13 from Station Square to VC Incorporated
5. 10:00 - 11:00 June 11 Lisp, Tutorial Class 2:00 - 3:00 June 11 Craig, Prodigy 3:00 - 4:00 June 11 User Studies, Lab Jill 6:00 June 11 Dinner, Allen
6. 10:00 - 11:00 June 11 Lisp, Tutorial Class
7. schedule <the> Lisp <,> Tutorial Class on June 11 from 10:00 - 11:00
8. 2:00 - 3:00 June 11 meeting with Craig <, subject> Prodigy
9. 3:00 - 4:00 June 11 User Studies Lab Jill
10. 3:00 - 4:00 June 11 User Studies Lab <,> Jill
11. 6:00pm June 11 Dinner with Allen
12. Add the topic "Story Understanding and Generation" to the Natural Language Interfaces Seminar on June 12
13. Add "Story Understanding and Generation" to the Natural Language Interfaces Seminar on June 12
14. Airline schedule needed for Pittsburgh to Chicago [around 9:00pm]->at 9:00pm<
15. List airline schedule from Pittsburgh to Chicago on June 12 between 8:00 and 10:00
16. schedule flight #86 <from lv.> Pittsburgh 9:30pm ar. Chicago 9:54pm
17. cancel flight #54 on June 13
18. list airline schedule from Chicago to New York on June 13 between 8:00pm and 12:00[pm]->am<
19. schedule flight #103 1:00pm Chicago 2:00am New York

Session 3

1. cancel CMU to AISyS at 2:00 on June 12
2. add room 8220 to 10:00 Lisp Tutorial Class on June 11
3. change ending time from 4:00 to 4:30 for User Studies Lab with Jill on June 11
4. June 12 2:00 - 3:00 John <,> Speech Research
5. June 12 CMU to Airport 8:30pm
6. cancel User Studies Lab with Jill on June 11 at 3:00

7. change the meeting with Craig on June 11 to begin at 3:00 and end at 4:00
8. Meeting with Craig on June 11 3:00 to 4:00
9. June 11 Craig, Prodigy 3:00 to 4:00 instead of 2:00 to 3:00
10. June 11 2:00 to 3:00 Department Meeting
11. del dinner with Allen at 6:00 on June 11
12. June 12 3:00 to 4:00 Jill in <the User Studies Lab>
13. add 10:00 to 12:00 June 13 Roger at <the Univeristy of Chicago>
14. add 10:00 to 1:00 June 14 Columbia, Mike

Session 4

1. please list schedule for [Thursday] June 12
2. 8:00 to 9:00 June 12 meeting with John
3. change 2:00 - 3:00 from John, Speech Research to Jaime
4. del Chicago to New York Flight #103 on June 13
5. list <flight schedule> for June 13 from Chicago to New York
6. schedule Flight #71 8:05pm Chicago 11:05pm NY
7. change >june 12< Andy, Lunch from 12:00 - 1:30 to 12:30 to 1:30
8. change end time of lunch with VC on June 13 from 1:00 to 4:00
9. list schedule for June 13
10. schedule dinner with Roger at 5:00 on June 13
11. schedule 10:00 - 11:30 AISyS Meeting, Financial
12. schedule 10:00 - 11:30 AISyS Meeting, Financial June 16
13. schedule 12:00 - 1:00 Craig June 16
14. change 3:00 - 4:30 Jill, User Studies Lab from June 12 to June 14
15. list schedule for June 12
16. schedule 5:30 Dinner with Marsha June 12

Session 5

1. cancel flight #115 on June 15
2. list flight schedule for June 14 New York to Pittsburgh
3. schedule Flight #115 4:00pm NY 5:10pm Pittsburgh June 14
4. change meeting with Mike to end at 11:30 instead of 1:00 June 14
5. 12:00 Lunch with Bob June 14
6. 11:00 - 12:00 June 15 Jaime
7. cancel meeting with Roger June 13
8. 9:00 - 10:00 Lisp Tutorial Class June 16
9. change AISyS to CMU <for> Financial Meeting June 16
10. 10:00 - 11:00 Room 7220 with Andy June 17
11. 11:00 - 12:00 Room 7220, Speech Project June 17
12. change Newell to Anderson <on> CogSci Seminar June 17

B.7. Utterances from User 9

Session 1

1. Arrange Speech Meeting
2. Schedule <speech proj.> meeting
3. <schedule> 9:00am meeting June 5 to 10:00am to 11:30am
4. <schedule> 9:00am meeting June 6 to 10:00am to 11:30 June 6
5. schedule AISys meeting June 7 7:00pm
6. schedule AISys meeting 7:00pm June 7
7. AISys meeting 7:00pm June 7
8. 7:00pm June 7 AISys meeting
9. 12:00 - 1:30 Lunch with Andy June 8
10. <departure> from CMU to AISys 4:00pm June 8
11. leave 4:00pm <for> AISys
12. show schedule
13. 4:00pm Go from CMU to AISys
14. 4:00pm Go from CMU to AISys
15. show schedule
16. cancel 3:00pm <Go to> CMU to AISys
17. cancel 3:00pm <Go> from CMU to AISys
18. show schedule
19. show schedule for June 10
20. cancel 10:00am Meeting in/at Office with AISys June 10
21. schedule dinner with Dad 6:30pm June 10
22. show airline schedule
23. cancel flight #82 June 11 NY to Pgh
24. show airline schedule
25. schedule Flight 115 NY to PGH 4:00pm June 11
26. change location of COGSCI Seminar to Room 7220
27. change location of COGSCI Seminar to Room 7220 June 6
28. show allen schedule
29. see Allen schedule
30. Meeting with Allen 10:00am June 8
31. schedule meeting 10:00am with Allen June 8
32. schedule meeting 11:00am with Allen June 8

Session 2

1. change seminar speaker to van lehn
2. change speaker to Van Lehn June 6
3. show schedule

4. show schedule june 6
5. change speaker COGSCI SEMINAR to Van Lehn June 6
6. meeting June 6 3:00pm to 4:00pm Porter Hall 4300 Jill Larkin
7. show schedule
8. schedule meeting with Jimm Larkin on June 6 3:00pm to 4:00pm Porter Hall 4300
9. June 6 meeting with Jill Larkin 3:00pm to 4:00pm Porter Hall 4300
10. show schedule June 12
11. cancel 15-731 in/at 5409 June 12 1:30pm to 3:00pm
12. June 12 cancel class 15-731 at 1:30 in 5409
13. show schedule June 8
14. show airlines schedule Pgh to Chicago
15. show airline schedule Pgh to Chicago June 8
16. reservation June 8 to Chicago Flight #192 at 8:10pm
17. June 9 cancel flight 54 at 7:00pm
18. show <flight schedule> chicago to new york June 9
19. schedule Flight #103 11:00pm CHI to New York June 9
20. show schedule June 9
21. June 9 lunch 12:00pm - 1:00pm with VC at VC Inc.
22. June 9 12:00pm - 1:00pm at VC Incorporated
23. June 9 lunch at VC Incorporated 12:00pm - 1:00pm
24. June 7 LISP tutorial 10:00am - 11:00am
25. show schedule June 7
26. June 7 PRODIGY meeting with Craig 2:00pm - 3:00pm
27. show schedule june 7
28. June 7 User studies lab with Jill 3:00pm - 4:00pm
29. June 7 dinner with allen 6:00pm
30. June 8 show schedule
31. June 8 Seminar subject "Story Understanding and Generation"

Session 3

1. june 8 show calendar
2. June 8 cancel from CMU to AISYS
3. show June 7 calendar
4. June 7 10:00am - 11:00am LISP tutorial in Room 8220
5. June 7 show calendar
6. June 7 10:00am - 11:00am LISP TUTORIAL in Room 8220
7. June 7 show calendar
8. June 7 change meeting 3:00pm - 4:00pm in user studies lab to 3:00pm - 4:30pm
9. June 8 show calendar
10. June 8 2:30pm - 3:00pm with John about Speech research

11. June 8 leave CMU <for> Airport 8:30pm
12. June 7 cancel meeting with Allen user studies lab 5:00pm - 6:00pm
13. June 9 change meeting with Craig <on> prodigy <at> 2:00 - 3:00 pm to 3:00pm - 4:00pm
14. department meeting June 7 2:00pm - 3:00pm
15. cancel dinner with Allen 6:00pm June 7
16. Meeting 9:00am - 10:00am with Jill in user studies lab June 8
17. show schedule June 9
18. show flight schedule june 9
19. meeting with Roger 10:00am - 12:00pm <University of Chicago> June 9
20. meeting at Columbia University with Mike June 10 10:00am - 1:00pm

Session 4

1. show calendar June 8
2. meeting with John 10:30am - 11:30am June 8
3. meeting with John 8:30am - 9:30am June 8
4. change Speech research meeting with John to meeting with Jaime 2:00pm - 3:00pm June 8
5. show calendar June 8
6. topic of meeting with Jaime 2:00pm - 3:00pm unknown
7. subject of meeting with Jaime 2:00pm - 3:00pm delete subject
8. show flight schedule June 9
9. cancel flight #103 June 9 CHI - NY
10. show calendar June 9
11. show flight schedule june 9
12. change CHI flight from #103 to #71 at 8:05pm June 9
13. cancel flight #103 June 9
14. schedule flight #71 CHI to NY at 8:05 pm June 9
15. change lunch with Andy from 12:00 to 12:30 pm June 8
16. change lunch with andy to 12:30 pm June 9
17. change lunch with andy on June 8 from 12:00pm to 12:30pm - 1:30pm
18. change lunch with <VC Inc> June 9 from 12:00pm - 1:00pm to 12:00pm - 4:00 pm
19. show june 9 calendar
20. dinner with roger June 9 5:00pm - 7:00pm
21. show calendar June 12
22. financial meeting at AISYS 10:00am - 11:30am june 12
23. meeting with craig 12:00pm - 1:00pm June 12
24. change meeting June 8 3:00pm - 4:30pm in User studies lab to June 12 3:00pm - 4:30pm in user study lab
25. show calendar june 8

26. dinner with tom june 8 6:00pm

Session 5

1. show <flights> June 11
2. show flight schedule June 10
3. cancel flight #115 June 11
4. flight #115 June 10
5. show calendar June 10
6. change meeting June 10 at Columbia with Mike 10:00am -1:00 pm to 10:00am - 11:30am
7. lunch with Bob June 10 at 12:00pm
8. show calendar June 11
9. meeting with Jaime June 11 11:00am - 12:00pm
10. Cancel meeting with Roger June 9 10:00am - 12:00pm
11. show calendar for June 12
12. LISP tutorial 9:00am - 10:00am June 12
13. financial meeting June 12 10:00am - 11:30am <at> CMU
14. show calendar June 13
15. meeting with Andy 10:00am -11:00am in Room 7220 June 13
16. Speech Project in Room 7220 June 13 11:00am - 12:00pm
17. Room 7220 11:00am - 12:00pm Speech Project
18. show June 13 calendar
19. June 13 COGSCI Seminar in 5409 12:00pm - 1:30pm <with> Anderson <not> Newell

Session 6

1. show June 12 calendar
2. move LISP TUTORIAL class from June 12 9:00am - 10:00am to June 13 9:00am -10:00am
3. change LISP Tutorial from June 12 to June 13 9:00am - 10:00am
4. change lunch with Craig June 12 to 12:00pm - 1:30pm
5. cancel Meeting with Andy June 13
6. June 12 change Meeting in User Studies Lab 3:00pm - 4:30pm to Office
7. AI Seminar Drew McDermott 2:00pm - 3:00pm "Non-Monotonic Logics" June 12
8. June 13 11:00am - 12:00pm Speech Project change to 8220
9. show June 21 calendar
10. LUnch June 21 with Bob 12:00pm
11. show flight schedule June 21
12. flight #250 8:00am June 21
13. June 21 meeting with Mike at Columbia 11:00

14. show June 21 calendar
15. show flight schedule June 21
16. flight #265 June 21

Session 7

1. Speech project meeting June 13 in 7220
2. June 13 speech project meeting in 8220 changed to 7220
3. show flight schedule June 14
4. show June 14 schedule
5. show flight schedule June 14
6. change flight #250 June 14 to flight #616 June 14
7. change meeting at Columbia with Mike to 9:00am - 10:00am June 14
8. lunch June 14 at 12:00pm with Richard not Bob
9. show calendar June 15
10. leave CMU to AISYS June 15 at 11:00am
11. financial meeting June 15 at 12:00pm
12. June 15 Natural Language Interfaces seminar Room 5409 at 9:00am to 10:30am with Jaime
13. Natural Language Interfaces seminar with Jaime on June 15 9:30am - 10:30am in 5409
14. show schedule June 15
15. Natural Language Interfaces seminar with Jaime in 5409 9:30am - 10:30am June 15
16. cancel AI seminar June 16 10:00am - 11:00am
17. show flight schedule June 16
18. flight #86 to Chicago June 16
19. flight #11 to Pittsburgh June 17
20. show calendar June 17
21. 15-731 class June 19 in room 7220 1:30pm - 3:00pm
22. show calendar June 19
23. change 15 731 class June 19 1:30pm - 3:00pm to 7220
24. Move 15-731 class on June 19 at 1:30pm - 3:00pm from 5409 to 7220
25. change class on June 19 at 1:30pm - 3:00pm to room 7220

Session 8

1. show calendar June 15
2. change June 15 seminar at 9:30am to Jill
3. show calendar June 16
4. leave AISYS for CMU at 2:30pm June 16
5. LISP tutorial class June 16 3:00pm - 4:00pm
6. show flight schedule June 17

7. cancel flight #11 June 17
8. flight #16 at 8:20pm June 17
9. show calendar June 19
10. meeting with Craig 10:00am - 11:00am June 19
11. meeting June 19 with Craig at 10:00am about Prodigy
12. Prodigy meeting with Craig at 10:00am on June 19
13. cancel meeting with Craig June 19 at 10:00am
14. Prodigy meeting with Craig on June 19 at 10:00am - 11:00am
15. show calendar June 19
16. Lunch June 19 at 12:00pm with Tom
17. meeting at 10:00am - 11:00am with Andy on June 20
18. show calendar June 20
19. financial meeting with VC at station square 2:00pm - 3:00pm June 20
20. show June 22 calendar
21. cancel natural language interfaces seminar at 9:30am on June 22
22. AI seminar " Explanational based reasoning" 4:00pm - 5:30pm with Mitchell June 22

Session 9

1. show June 15 calendar
2. change Natural Language Interfaces seminar at 9:30am to AI seminar at same <time>
3. cancel LISP tutorial June 16 at 3:00pm
4. June 19 15-731 class move to 5409 same time
5. change class 15-731 June 19 from 7220 to 5409 at 1:30pm
6. show June 20 calendar
7. change meeting with Andy from office to 7220 June 20 at 10:00am
8. show calendar June 21
9. lunch with Tom June 21
10. show calendar June 23
11. show flight schedule June 23
12. flight #458 to NY at 6:55pm June 23
13. June 24 show calendar
14. show flight schedule June 24
15. flight #115 4:00pm to Pgh June 24
16. meeting with Mike 10:00am - 12:pm June 24
17. meeting with Mike June 24 at 10:00am - 12:00pm
18. Dinner June 24 with Allen
19. show calendar June 22
20. June 22 User Studies Lab 10:00am - 11:00am Jill
21. June 22 change AI seminar with Mitchell to Craig on Prodigy 4:00pm - 5:00pm

B.8. Utterances from User 10

Session 1

1. The speech project meeting on June 6 <will last until> 11:30
2. Cancel the June 5 meeting with John about speech research
3. Set up a meeting on June 7 at 7:00 PM at AISys
4. Arrange a meeting on June 8 from 12 to 1:30 for lunch with Andy
5. Set up a meeting 12:00 to 1:30 PM on June 8 with Andy
6. Set up a meeting 12 to 1:30 on June 8 with Andy at <the Faculty Dining Room>
7. <Leave for> meeting at AISys on June 8 at 4:00 <instead of> 3:00
8. Leave at 4:00 on June 8 for meeting at AISys
9. Go to AISys on June 8 at 4:00
10. View schedule
11. <Do not go> to AISys on June 8 at 3:00 PM
12. Cancel 3:00 PM on June 8 to AISys
13. view schedule
14. Cancel Saturday June 10 AISys
15. Meet Dad <for> dinner June 11 at 6:00 PM
16. Dinner June 11 6:30 PM with Dad
17. Dinner with Dad June 11 at 6:30 PM at Poli's
18. Change June 11 NY to Pittsburgh flight to leave NY about 6:15 PM
19. view airline schedule June 11 NY to Pittsburgh
20. Change June 11 NY to Pittsburgh from flight 82 to flight 265
21. Change from flight 82 to flight 265 on June 11 NY to Pittsburgh
22. Change flight 82 to flight 265 on June 11
23. view airline schedule June 11 NY to Pittsburgh
24. Change flight 82 to flight 115 on June 11
25. Change <room> of June 6 Cogsci seminar to 7220
26. Change June 6 Cogsci seminar to WeH 7220
27. view schedule June 8
28. schedule meeting with Allen June 11 10:30 to 11:30
29. schedule meeting with allen on June 8 at 10:30 PM in <his office>
30. schedule meeting with allen on June 8 at 10:30 AM in WeH 5216

Session 2

1. On June 6 change Cogsci Seminar speaker to Van Lehn
2. Schedule meeting with Jill Larkin June 6 3:00 - 4:00 in Room 4300
3. Schedule meeting with Jill Larkin on June 6 in Porter Hall 4300
4. Change June 6 meeting with Jill Larkin to Porter Hall 4300
5. cancel June 12 1:00 class 15-731

6. view airline schedule
7. schedule flight 192 Pittsburgh to Chicago on June 8
8. Change flight 54 June 9 to flight 103
9. Change June 9 lunch at Station Square to VC Incorporated
10. schedule June 7 lisp tutorial class 10:00 - 11:00 am
11. Schedule PRODIGY meeting June 7 <at> 2:00 - 3:00 PM with Craig
12. Schedule meeting June 7 3:00 - 4:00 PM with Jill in <the User Studies Lab>
13. Schedule dinner June 7 at 6:00 PM with Allen
14. <Add> subject "Story Understanding and Generation" to June 8 Natural Language Interfaces Seminar
15. Subject of June 8 Natural Language Seminar is "Story Understanding and Generation"

Session 3

1. Cancel June 8 meeting at AISys
2. Cancel June 8 meeting at 2:00 at AISys
3. Cancel June 8 at 2:00 at AISys
4. Lisp Tutorial Class on June 7 at 10:00 am <is in> Room 8220
5. June 7 meeting with Jill at 3:00 will last until 4:30 PM
6. Schedule meeting on June 8 at 2:00 - 3:00 PM with John about speech research
7. <Leave for> airport at 8:30 PM on June 8
8. Cancel June 7 meeting at 5:00 with allen
9. Change June 9 meeting with Craig to 3:00 - 4:00 PM
10. Schedule June 7 department meeting at 2:00 - 3:00 PM
11. Cancel June 7 dinner with Allen
12. Schedule meeting June 8 with Jill 9:00 - 10:00 am in User Studies Lab
13. Schedule meeting June 9 10:00 - 12:00 <noon> with Roger at <the University of Chicago>
14. Schedule meeting June 10 10:00 am - 1:00 PM with Mike at Columbia University

Session 4

1. view June 8 schedule
2. schedule June 8 4:30 - 5:30 PM meeting with John
3. cancel June 8 4:30 meeting with John
4. schedule June 8 meeting with John at 8:30 a.m.
5. Change June 8 meeting with John to Jaime
6. change June 8 meeting at 2:00 to Jaime
7. change June 8 meeting at 8:30 to John
8. view airline schedule
9. change June 9 flight 103 to flight 71
10. Change June 8 lunch with andy to 12:30 PM

11. June 9 lunch with VC will last until 4:00 PM
12. view schedule june 9
13. schedule June 9 dinner with roger at 6:00 PM
14. view schedule June 9
15. Schedule meeting June 12 10:00 - 11:30 with AISYS about finances
16. schedule June 12 meeting with Craig 12:00 to 1:00 PM
17. view schedule june 12
18. Move June 8 meeting with Jill to June 12 3:00 - 4:30 PM
19. view schedule june 8
20. schedule dinner with tom at 7:00 PM on June 8

Session 5

1. view airline schedule June 11
2. view schedule june 9
3. view schedule june 10
4. view airline schedule june 10
5. change June 11 flight 115 to June 10 flight 115
6. Change flight 115 from June 11 to June 10
7. appointment with Mike on June 10 will last until 11:30 AM
8. schedule lunch with Bob at 12:00 on June 10
9. view schedule june 11
10. schedule meeting with Jaime June 11 11:00 - 12:00 AM
11. cancel June 9 meeting with Roger
12. view schedule june 12
13. schedule lisp tutorial class 9:00 - 10:00 on June 12
14. move June 12 financial meeting from AISys to CMU
15. schedule meeting June 13 from 10 to 11 AM with Andy in Room 7220
16. schedule speech project meeting June 13 11-12 in Room 7220
17. Speaker at June 13 Cogsci seminar <will be> Anderson not Newell
18. change June 13 cogsci seminar speaker to Anderson

Session 6

1. view schedule june 12
2. Change June 12 lunch with Craig to 12:30 - 1:30 PM
3. Cancel June 13 meeting with Andy
4. view schedule june 13
5. schedule lisp tutorial class June 12 9 - 10
6. schedule lisp tutorial class June 13 9-10
7. view schedule june 12
8. schedule AI seminar by Drew McDermott June 12 <on the topic of> Non-Monotonic Logics 2-3

9. Schedule June 12 2-3 PM AI seminar with Drew McDermott <on Non-Monotonic Logics>
10. Schedule June 12 2-3 PM AI seminar with Drew McDermott <on the subject of Non-Monotonic Logics>
11. Schedule June 12 2-3 PM AI seminar by Drew McDermott about <Non-Monotic Logics>
12. Schedule June 12 2-3 AI seminar with Drew McDermott about <Non-Monotonic Logics>
13. Move June 12 meeting with Jill to office
14. Move June 13 speech project meeting to 8220
15. view schedule june 21
16. view airline schedule june 21
17. schedule June 21 flight 250
18. schedule June 21 meeting with Mike at Columbia at 11:00
19. schedule June 21 lunch meeting with Bob at 12:30
20. schedule June 21 meeting with Bob at 12:30 at Metropolitan Museum <of> Art
21. view airline schedule June 21
22. schedule June 21 flight 265

Session 7

1. Change June 13 speech meeting to Room 7220
2. June 13 meeting at 11:00 <is with the> Speech Project
3. view airline schedule june 14
4. change june 14 flight 250 to flight 616
5. view schedule june 14
6. change june 14 meeting with mike to 9-10 AM
7. change june 14 lunch from Bob to Richard
8. view schedule june 15
9. view schedule june 14
10. Go to AISys at 11:00 June 15
11. schedule meeting june 15 at noon about finances
12. view schedule june 15
13. june 15 meeting at noon <is at> AISys
14. june 15 natural language seminar speaker is Jaime
15. Jaime <will speak at> the june 15 natural language seminar
16. view schedule june 15
17. Change speaker <at> June 15 Natural Language seminar to Jaime
18. cancel june 16 ai seminar
19. view schedule june 16
20. view airline schedule june 16
21. schedule flight 192 on June 16

22. schedule flight 11 on June 17
23. view schedule june 19
24. change june 19 class to room 7220

Session 8

1. Change June 15 Natural Language Seminar speaker to Jill
2. view schedule June 16
3. Leave AISys at 2:30 on June 16 <to return> to CMU
4. Schedule Lisp Tutorial Class June 16 3-4 PM
5. view schedule june 17
6. view airline schedule
7. change june 17 from flight 11 to flight 16
8. Change from flight 11 to flight 16 on June 17
9. Change June 17 flight 11 to flight 16
10. schedule meeting on June 19 from 10-11 am with Craig about Prodigy
11. schedule lunch with Tom at noon on June 19
12. schedule meeting with Andy from 10-11 am on June 20
13. schedule meeting on June 20 2-3 pm at Station Square with VC about Finance
14. cancel natural language seminar on June 22
15. schedule ai seminar with Mitchell 4-5 pm June 22 about Explanation Based Reasoning

Session 9

1. Change June 15 Natural Language Seminar to AI Seminar
2. cancel June 16 lisp tutorial class
3. Change June 19 class to room 5409
4. meeting with Andy on June 20 is in room 7220
5. schedule lunch with Tom on June 21
6. view airline schedule
7. schedule flight 458 on June 23
8. schedule flight 115 on June 24
9. schedule meeting on June 24 10-12 am with Mike
10. schedule dinner June 24 with Allen
11. schedule meeting June 22 10-11 am with Jill in the User Studies Lab
12. change June 22 ai seminar speaker to Craig about Prodigy
13. June 22 ai seminar speaker is Craig speaking about prodigy
14. Change June 22 AI seminar speaker to craig
15. change June 22 ai seminar subject to Prodigy

Appendix C
Performance Measurements for User Data

This appendix contains the raw data upon which much of the discussion in Chapter 9 is based. All data reflect CHAMP's performance on the user's utterances. Thus, data presented for users in adaptive conditions in the hidden-operator experiments (Users 1, 2, 3, 4, 5, and 7) reflect the modifications discussed in Section 9.1.1. The reader is encouraged to review that section before continuing.

C.1. Users in Hidden-operator Experiments

A summary is presented for each user who participated in the hidden-operator experiments. The meaning of each abbreviation is explained below:

S#	the session number. A star next to the number indicates that the supplementary instructions to work quickly were given at the beginning of that session.
Utt	the number of utterances after modification. After breaking them apart, *k* conjunctive phrases count as *k* sentences.
New[raw]	the number of derived components learned by CHAMP during the session. The measure includes neither components added by hand to compensate for differences between the simulation and implementation (see Add, below) nor instances of non-deviant learning (see L0, below). The first number takes competition relations into consideration, while the bracketed number shows the raw count. Thus, "3[6]" means that six components were derived as three competition pairs.
Add	the number of utterances after which extra grammar components were added by hand. Additions not marked by "(F)" represent learning at Deviation-level 0 (new instances, abbreviations, or permanent spelling corrections).
Rem	the number of components removed by a resolved competition.
Int	the number of utterances requiring some kind of interference (other than Add) to compensate for differences between the simulation and implementation.
Dif	the number of utterances accepted by the hidden-operator but rejected by CHAMP.

L0 the number of components derived at Deviation-level 0 (new instances, abbreviations, or permanent spelling corrections).

Data for users in the hidden-operator experiments also include a breakdown of the reasons for interferences (see Figure 9-1 for a discussion of each category). In this measure, a sentence may be counted more than once if it contains more than one type of interference. If, for example, a sentence contained a bracketed lexical extension and a substitution of concrete time for relative time then it would be counted only once in the "Int" column of the summary, but once under each of "lexical extension" and "implemented semantics" in the interference breakdown.

Instances of acceptance during the simulation that resulted in rejection for CHAMP are presented after the interference breakdown. In this table, "hidden-operator" error refers to occasions when CHAMP correctly rejects an utterance that the hidden-operator accepted. Thus, the number of sentences CHAMP should have understood but could not does *not* include that number.

To compute the number of new constructions per learning opportunity (as shown in Figures 9-2 through 9-5), we add the number of derived components considering competition relations to the number of sentences after which non-zero-level learning was added (Add(F)). To compute the number of unparsable sentences under CHAMP, we add the number unparsable under the simulation to the number in Dif. This means that the number of utterances seen by CHAMP during a particular session (Utt) is equal to the number of opportunities plus the number of unparsable utterances for CHAMP minus Add(F).

C.1.1. Data for User 1

Summary

S#	Utt	New[raw]	Add	Rem	Int	Dif	L0
1	9	7 [7]	0	0	3	0	1
2	15	4 [4]	0	0	7	0	11
3	16	3 [3]	0	0	2	1	5
4	20	1 [1]	0	0	6	0	3
5*	13	3 [3]	1	0	4	0	1
6	11	2 [2]	0	0	3	1	2
7	15	4 [4]	0	0	3	1	1
8	13	3 [3]	1(F)	0	6	2	1
9	15	4 [5]	0	0	4	0	0
	127	31 [32]	2	0	38	5	25

Interference Breakdown

Reason for interference	% utterances
unimplemented semantics	7/127 (6%)
relative->concrete time (5)	
change to concept definition (2)	
redundancy	3/127 (2%)
day of week (1)	
other (2)	
conjunction	16/127 (13%)
lexical extension problem	11/127 (8%)
forced rejections	1/127 (1%)
forced acceptances	3/127 (2%)

Differences Breakdown

Reason for difference	% utterances
hidden-operator error	2/127 (2%)
constructive inference	1/127 (1%)
single segment assumption	1/127 (1%)
inference capabilities	1/127 (1%)

New constructions/opportunity for learning & unparsable utterances

S#	CHAMP	hidden-op	un/C	un/h-o
1	7/9 (.78)	4/9 (.44)	0	0
2	4/14 (.29)	7/14 (.50)	1	1
3	3/12 (.25)	2/13 (.15)	4	3
4	1/19 (.05)	3/17 (.18)	1	1
5*	4/13 (.31)	3/11 (.27)	0	0
6	2/10 (.20)	5/11 (.45)	1	0
7	5/13 (.38)	2/14 (.14)	2	1
8	3/11 (.27)	2/9 (.22)	3	1
9	4/15 (.27)	0/14 (.00)	0	0
			12/127 (9%)	7/119 (6%)

C.1.2. Data for User 2

Summary

S#	Utt	New[raw]	Add	Rem	Int	Dif	L0
1	17	9 [10]	3(F)	0	9	3	3
2	17	6 [6]	3(F)	0	11	4	5
3	13	4 [6]	0	1	3	0	7
4	20	3 [3]	0	0	7	2	3
5	17	2 [2]	0	0	4	1	1
6	16	2 [3]	0	0	3	0	1
7*	16	11 [14]	0	0	5	1	1
8	12	1 [1]	0	0	4	0	4
9	16	2 [2]	0	0	3	0	1
	144	40 [47]	6	1	49	11	26

Interference Breakdown

Reason for interference	% utterances
unimplemented semantics	11/144 (8%)
relative->concrete time (9)	
change to concept definition (2)	
redundancy	12/144 (8%)
day of week (6)	
other (6)	
conjunction	8/144 (6%)
lexical extension problem	23/144 (16%)
forced rejections	2/144 (1%)

Differences Breakdown

Reason for difference	% utterances
hidden-operator error	3/144 (2%)
constructive inference	4/144 (3%)
single segment assumption	4/144 (3%)

New constructions/opportunity for learning & unparsable utterances

S#	CHAMP	hidden-op	un/C	un/h-o
1	12/13 (.92)	13/15 (.87)	5	2
2	6/11 (.55)	13/13 (1.00)	7	3
3	4/13 (.31)	3/13 (.23)	0	0
4	3/17 (.18)	4/18 (.22)	3	1
5	2/16 (.13)	2/17 (.12)	1	0
6	2/16 (.13)	2/16 (.13)	0	0
7*	11/14 (.79)	6/15 (.40)	2	1
8	1/12 (.08)	1/11 (.09)	0	0
9	2/16 (.13)	1/15 (.07)	0	0
			18/144 (13%)	7/140 (5%)

C.1.3. Data for User 3

Summary

S#	Utt	New[raw]	Add	Rem	Int	Dif	L0
1	15	9 [10]	0	0	6	5	1
2	16	8 [14]	1	0	5	6	7
3	16	11 [17]	1(F)	5	7	1	5
4	18	7 [12]	0	4	8	0	3
5	13	5 [8]	0	2	3	0	1
6	14	2 [3]	0	0	2	2	0
7*	16	7 [11]	0	1	4	2	0
8	15	6 [6]	0	0	6	2	0
9	16	2 [6]	0	4	1	0	1
	138	57 [87]	2	16	41	18	18

Interference Breakdown

Reason for interference	% utterances
unimplemented semantics	5/139 (4%)
relative->concrete time (4)	
change to concept definition (1)	
redundancy	3/138 (2%)
day of week (3)	
conjunction	7/138 (5%)
lexical extension problem	29/138 (21%)
forced acceptances	4/138 (3%)

Differences Breakdown

Reason for difference	% utterances
hidden-operator error	1/138 (1%)
constructive inference	8/138 (6%)
single segment assumption	9/138 (7%)

New constructions/opportunity for learning & unparsable utterances

S#	CHAMP	hidden-op	un/C	un/h-o
1	9/6 (1.50)	8/11 (.73)	9	4
2	8/9 (.89)	11/15 (.73)	7	1
3	12/13 (.92)	7/13 (.54)	4	3
4	7/16 (.44)	7/15 (.47)	2	2
5	5/13 (.38)	4/13 (.31)	0	0
6	2/11 (.18)	5/13 (.38)	3	1
7*	7/13 (.54)	4/15 (.27)	3	1
8	6/11 (.55)	3/10 (.30)	4	2
9	2/15 (.13)	1/15 (.07)	0	0
			32/138 (23%)	14/134 (11%)

C.1.4. Data for User 4

Summary

S#	Utt	New[raw]	Add	Rem	Int	Dif	L0
1	16	7 [9]	2(F)	0	13	3	2
2	20	11 [18]	1	2	20	1	7
3	12	4 [5]	0	3	12	0	5
4	16	0 [0]	0	0	16	2	4
5	12	2 [2]	0	0	12	0	3
6*	13	4 [4]	0	0	13	0	2
7	16	0 [0]	0	0	16	1	1
8	11	0 [0]	0	0	11	1	1
9	14	3 [3]	0	0	14	0	1
	130	31 [41]	3	5	130	8	26

Interference Breakdown

Reason for interference	% utterances
unimplemented semantics	17/130 (13%)
relative->concrete time (13)	
change to concept definition (4)	
redundancy	127/130 (98%)
day of week (127)	
other (1)	
lexical extension problem	21/130 (16%)
forced rejections	2/130 (2%)
forced acceptances	9/130 (7%)

Differences Breakdown

Reason for difference	% utterances
hidden-operator error	1/130 (1%)
constructive inference	4/130 (3%)
single segment assumption	3/130 (2%)

New constructions/opportunity for learning & unparsable utterances

S#	CHAMP	hidden-op	un/C	un/h-o
1	9/12 (.75)	11/14 (.79)	5	2
2	11/15 (.73)	10/16 (.63)	5	4
3	4/12 (.33)	1/12 (.08)	0	0
4	0/13 (.00)	1/15 (.07)	3	1
5	2/12 (.17)	0/12 (.00)	0	0
6*	4/13 (.31)	2/13 (.15)	0	0
7	0/15 (.00)	3/16 (.19)	1	0
8	0/9 (.00)	2/10 (.20)	2	1
9	3/14 (.21)	2/14 (.14)	0	0
			16/130 (12%)	8/130 (6%)

C.1.5. Data for User 5

Summary

S#	Utt	New[raw]	Add	Rem	Int	Dif	L0
1	11	5 [5]	0	0	5	4	0
2	14	5 [5]	0	0	7	4	3
3	13	10 [12]	2	0	8	0	8
	38	20 [22]	2	0	20	8	11

Interference Breakdown

Reason for interference	% utterances
redundancy	12/38 (32%)
day of week (10)	
other (2)	
lexical extension problem	11/38 (29%)
forced rejections	1/38 (3%)
forced acceptances	1/38 (3%)

Differences Breakdown

Reason for difference	% utterances
hidden-operator error	1/38 (3%)
constructive inference	3/38 (8%)
single segment assumption	3/38 (8%)
inference capabilities	1/38 (3%)

New constructions/opportunity for learning & unparsable utterances

S#	CHAMP	hidden-op	un/C	un/h-o
1	5/3 (1.67)	3/7 (.43)	8	4
2	5/9 (.56)	12/13 (.92)	5	1
3	11/13 (.85)	6/12 (.50)	1	1
			14/38 (37%)	6/38 (16%)

C.1.6. Data for User 7

Summary

S#	Utt	New[raw]	Add	Rem	Int	Dif	L0
1	19	10 [14]	3(F)	0	3	4	2
2	19	11 [11]	3(F)	0	8	4	4
3	14	8 [11]	0	4	3	0	9
4	16	3 [3]	0	2	3	1	2
5	12	3 [3]	0	2	2	1	1
	80	35 [42]	6	8	19	10	18

Interference Breakdown

Reason for interference	% utterances
implemented semantics	2/80 (3%)
relative->concrete time (2)	
redundancy	4/80 (5%)
day of week (2)	
other (2)	
lexical extension problem	13/80 (16%)
forced acceptances	3/80 (4%)

Differences Breakdown

Reason for difference	% utterances
hidden-operator error	2/80 (3%)
constructive inference	5/80 (6%)
single segment assumption	3/80 (4%)

New constructions/opportunity for learning & unparsable utterances

S#	CHAMP	hidden-op	un/C	un/h-o
1	13/13 (1.00)	15/14 (1.07)	9	5
2	14/12 (1.17)	12/15 (.80)	8	4
3	8/12 (.67)	3/12 (.25)	2	2
4	3/15 (.20)	3/16 (.19)	1	0
5	3/11 (.27)	3/12 (.25)	1	0
			21/80 (26%)	11/80 (14%)

C.2. Users in On-line Experiments

Some preprocessing of the hidden-operator data was necessary to recreate the conditions of the protocol as closely as possible given the differences between the simulation and the implementation. Since the preprocessing affected the values for some measures that are of general interest, those values were not computed for the hidden-operator data, but were computed for the on-line users.

The data for on-line users includes two summaries: a general summary and a summary of learning measures. Abbreviations relevant to the summaries are explained below:

S#	the session number. A star next to the number indicates that the supplementary instructions to work quickly were given at the beginning of that session.
Utt	the total number of utterances for the session.
Acc	the total number of utterances accepted for the session.
@0	the number of acceptances without error recovery.
@1	the number of acceptances at Deviation-level 1.
@2	the number of acceptanes at Deviation-level 2.
Rej(p/r)	the number of utterances rejected by Parse/Recovery.
Rej(res)	the number of utterances rejected during Resolution.
LE	the number of utterances in which lexical extensions were bracketed by the experimenter before the sentence was passed to CHAMP.
L0	the number of components derived at Deviation-level 0 (new instances, abbreviations, or permanent spelling corrections).
Raw	the number of grammatical components derived through error recovery and adaptation.
w/Comp	the raw values adjusted to take competition into consideration.
Boot	short for "bootstrapping," the number of accepted utterances for which all explanations required at least one learned constituent (the constituent may have been learned at Deviation-level 0).
Boot+	the number of instances of learning that relied on bootstrapping.

In the breakdown of rejections given after the summary, the row labelled "experimenter" indicates the number of rejections caused by the experimenter missing a lexical extension. The row labelled "user" indicates the number of rejections caused by the user when CHAMP correctly parsed the sentence but the user misunderstood the stimuli or changed her mind.

C.2.1. Data for User 9

Summary

S#	Utt	Acc	@0	@1	@2	Rej(p/r)	Rej(res)	LE	L0
1	32	17	8	4	5	8	7	7	3
2	31	22	15	5	2	8	1	1	9
3	20	20	16	2	2	0	0	3	5
4	26	19	19	0	0	4	3	1	4
5*	19	18	14	3	1	0	1	3	2
6	16	15	12	2	1	1	9	0	2
7	25	21	21	0	0	2	2	0	2
8	22	20	20	0	0	0	2	0	1
9	21	19	16	3	0	2	0	1	0
	212	171	141	19	11	25	16	16	28

Breakdown of Rejections

Reason for rejection	% utterances
deviance or resolution-time conflict	23/212 (11%)
deviance (unreachable)	5/212 (2%)
recovery-time constraints	6/212 (3%)
experimenter	3/212 (1%)
user	4/212 (2%)

Average number of states examined (states/utterances), by session

S#	Accept@0	Accept@1	Accept@2	Reject
1	10.75 (86/8)	79.00 (316/4)	201.80 (1009/5)	281.73 (4226/15)
2	33.40 (501/15)	202.40 (1012/5)	575.00 (1150/2)	237.11 (2134/9)
3	75.31 (1205/16)	150.00 (300/2)	1427.50 (2855/2)	0.00 (0/0)
4	77.89 (1480/19)	0.00 (0/0)	0.00 (0/0)	275.71 (1930/7)
5*	51.50 (721/14)	247.00 (741/3)	458.00 (458/1)	297.00 (297/1)
6	73.67 (884/12)	675.00 (1350/2)	456.00 (456/1)	1584.00 (1584/1)
7	70.14 (1473/21)	0.00 (0/0)	0.00 (0/0)	284.00 (1136/4)
8	57.95 (1159/20)	0.00 (0/0)	0.00 (0/0)	180.00 (360/2)
9	90.81 (1453/16)	665.67 (1997/3)	0.00 (0/0)	382.50 (765/2)

Average number of seconds in Parse/Recovery (seconds/utterances), by session

S#	Accept@0	Accept@1	Accept@2	Reject
1	1.09 (8.73/8)	4.82 (19.29/4)	9.33 (46.65/5)	15.39 (230.88/15)
2	1.90 (28.50/15)	9.16 (45.81/5)	17.84 (35.68/2)	17.42 (156.74/9)
3	3.79 (60.65/16)	7.35 (14.69/2)	261.84 (523.68/2)	0.00 (0.00/0)
4	4.37 (82.97/19)	0.00 (0.00/0)	0.00 (0.00/0)	26.92 (188.47/7)
5*	3.79 (52.99/14)	13.13 (39.40/3)	67.51 (67.51/1)	24.26 (24.26/1)
6	4.03 (48.36/12)	54.15 (108.29/2)	30.29 (30.29/1)	394.65 (394.65/1)
7	3.50 (73.47/21)	0.00 (0.00/0)	0.00 (0.00/0)	22.79 (91.14/4)
8	3.07 (61.40/20)	0.00 (0.00/0)	0.00 (0.00/0)	14.19 (28.37/2)
9	3.11 (49.77/16)	46.51 (139.53/3)	0.00 (0.00/0)	17.13 (34.29/2)

Average number of roots (roots/utterances), by session

S#	After Parse/Recovery	After Resolution
1	0.96 (23/24)	1.06 (18/17)
2	2.04 (47/23)	1.23 (27/22)
3	2.35 (47/20)	1.50 (30/20)
4	1.77 (39/22)	1.32 (25/19)
5*	2.32 (44/19)	1.33 (24/18)
6	3.93 (59/15)	1.93 (29/15)
7	2.13 (49/23)	1.57 (33/21)
8	1.95 (43/22)	1.40 (28/20)
9	2.74 (52/19)	1.58 (30/19)

Worst case Parse/Recovery: Session 4, after P/R=60 roots, after Resolution=0 roots
Worst case Resolution: Session 6, after P/R=20 roots, after Resolution=12 roots

Summary of learning

S#	Raw	w/Comp	Rem	Boot	Boot+
1	16/17 (.94)	14/17 (.82)	2	6	4
2	14/22 (.64)	10/22 (.45)	3	19	4
3	7/20 (.35)	6/20 (.30)	1	19	4
4	0/19 (.00)	0/19 (.00)	0	18	0
5*	5/18 (.28)	5/18 (.28)	0	16	4
6	6/15 (.40)	4/15 (.27)	0	12	3
7	0/21 (.00)	0/21 (.00)	0	20	0
8	0/20 (.00)	0/20 (.00)	0	19	0
9	5/19 (.26)	3/19 (.16)	0	17	3

% of accepted utterances understood via previous learning: 146/171 (85%)
% of learning episodes that relied on bootstrapping: 24/30 (80%)

Response-time synopsis

Utterances parsed in under 10 seconds: 164/212 (77%)
Utterances parsed between 10 and 60 seconds: 40/212 (19%)
Utterances requiring more than 60 seconds: 8/212 (4%)
Actual values > 60: (78.78, 84.78, 394.65, 67.51, 115.10, 388.24, 135.44, 66.49)

C.2.2. Data for User 10

Summary

S#	Utt	Acc	@0	@1	@2	Rej(p/r)	Rej(res)	LE	L0
1	30	16	6	5	5	8	6	8	6
2	15	13	10	2	1	1	1	3	9
3	14	12	9	2	1	0	2	3	6
4	20	20	18	2	0	0	0	0	3
5	18	16	16	0	0	2	0	1	1
6*	22	15	14	1	0	6	1	6	7
7	24	22	19	3	0	1	1	4	1
8	15	13	12	0	1	2	0	1	3
9	15	13	13	0	0	2	0	0	0
	173	140	117	15	8	22	11	26	36

Breakdown of Rejections

Reason for rejection	% utterances
deviance or resolution-time conflict	8/173 (5%)
deviance (unreachable)	9/173 (5%)
maximal subsequence	3/173 (17%)
single segment	3/173 (17%)
inference	3/173 (17%)
experimenter	1/173 (1%)
user	6/173 (3%)

Average number of states examined (states/utterances), by session

S#	Accept@0	Accept@1	Accept@2	Reject
1	30.50 (183/6)	92.80 (464/5)	197.00 (985/5)	230.93 (3233/14)
2	55.40 (554/10)	224.00 (448/2)	161.00 (161/1)	67.50 (135/2)
3	52.56 (473/9)	148.50 (297/2)	1131.00 (1131/1)	38.50 (77/2)
4	48.33 (870/18)	320.00 (640/2)	0.00 (0/0)	0.00 (0/0)
5	32.06 (513/16)	0.00 (0/0)	0.00 (0/0)	185.00 (370/2)
6*	31.21 (437/14)	10.00 (10/1)	0.00 (0/0)	93.14 (652/7)
7	29.68 (564/19)	107.33 (622/3)	0.00 (0/0)	169.00 (338/2)
8	25.83 (310/12)	0.00 (0/0)	3295.00 (3295/1)	332.00 (664/2)
9	32.31 (420/13)	0.00 (0/0)	0.00 (0/0)	203.50 (407/2)

Average number of seconds in Parse/Recovery (seconds/utterances), by session

S#	Accept@0	Accept@1	Accept@2	Reject
1	1.35 (8.08/6)	4.42 (22.09/5)	8.65 (43.23/5)	11.92 (166.88/14)
2	2.61 (26.05/10)	7.63 (15.25/2)	8.31 (8.31/1)	5.74 (11.49/2)
3	3.40 (30.62/9)	11.06 (22.13/2)	53.87 (53.87/1)	6.41 (12.83/2)
4	2.27 (40.91/18)	14.70 (29.41/2)	0.00 (0.00/0)	0.00 (0.00/0)
5	1.79 (27.15/16)	0.00 (0.00/0)	0.00 (0.00/0)	18.51 (37.03/2)
6*	1.29 (25.88/14)	1.29 (1.29/1)	0.00 (0.00/0)	6.95 (48.62/7)
7	1.47 (27.99/19)	8.54 (25.61/3)	0.00 (0.00/0)	6.58 (13.16/2)
8	1.82 (21.79/12)	0.00 (0.00/0)	410.32 (410.32/1)	11.95 (23.89/2)
9	1.56 (20.22/13)	0.00 (0.00/0)	0.00 (0.00/0)	10.50 (21.00/2)

Average number of roots (roots/utterances), by session

S#	After Parse/Recovery	After Resolution
1	1.31 (29/22)	1.00 (16/16)
2	2.14 (30/14)	1.15 (15/13)
3	4.50 (63/14)	1.67 (20/12)
4	2.15 (43/20)	1.25 (25/20)
5	1.13 (18/16)	1.00 (16/16)
6*	1.75 (28/16)	1.13 (17/15)
7	1.61 (37/23)	1.18 (26/22)
8	6.00 (78/13)	1.23 (16/13)
9	1.46 (19/13)	1.00 (13/13)

Worst case Parse/Recovery: Session 8, after P/R=63 roots, after Resolution=3 roots
Worst case Resolution: Session 3, after P/R=11 roots, after Resolution=6 roots

Summary of learning

S#	Raw	w/Comp	Rem	Boot	Boot+
1	15/16 (.94)	15/16 (.94)	0	6	3
2	5/13 (.38)	4/13 (.31)	0	11	2
3	6/12 (.50)	4/12 (.33)	1	12	3
4	2/20 (.10)	2/20 (.10)	0	20	2
5	0/16 (.00)	0/16 (.00)	0	15	0
6*	1/15 (.07)	1/16 (.06)	0	15	1
7	3/22 (.14)	3/22 (.14)	0	20	3
8	4/13 (.31)	2/13 (.15)	0	13	1
9	0/13 (.00)	0/13 (.00)	0	11	0

% of accepted utterances understood via previous learning: 123/140 (88%)
% of learning episodes that relied on bootstrapping: 15/23 (65%)

Response-time synopsis

Utterances parsed in under 10 seconds: 157/173 (91%)
Uutterances parsed between 10 and 60 seconds: 15/173 (9%)
Utterances requiring more than 60 seconds: 1/173 (<1%)
Actual values > 60: (410.317)

References

1. Anderson, J. R., "Induction of Augmented Transition Networks," *Cognitive Science,* 1977.

2. Anderson, J. R., *The Architecture of Cognition,* Harvard University Press, Cambridge, MA, 1983.

3. Ballard, B. W., Lusth, J. C., Tinkham, N. L., "LDC-1: A Transportable, Knowledge-Based Natural Language Processor for Office Environments," *ACM Transactions on Office Information Systems,* Vol. 2, No. 1, 1984.

4. Berwick, R. C., "Learning Word Meanings from Examples," *Proceedings of the Eighth International Joint Conference on Artificial Intelligence,* August 1983.

5. Berwick, R., *The Acquisition of Syntactic Knowledge,* MIT Press, Cambridge, MA, 1985.

6. Berwick, R.C., and Pilato, S., "Learning Syntax by Automata Induction," *Machine Learning,* Vol. 2, 1987.

7. Carbonell, J. G., "Towards a Self-Extending Parser," *17th Annual Meeting of the Association for Computational Linguistics,* 1979.

8. Carbonell, J. G., *Subjective Understanding: Computer Models of Belief Systems,* PhD dissertation, Yale University, 1979.

9. Carbonell, J. G. and Hayes, P. J., "Recovery Strategies for Parsing Extragrammatical Language," *American Journal of Computational Linguisitics,* Vol. 9, No. 3-4, 1983.

10. Chomsky, N., *Aspects of the Theory of Syntax,* MIT Press, Cambridge, MA, 1965.

11. Cobourn, T. F., "An Evaluation of Cleopatra, a Natural Language Interface for CAD," Master's thesis, Carnegie Mellon University, December 1986.

12. Davidson, J., and Kaplan S. J., "Parsing in the Absence of a Complete Lexicon," *18th Annual Meeting of the Association for Computational Linguistics,* 1980.

13. Durham, I., Lamb, D. A., and Saxe, J. B., "Spelling Correction in User Interfaces," *Communications of the ACM,* Vol. 26, 1983.

14. Fain, J., Carbonell, J. G., Hayes, P. J., and Minton, S. N., "MULTIPAR: A Robust Entity-Oriented Parser," *Proceedings of the Seventh Annual Conference of The Cognitive Science Society,* 1985.

15. Fillmore, C., "The Case for Case," in *Universals in Linguistic Theory,* Bach and Harms, eds., Holt, Rinehart, and Winston, 1968.

16. Fink, P. K., and Biermann, A. W., "The Correction of Ill-Formed Input using History-Based Expectation with Applications to Speech Understanding," *Computational Linguistics,* Vol. 12, No. 1, 1986.

17. Frederking, R. E., *Natural Language Dialogue in an Integrated Computational Model,* PhD dissertation, Carnegie Mellon University, 1986.

18. Furnas, G. W., "Statistical Semantics: Analysis of the Potential Performance of Key-Word Information Systems," *The Bell System Technical Journal,* Vol. 62, No. 6, 1983.

19. Good, M. D., Whiteside, J. A., Wixon, D.R., and Jones, J.J., "Building a User-Derived Interface," *Communications of the ACM,* Vol. 27, No. 10, 1984.

20. Granger, R. H., "FOUL-UP: A Program That Figures Out Meanings of Words from Context," *Proceedings of the Fifth International Joint Conference on Artificial Intelligence,* 1977.

21. Granger, R. H., "The NOMAD System: Expectation-Based Detection and Correction of Errors during Understanding of Syntactically and Semantically Ill-Formed Text," *American Journal of Computational Linguistics,* Vol. 9, 1983.

22. Grosz, B., "TEAM: a Transportable Natural Language Interface System," *Conference on Applied Natural Language Processing,* 1983.

23. Harris, L. R., "A System for Primitive Natural Language Acquisition," *International Journal for Man-Machine Studies,* 1977.

24. Harris, L. R., "Experience with ROBOT in 12 Commercial Natural Language Database Query Applications," *Proceedings of the Sixth International Joint Conference on Artificial Intelligence,* 1979.

25. Harris, L. R., "Experience with INTELLECT," *The AI Magazine,* Vol. V, No. 2, 1984.

26. Hass, N., and Hendrix, G. G., "Learning by Being Told: Acquiring Knowledge for Information Management," in *Machine Learning, An Artificial Intelligence Approach,* Michalski, R. S., Carbonell, J. G., and Mitchell, T. M., eds., Tioga Publishing Company, 1983.

27. Hauptmann, A. G., Young, S. R., and Ward, W. H., "Using Dialog-Level Knowledge Sources to Improve Speech Recognition," *Proceedings of the Seventh National Conference on Artificial Intelligence, American Association for Artificial Intelligence,* 1988.

28. Hendrix, G. G., Sacerdoti, E. D., Sagalowicz, D., and Slocum, J., "Developing a Natural Language Interface to Complex Data," *ACM Transactions on Database Systems,* Vol. 3, No. 2, 1978.

29. Kaplan, S. J., *Cooperative Responses from a Portable Natural Language Data Base Query System,* PhD dissertation, University of Pennsylvania, 1979.

30. Kelley, J. F., *Natural Language and Computers: Six Empirical Steps for Writing an Easy-to-Use Computer Application,* PhD dissertation, Johns Hopkins University, 1983.

31. Kelley, J. F., "An Iterative Design Methodology for User-Friendly Natural Language Office Information Applications," *ACM Transactions on Office Information Systems,* Vol. 2, No. 1, 1984.

32. Kelley, J. F., "CAL—A Natural Language program developed with the OZ Paradigm: Implications for Supercomputing Systems," *Proceedings First International Conference on Supercomputing Systems,* 1985.

33. Kelley, J. F., and Chapanis, A., "How professional persons keep their calendars: Implications for computerization," *Journal of Occupational Psychology,* No. 55, 1982.

34. Keyes, J., "Wall Street Speaks English," *AI Expert,* July 1988.

35. Kwasny, S. C., and Sondheimer, N. K., "Ungrammaticality and Extra-grammaticality in Natural Language Understanding Systems," *17th Annual Meeting of the Association for Computational Linguistics,* 1979.

36. Langley, P., "A Model of Early Syntactic Development," *20th Annual Meeting of the Association for Computational Linguistics,* 1982.

37. Lehman, J. F., and Carbonell, J. G., "Learning the User's Language, A Step Towards Automated Creation of User Models," in *User Modelling in Dialog Systems,* Wahlster, W., and Kobsa, A., eds., Springer-Verlag, 1989.

38. Malhotra, A., "Knowledge-Based English Language Systems for Management Support: An Analysis of Requirements," *Proceedings of the Fourth International Joint Conference on Artificial Intelligence,* 1975.

39. Malhotra, A., "Design Criteria for a Knowledge-Based English Language System for Management: An Experimental Analysis," *Technical Report MAC TR-146, Massachusetts Institute of Technology,* 1975.

40. Michaelis, P. R., Chapanis, A., Weeks, G., and Kelly, M. J., "Word Usage in Interactive Dialog with Restricted and Unrestricted Vocabularies," *IEEE Transactions on Professional Communication,* Vol. PC-20, No. 4, 1977.

41. Miller, P. L., "An Adaptive Natural Language System that Listens, Asks and Learns," *Proceedings of the Fourth International Joint Conference on Artificial Intelligence,* 1975.

42. Minton, S. N., Hayes, P. J., Fain, J. E., "Controlling Search in Flexible Parsing," *Proceedings of the Ninth International Joint Conference on Artificial Intelligence,* 1985.

43. Mitchell, T., Keller, R., Kedar-Cabelli, S., "Explanation-Based Generalization: A Unifying View," *Machine Learning,* Vol. 1, No. 1, 1986.

44. Montgomery, C. A., "Is Natural Language an Unnatural Query Language?," *ACM Annual Conference,* 1972.

45. Neal, J.G., and Shapiro, S. C., "Talk About Language !?," *Technical Report, S.U.N.Y. at Stony Brook,* 1985.

46. Petrick, S. R., "On Natural Language Based Computer Systems," *IBM Journal of Research and Development*, Vol. 20, 1976.

47. Pinker, S., *Language Learnability and Language Development*, Harvard University Press, Cambridge, MA, 1984.

48. Rich, E., "Natural Language Interfaces," *Computer*, Vol. 17, No. 9, 1984.

49. Salveter, S. C., "On the Existence of Primitive Meaning Units," *18th Annual Meeting of the Association for Computational Linguistics*, 1980.

50. Schank R. C., *Conceptual Information Processing*, North-Holland, Amsterdam, 1975.

51. Schank, R., and Abelson, R., *Scripts, Plans, Goals, and Understanding*, Lawrence Earlbaum Associates, Inc., Hillsdale, NJ, 1977.

52. Selfridge, M., "A Computer Model of Child Language Acquisition," *Proceedings of the Seventh International Joint Conference on Artificial Intelligence*, 1981.

53. Selfridge, M., "A Computer Model of Child Language Learning," *Artificial Intelligence*, Vol. 29, 1986.

54. Selfridge, M., "Integrated Processing Produces Robust Understanding," *Computational Linguistics*, Vol. 12, No. 2, 86.

55. Shneiderman, B., "A Note on Human Factors Issues of Natural Language Interaction with Database Systems," *Information Systems*, Vol. 6, No. 2, 1981.

56. Siklossy, L., "Natural Language Learning by Computer," in *Representation and Meaning: Experiments with Information Processing Systems*, Simon, H. A., and Siklossy, L., eds., Prentice-Hall, Inc., 1972.

57. Slator, B. M., Anderson, M. P., and Conley, W., "Pygmalion at the Interface," *Communications of the ACM*, Vol. 29, No. 7, 1986.

58. Tennant, H., *Evaluation of Natural Language Processors*, PhD dissertation, University of Illinois, 1980.

59. Thompson, B. H., and Thompson, F. B., "Introducing ASK: A Simple Knowledgeable System," *Conference on Applied Natural Language Processing*, 1983.

60. Waltz, D. L., "An English Language Question Answering System for a Large Relational Database," *Communications of the ACM*, Vol. 21, 1978.

61. Watt, W. C., "Habitability," *American Documentation*, July 1968.

62. Weischedel, R. M., and Black, J. E., "Responding Intelligently to Unparsable Inputs," *American Journal of Computational Linguisitics*, Vol. 6, No. 2, 1980.

63. Wilensky, R., "Talking to UNIX in English: An Overview of an On-line Consultant," *The AI Magazine*, Vol. V, No. 1, 1984.

64. Woods, W. A., "Transition Network Grammars for Natural Language Analysis," *Communications of the ACM*, Vol. 13, No. 10, 70.

65. Young, S. R., Hauptmann, A. G., Ward, W. H., Smith, E. T., and Werner, P., "High Level Knowledge Sources in Usable Speech Recognition Systems," *Communications of the ACM*, Vol. 32, No. 2, 1989.

66. Young, S. R., and Ward, W. H., "Towards Habitable Systems: Use of World Knowledge to Dynamically Constrain Speech Recognition," *Second Symposium on Advanced Man-Machine Interfaces through Spoken Language*, 1988.

67. Zernik, U. and Dyer, M. G., "Failure-Driven Acquisition of Figurative Phrases by Second Language Speakers," *Proceedings of the Seventh Annual Conference of The Cognitive Science Society*, 1985.

68. Zernik, U., *Strategies in Language Acquisition: Learning Phrases from Examples in Context*, PhD dissertation, University of California, Los Angeles, 1987.

69. Zernik, U., "Learning Idioms -- With and Without Explanation," *Proceedings of the Tenth International Joint Conference on Artificial Intelligence*, 1987.

70. Zernik, U., "Language Acquisition: Learning a Hierarchy of Phrases," *Proceedings of the Tenth International Joint Conference on Artificial Intelligence*, 1987.