Growing Semantic Grammars

by

Marsal Gavaldà i Camps

B.S., Universitat Politècnica de Catalunya, 1994

M.S., Carnegie Mellon University, 1996

Submitted to the Graduate Faculty of

the Language Technologies Insititue in partial fulfillment

of the requirements for the degree of

Doctor of Philosophy

Carnegie Mellon University

2000

Carnegie Mellon University

Ph.D. Program in Language and Information Technologies

This dissertation was presented

by

Marsal Gavaldà i Camps

It was defended on

4 August 2000

and approved by

Dr. Alex Waibel, Chair

Dr. Jaime Carbonell

Dr. Jill Lehman

Dr. Stephanie Seneff (M.I.T.)

Dr. Wayne Ward (University of Colorado)

Dedicated to
Maria Teresa Camps i Bernabé,
Domènec Gavaldà i Aran,
and 翁佳行

Foreword

I first arrived in Pittsburgh during the unusually cold winter of 1994, as a "visiting scholar"[1] to Carnegie Mellon's Center of Machine Translation from the Universität Fridericiana zu Karlsruhe, in Karlsruhe, Germany, (where, incidentally, I was an exchange student from the Universitat Politècnica de Catalunya, in Barcelona, Catalonia) and I stayed the first night at the house of CMT staff member Noah Coccaro, with the verbal assurance that the heating was running full steam. Well —guess what?— the heating in my room broke down during that night.

From such an auspicious beginning, it was hard to imagine that I would eventually spend over six years in Pittsburgh, first as a master's student in the Computational Linguistics program at CMU's Philosopy Department, and later as a Ph.D. student in the Language and Information Technologies program at the newly created Language Technologies Institute.

The usual ups and downs of graduate student life notwithstanding, I've had a great time at CMU and I fondly remember past and present classmates, coworkers and professors. I particularly recall one autumn day in 1995, when my fellow student Laura Mayfield and I went to see our common advisor, Prof. Prof. Dr. Alexander H. Waibel, with the big news. We were sitting down together on the floor of the 4600 corridor in Wean Hall waiting for him. Eventually, Alex opened the door, we stood up, and I said it: "Alex, we have something to tell you: Laura and I are getting married." And, indeed, that Christmas we got married in Asia and I moved into her apartment. "But not to each other." As it turns out, we both got married to our respective spouses, Laura in Japan and I in China, and my wife and I took over Laura's lease.

Will a computer[2] ever be capable of "understanding" this little anecdote? (And how about "finding it amusing"?) We do not know for sure, much of it depends on what passes as "understanding," but making machines behave as though they understood language is the underlying theme of this dissertation.

Since early childhood I have been fascinated by language, this symbolic net that our minds cast upon the world (including themselves!) and that somehow mediates most of our cogni-

[1] Official title in quotes because, at that time, I was still an undergrad.
[2] A computer that is not a *Homo sapiens sapiens*, to be precise.

tive processes. I remember repeating the pronunciation of foreign words (as sweet as candy), drawing evolutionary language trees (is there a single root?), and eventually going through consecutive periods of very intensive language learning (English, German, Japanese, Chinese, all wondrous escapes from the oppressive presence of the Spanish language in my native country of Catalonia).

And when my adventurous father brought home a Hewlett Packard HP-85 digital computer, in the autumn of 1980, I immediately sensed that this was an extraordinary object: it was *programmable*, its monochrome, 256×192-pixel screen, could draw anything imaginable and then some![3]

Thus, it is not surprising that as soon as I heard of natural language processing and computational linguistics I felt a familiar attachment and an eagerness to pursue this field.

And not surprisingly either, this dissertation is dedicated to my parents: Maria Teresa Camps i Bernabé and Domènec Gavaldà i Aran, and to my wife: 翁佳行 (Jiaxing Weng), for, among the many reasons too numerous and intimate to detail, their unwavering support and encouragement to complete it.

[3]And yet, at the same time, I realized that the number of possible distinct images was *finite*: $2^{256 \cdot 192}$, which is about $1.68 \cdot 10^{14,796}$.

Abstract

One of the greatest difficulties in the successful deployment of natural language understanding (NLU) systems lies in the inherent richness of human language: it is impossible to capture a priori all the different ways in which people may express a particular idea. This dissertation demonstrates, through the explicit construction and real-life testing of the GSG computer system, that this problem can be solved by enabling the NLU system to automatically learn new semantic mappings, without requiring either large amounts of hand-coded world or task-specific knowledge, or technical expertise in the users of the end application.

When GSG does not understand what the user says, it makes educated guesses, poses confirmation and clarification questions, and remembers the meaning of the new words and constructions by formulating and generalizing new rules and merging them with the existing grammar. GSG incorporates external knowledge sources (semantic grammar, part-of-speech tagger, syntactic grammar, end-application constraints), constructs internal knowledge sources (Ontology, Parsebank, Hypotactical and Paratactical Models) and combines learning strategies (All-top parsing, Anchor Mother Prediction, Daughter Argument Selection, Vertical and Horizontal Generalization) into a coherent, mixed-initiative conversation with the end-user, as an epiphenomenon of which, the original semantic grammar, written in a standard formalism such as JSGF, is judiciously and seamlessly extended.

Thus, GSG provides an extremely robust NLU interface, and, at the same time, significantly reduces grammar development time, because the original grammar, while complete with respect to the semantic representation of the domain at hand, need only cover a small portion of the surface variability, for it will be automatically extended as the users interact with the system.

GSG has been tested in two different applications with excellent results: correct rules are acquired and the resulting grammar increases its semantic accurarcy on independent test sets between 20 and 32 absoulte percentage points, which amounts to an error reduction factor between 1.35 and 16.51.

Moreover, users feel empowered by the ability to teach their language patterns to the system.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

The dream of communicating with computers via natural language (the best medium for the exchange of ideas that we know of) is beginning to be fulfilled: Speech recognition and natural language understanding by computers are no longer science fiction. In fact, the field of language technologies is experiencing a blossoming of research prototypes (e.g, the C-STAR speech translation project [Waibel, 1996]), marketable consumer products (e.g., dictation systems) and deployed industrial "solutions" (e.g., call centers) which, albeit severely limited, begin to be truly useful.

It is within this context that this dissertation becomes relevant: As natural language understanding systems transition from research labs into people's everyday life, and as they evolve from the system-directed interactions of interactive voice response systems based on isolated-word recognizers and fixed-menu navigation, to the mixed-initiative dialogues carried out in spoken dialogue systems based on large-vocabulary continuous speech recognizers and flexible dialogue managers (see, for example, [Allen et al., 1996; Denecke, 1997; Walker et al., 1998; Rudnicky et al., 1999; Zue et al., 2000]), the overall *experiential* quality of the human-computer interaction becomes increasingly important. That is, beyond the obvious factors of speech recognition accuracy and speech synthesis naturalness, the most critical challenge is that of providing a conversational interaction that feels natural to human users (cf. [Glass, 1999]). And this challenge, I believe, can only be met by building systems that possess some degree of linguistic, reasoning, and learning abilities.

The thesis of this dissertation is that a first step in this direction can be achieved by enabling natural language understanding (NLU) systems to learn the meaning of new words and constructions, and it is validated through the description of GSG, an NLU system that is able to dynamically extend its linguistic knowledge through simple, natural language only interactions with non-expert users. On a purely on-need basis, i.e., when the system does not understand what the user says, GSG makes educated guesses, poses confirmation and clarification questions, and learns new semantic mappings from the answers given by the users, as well as from other linguistic information that they may volunteer.

Granted, there is no magic wand that can solve the complex issue of natural language understanding: Mapping a string of words into some form of semantic representation is as difficult as painting the thoughts the reader is having at this moment. However, this thesis represents a step in the direction of facilitating human-computer interaction through natural language by demonstrating that computers can be programmed to learn natural language in a way akin to how humans do it, i.e., by asking clarification questions, making generalizations, and detecting conflicts with previous knowledge.[1] As we shall see, GSG incorporates external knowledge sources (semantic grammar, part-of-speech tagger, syntactic grammar, end-application constraints), constructs internal knowledge sources (Ontology, Parsebank, Hypotactical and Paratactical Models) and combines learning strategies (All-top parsing, Anchor Mother Prediction, Daughter Argument Selection, Vertical and Horizontal Generalization) into a coherent, mixed-initiative conversation with the end-user, as an epiphenomenon of which, the original semantic grammar is judiciously and seamlessly extended.

1.1 Reader's Guide

This document is organized into eight chapters and five appendices. The body of the text provides an explanation of the problem, a detailed description of the proposed solution in the form of the GSG[2] system, and an account and analysis of the experiments conducted to validate it; the appendices provide all the relevant data, including grammars, user utterances, and learning episodes.

In more detail, the following **Chapter 2: Dissertation Overview**, states the general problem of extragrammaticality and presents a condensed version of the solution proposed in this dissertation; **Chapter 3: Related Work** surveys the rich substrate of research upon which GSG has grown; **Chapter 4: Philosophy and Modus Operandi of** GSG restates the motivation for and the overall architecture of GSG; **Chapter 5: The** SOUP **Parser** describes SOUP, the parsing engine that lies at the heart of GSG; **Chapter 6: The** GSG **Learning System** explains in detail the algorithms for interactive parse construction and rule management that form the core of the thesis; **Chapter 7: Experiments and Results** describes the different experiments conducted and presents their results; finally,

[1] Although no claim is made that GSG's algorithms resemble the ones humans employ.
[2] Abbreviation of GROWING SEMANTIC GRAMMARS.

	C++		Java		Total	
	Lines	Classes	Lines	Classes	Lines	Classes
Soup	33,532	36	5,798	20	39,330	56
Gsg	24,488	15	25,550	75	50,038	90
Total	58,020	51	31,348	95	**89,368**	**146**

Table 1.1: Number of lines of code and classes for the implementation of Soup and Gsg. (Does not include automatically generated C++ code to interface with Java).

Chapter 8: Conclusion, summarizes the main contributions of this dissertation and sketches future directions for research.

Note that Chapters 3 (Related Work) and 5 (The Soup Parser) may be skipped without loss of continuity.

As for the appendices, **Appendix A: Small Grammar to Illustrate Soup's Heuristics** lists a grammar for illustrative purposes; **Appendix B: Syntactic Grammar** presents the syntactic knowledge of English given to the system, namely, the part-of-speech tag set, an explanation of the top-level categories of the English syntactic grammar, and the syntactic grammar itself; **Appendix C: Semantic Grammars** records the semantic knowledge given to the system, namely, the Metagrammar, for all applications, followed by the kernel grammars for the E-Mail Task and the Musicbox Task; **Appendix D: Utterances from User Sessions** lists the utterances that occurred in the experiments, both for the E-Mail Task and the Musicbox Task; finally, **Appendix E: Learning Episodes** presents in full detail all the learning episodes (and cancelations) that took place during the user sessions.

1.2 A Note on the Implementation

The entire computer system described in this dissertation has been implemented by the author from the ground up using the C++ and Java programming languages. Since the coding of an initial version of the Soup parser, began in the summer of 1996 until now (for years later), Soup itself has been rewritten twice and the final code, including the learning components of Gsg, has grown to encompass about 90,000 lines of code and 146 classes[3] (see Table 1.1).

[3]In the software-development, object-oriented sense of a class being a data structure and its functionality.

1.3 Grammar Format

For consistency's sake, all grammars and grammar-related examples in this dissertation are written according to the Java Speech Grammar Format (JSGF), which is part of the Java Speech Application Development Interface [JSAPI, 1998]. For a complete reference, see [JSGF, 1998], but, as a way of summary, JSGF grammars are context-free grammars where:

- A grammar is a set of rules.
- A rule is composed of a left-hand side or nonterminal and a right-hand side or rule body, following the pattern *LHS* = *RHS*;, e.g. <greeting> = hello;.
- Nonterminals that are starting symbols of the grammar (i.e., top-level NTs) are marked by the keyword public in front of their definition.
- Nonterminals are enclosed in angle brackets, e.g. <greeting>.
- Right-hand sides (RHSs) can mix terminals and nonterminals, e.g. how about <time>.
- Rule alternatives are separated by '|', e.g. <greeting> = hello | how are you;.
- Optionality is expressed by enclosing the optional subRHS in square brackets, e.g. how are you [today] matches *how are you* and *how are you today*.
- Repeatability is expressed by a '+' immediately after the repeatable subRHS, e.g. bye+ matches *bye*, *bye bye*, *bye bye bye*, etc.
- Optionality and repeatability (i.e., Kleene star) can be compactly expressed by a '*' immediately after the optional and repeatable subRHS, e.g. good bye* matches *good*, *good bye*, *good bye bye*, etc.
- Brackets can be used to delimit scope, e.g. how are you [today | this (morning | afteroon | evening) | tonight] matches *how are you*, *how are you today*, *how are you this morning*, *how are you this afternoon*, *how are you this evening*, and *how are you tonight*.

1.4 Abbreviations

Most of the abbreviations used in this dissertation are expanded below.

NLU:	natural language understanding
CFG:	context-free grammar
NT:	grammar nonterminal
LHS:	left-hand side (same as a nonterminal in a context-free grammar)
RHS:	right-hand side, rule body
subRHS:	part of a right-hand side
T:	grammar terminal, word

Chapter 2

Dissertation Overview

This chapter presents the problem that motivates this dissertation, possible approaches to tackle it, and a high-level description of both the approach taken and its implementation through the GSG system. A concrete example and a summary of results are provided, and pointers to the more detailed explanation that constitutes the rest of the document are included throughout.

2.1 Motivation

A fundamental step in natural language understanding[1] is the transformation of a sequence of words (for example, the result of the decoding of an utterance by a speech recognizer) into a meaning representation that can be used to reason about the intention of the speaker and may lead to the execution of an action. An example would be an NLU system that manages your electronic mail. In that case, one would want to transform, say, *read again the last message from alice* into a logical representation, such as the one in Figure 2.1, which can be processed and executed by an e-mail client.[2]

2.1.1 Defining Semantic Mappings

The most straightforward way of defining such a transformation is via a semantic grammar, so called because its nonterminals represent semantic concepts (such as `<senderName>`). A possible semantic grammar for the above example is shown in Figure 2.2.

[1]Natural language understanding can be described as the programming of computers so that they behave as though they understood natural language (see, for example, [Allen, 1995; Jurafsky and Martin, 2000] for textbook introductions to the field). Note that this definition sidesteps the philosophical issue of whether a (non-human) computer can ever "really understand" natural language, and it does so by adopting Alan Turing's realization that, for all practical purposes, *behavior* is what really counts (as in the ultimate artificial intelligence test known as the Turing Test [Turing, 1950; Dennett, 1984], which will be passed when one can no longer distinguish whether one's interlocutor is a human or a machine).

[2]Of course, the e-mail address of *alice* will still have to be looked up in an address book.

$$\begin{bmatrix} \texttt{readMessage} & \\ & \begin{bmatrix} \texttt{index} & \textit{last} \\ \texttt{senderName} & \textit{alice} \end{bmatrix} \end{bmatrix}$$

Figure 2.1: Feature structure representing the meaning of *read again the last message from alice*. A feature structure can be described as a set of attribute–value pairs (e.g., attribute `index`, value *last*), where values can themselves be feature structures (e.g., the complex value of attribute `readMessage`).

```
public <readMessage> = <_read> [again] [the] [<index>] <_message> <_sender>;
       <_read>       = read | speak;
       <index>       = first | last;
       <_message>    = message | messages;
       <_sender>     = (from | sent by) <senderName>;
       <senderName>  = alice | bob;
```

Figure 2.2: Example of a semantic grammar to parse *read again the last message from alice*. As explained in §1.3, the grammar formalism follows JSGF, where top-level rules (i.e., starting symbols of the grammar) are denoted by the keyword `public`, grammar nonterminals are surrounded by '`<`' and '`>`,' alternatives separated by '`|`,' and optional constituents are enclosed by '`[`' and '`]`.' Additionally, the convention of making the names of auxiliary nonterminals begin with '_' is used, as in `<_message>`.

Semantic grammars are of course not the only way to obtain the meaning of a sentence. For example, a common approach is to use a syntactic grammar to identify thematic rôles, such as *agent*, *theme*, *instrument*, *benefactee*, *experiencer*, etc. (see for example [Napoli, 1993]), and then use this information to build a semantic representation of the sentence. However, this process still requires domain-dependent semantics if one is to arrive at meaning representations (such as the one in Figure 2.1) that are suitable for further processing and that lead to the actual execution of commands in the back-end application.

Therefore, it can be argued that the level of difficulty of describing any semantic mapping is comparable to that of defining a semantic grammar. And, throughout this dissertation, semantic grammars are the semantic mapping of choice.

Going back to the example above, we see that the parsing[3] of *read again the last message from alice* using the semantic grammmar in Figure 2.2 yields the parse tree depicted in Figure 2.3. Furthermore, this parse tree can be easily transformed[4] into the desired feature structure in Figure 2.1.

[3] Structural analysis of a sentence according to a grammar.

[4] Deterministic, context-free conversion of a parse tree into a feature structure by mapping the parse tree nodes that correspond to principal nonterminals (i.e., those nonterminals whose names do not begin with an underscore '_') into feature structure attributes, whose values are, recursively, feature structures extracted from the parse subtrees, or primitive values when the parse tree leaves are reached.

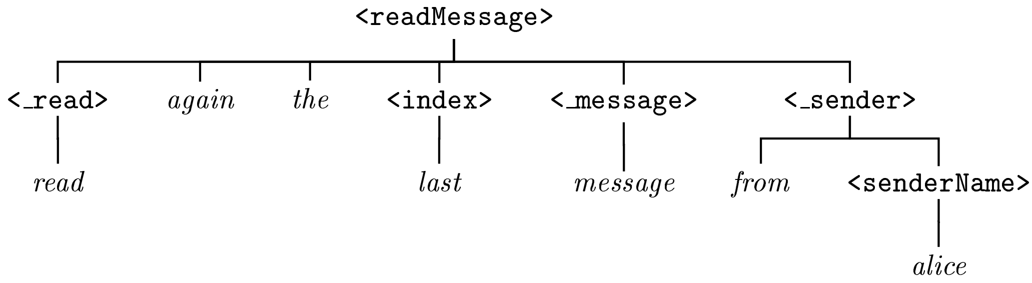


Figure 2.3: Parse tree of *read again the last message from alice* according to the grammar in Figure 2.2.

2.1.2 Defining Semantic Grammars

The specification of a semantic grammar comprises two main components: one is the definition of the *ontology*, that is, a hierarchy of the concepts that are relevant to the domain at hand, or, in other words, the principal NTs and their dominance relations, which constitute the skeleton of the semantic grammar; the other is the construction of the *rules* that tie together nonterminals with terminals and other nonterminals.

The first component, the definition of the ontology, is a design decision about what concepts are deemed important in a particular domain. For example, according to the example grammar in Figure 2.2, the word *again* in the sentence *read again the last message from alice* does not contribute to the meaning representation (and indeed *read the last message from alice* would obtain the exact same feature structure as in Figure 2.1). That is a design choice, a decision about the granularity, resolution, or amount of detail that the application developer considers sufficient for the application in question. It is a process that is basically driven by the desired behavior of the system as a whole: in the case of an e-mail client, the distinction between, say, *the last message* and *the first message* is important since a different behavior (answer from the end-application) is expected[5] and hence the neccessity for the concept of `<index>`. On the other hand, it may be perfectly fine that the answer to, or execution of, *read again the last message from alice* be exactly the same as that of *read the last message from alice* and thus the need for an "`<again>`" concept is obviated.

The second component, the writing of the grammar rules, establishes how the concepts are to be expressed. For example, the rule for `<_sender>` in Figure 2.2 expects a `<senderName>` to be introduced by either *from* or *sent by*, as in *from alice* or *sent by alice*.

[5] Unless, of course, there is no message, or a just a single message, in which case the first message is the same as the last one.

But here we encounter the crux of the problem: What if one were to say *read the message that alice sent* rather than *read the message sent by alice*? Unfortunately, even though the two sentences mean the same thing, since the grammar does not allow the former there would be no parse tree, and thus no feature structure, and the end-application would not be able to perform any action.

But, in these situations, could the NLU system not show a bit of intelligence, rather than request a rephrase from the user?

2.1.3 Problem Statement

This is precisely the question that this dissertation addresses:

> Is it possible to automatically extend a semantic grammar?
> If so, how much domain knowledge and user expertise is necessary?

The answer to these questions, in the form of the GSG system, constitutes the core of this dissertation.

2.2 Approaches to Extending Semantic Grammars

Even within the semantic grammar paradigm, there are several approaches one could take to extend a semantic mapping in the face of extragrammatical[6] sentences.

Chapter 3 provides a survey of some of the main approaches to the general problem of extragrammaticality, but, as a brief remark, some of the principal paradigms are discussed below.

In one end of the spectrum one would find the traditional way of grammar extension, namely, to collect the sentences that cannot be parsed by the current grammar and ask an expert human grammar writer to carefully update the grammar to cover the new examples. This approach obviously suffers form two major drawbacks: it requires highly-skilled human labor and it takes a long time for new words and constructions to make it into the grammar. Another approach would be to apply machine learning techniques (such as Bayes classifiers, vector-space models, etc. (see [Mitchell, 1997])) in order to build a classifier that, given an utterance, predicts its most likely top-level concept, akin to the lexical-based classification task of text categorization as described in, for example, [Nigam et al., 2000]. However, this

[6]Note the distinction between *ungrammatical* and *extragrammatical*. Ungrammatical is reserved for sentences that are not grammatical in standard English (such as, say, **The boy eat fish*), whereas extragrammatical qualifies a sentence that lies beyond the language defined by a particular, formal grammar (such as the semantic grammars discussed in this dissertation) but which may not necessarily be ungrammatical.

approach would yield a flat classification and not a structure with embedded arguments such as a parse tree.

An approach that is able to retain the structure is minimal-distance parsing, where the system, upon encountering an extragrammatical sentence, finds the parsable sentence that is closest to the extragrammatical one, according to the Damerau-Levenshtein metric at the word level (see [Hall and Dowling, 1980]), that is, with the smallest number possible of word deletions, insertions and substitutions. But the main drawback of this approach is the exponential complexity of the algorithm, both in terms of the time and space required to find the closest grammatical sentence.

Yet another way would be to automatically infer, or induce, a grammar from a collection of sentences (see, for example, [Parekh and Honavar, 2000]). The problem with such an approach, beyond the known theoretical results that severely limit what can be learned, is that the resulting, inferred grammar will be rather incomprehensible and give rise to parse trees that are not suitable for generating the kinds of feature structures desired to communicate with the end-application.

On the other hand, a different paradigm could be attempted in which the training set is constituted by parse trees (as in a treebank) rather than sentences. Still, the problem in that case is that such an approach could be used to, for example, derive the initial, kernel grammar, but not as a means to expand an existing grammar to cover extragrammatical sentences for which, by definition, there are no trees.

Even more approaches are discussed in the following chapter, but the conclusion is that none of them provides the interactivity, combination of strategies, and rule management sophistication achieved by GSG.

2.3 The GSG Approach

The GSG system presented in this dissertation follows a simple idea: since what really matters for natural language understanding is the final representation of a sentence (e.g., in the form of a parse tree), given an extragrammatical sentence, efforts should be centered on constructing, with all available means, the correct semantic representation, leaving the extension of the grammar as a side-effect of this central objective. Moreover, since we are dealing with human-computer interfaces, one should not forget that the “available means” include the human interlocutors (the users of the end-application) who, in fact, are very much interested in the correct understanding of their sentences. In other words, the system, besides bringing to bear all the domain and linguistic knowledge it has been furnished with,

<_read>		<_message>		<senderName>	
read	*the*	*message*	*that*	*alice*	*sent*

Figure 2.4: Analysis of *read the message that alice sent* according to the grammar in Figure 2.2. Note that the sentence is extragrammatical but, still, a fragmentary analysis can be constructed.

should be able to engage the human users in a meaningful conversation and thus benefit from their knowledge.

This is what GSG does: it interactively constructs the correct semantic representation for an ungrammatical sentence via its own prediction models and learning strategies as well as via clarification dialogues mantained with the end-application user. Then, after it has arrived at the correct meaning representation, new grammar rules are extracted from the learning episode and potentially added to the grammar. To that effect, GSG employs a variety of external and derived knowledge soruces and learning strategies, as is explained in detail in both Chapters 4 and 6.

2.3.1 Example

To give an idea of the interactions and learning that occur in GSG, we can continue with the example that illustrated the core of the problem: What happens if we say *read the message that alice sent* to a GSG system that has been seeded only with the grammar in Figure 2.2? The sentence is extragrammatical and yet, via analyisis and clarification questions, GSG is able to infer the meaning and correctly expand the grammar. Again, the details will become clear in subsequent chapters (especially Chapters 4, 5 and 6), but what follows is a brief explanation of the mechanisms employed.

GSG, by running the SOUP parser in all-top mode (see Chapter 5, and §5.5.3.1 in particular), is able to extract the partial analysis depicted in Figure 2.4. That enables the conversational interaction shown in Figure 2.5.

The first question is formulated to establish the top-level concept of the unparsable sentence. Since the example grammar only has one, in this case the question is trivial but, in general, the prediction models (see §6.2.4) are invoked to rank top-level concepts by likelihood. The second question comes from the fact that <_sender> is a required daughter of <readMessage> and yet is not present in the partial analysis, so it is postulated that the

Interaction 2.4

```
> read the message that alice sent
I don't understand right away what you mean but let me guess...
Is "read the message that alice sent" a way to express read message, e.g.
"read message from alice"?
     1.  Yes
     2.  No
> yes
Is "that" a way to express sender, e.g.  "from alice"?
     1.  Yes
     2.  No
> "that alice sent" is a way to express sender
Thanks for teaching me the meaning of "read the message that alice sent"!
I understand "read the message that alice sent"
```

Figure 2.5: Interaction between the GSG system seeded only with the grammar in Figure 2.2 and a user. Text preceded by '>' is entered by the user.

unparsed fragment *that* is a way to express it. The user replies that it is in fact *that alice sent* what expresses `<_sender>`. This illustrates the mixed-initiative nature of the conversation: even when the expected answer is just *yes* or *no*, the system is able to accommodate a more complex response from the user.

And what is the end-result of this learning episode? The seamless extension of the grammar in Figure 2.2. As shown in Figure 2.6, the newly acquired subRHS *that* `<senderName>` *sent* is added under `<_sender>`, and the resulting, extended grammar is listed in Figure 2.7. This means that, after the relatively simple dialogue in Figure 2.5, the previoulsy extra-grammatical construction *that alice sent* (and *that bob sent*, etc.) is understood. This is a radically different behavior from that of asking the user for a rephrase!

2.3.2 Summary of Results

In order to test the validity of the approach in a real setting, experiments were conducted in two different applications: an e-mail client program, where one can check for new e-mail messages, respond to them, sort them, etc. (henceforth, the E-Mail Task) and a virtual music store, where one can listen to songs and buy them (henceforth, the Musicbox Task). In both cases the users, drawn from the population at Carnegie Mellon University minus the Language Technologies Institute, freely interacted with the system, with only a minimal initial instruction.

Chapter 7 delves into the details of the experiments and their results, but as a way of summary, Figure 2.8 is presented here. It shows not only that GSG is indeed able to engage

LE 1

Trigger utterance: *read the message that alice sent*

Total number of choices: 2

SubRHS learned under <_sender>:

that <sender> *sent*

Original rule:

```
<_sender> = (from | sent by) <senderName>;
```

Resulting rule:

```
<_sender> = (from | sent by) <senderName> | that <senderName> sent;
```

Score: 2

Figure 2.6: Summary of the learning episode resulting from the interaction in Figure 2.5. Note that the RHS for <_sender> has been extended with that <senderName> sent. Scores are assessed a posteriori for evaluation purposes. A score of 2 is "excellent."

```
public <readMessage> = <_read> [again] [the] [<index>] <_message> <_sender>;
       <_read>       = read | speak;
       <index>       = first | last;
       <_message>    = message | messages;
       <_sender>     = (from | sent by) <senderName> | that <senderName> sent;
       <senderName>  = alice | bob;
```

Figure 2.7: Resulting grammar after the learning episode in Figures 2.5 and 2.6. Note that now <_sender> can also be expressed through the pattern *that X sent*.

in clarification dialogues and interactively construct semantic representations that cover the users' utterances, but also that the extended grammar generalizes well over unseen data.

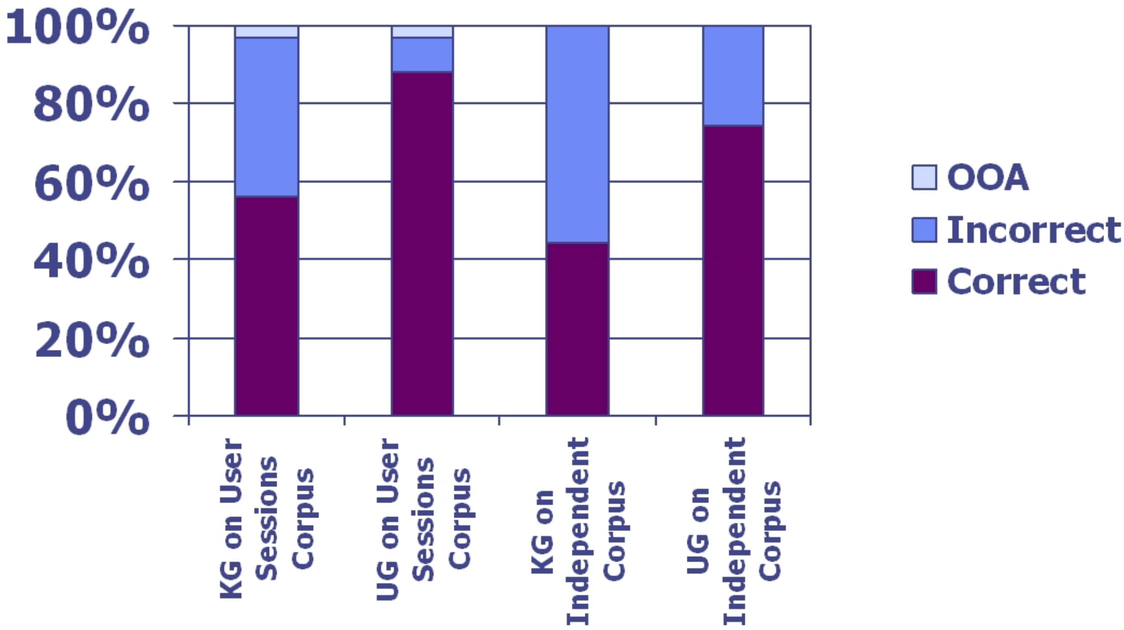


Figure 2.8: Semantic accuracy results for the E-Mail Task. The first column depicts the percentage of correct, incorrect, and out-of-application (OOA) utterances for the kernel grammar (KG) on the corpus comprised of the utterances taken from the user sessions (Session Corpus). The second column shows the performance of the union grammar (UG) on the same Session Corpus. The union grammar is created by adding, to the kernel grammar, all the rules acquired during the user sessions. The third and fourth columns repeat the experiment, this time over an independent corpus of sentences. These results are very satisfactory, as the union grammar gains over 30 absolute percentage points in semantic accuracy, both in the session corpus and in the independent corpus. (See Chapter 7, especially §7.4 for details.)

Chapter 3

Related Work

Extensive research in robust parsing and adaptive NLU systems is described in the literature. This chapter examines a few of the major approaches to the problem of extragrammaticality and contrasts them with GSG.

3.1 Two-stage Parse Repair

[Rosé, 1997] describes an interactive approach to resolve the problem of extragrammaticality, i.e., how to handle input sentences not fully covered by the existing grammar. In the ROSE system, when an utterance is not fully parsed but at least two subparses have been found, it is attempted to construct an overall analysis via the Combination and Interaction stages:

1. **Combination**. After obtaining partial parses from the parser, the ROSE system applies the Repair Hypothesis Formation genetic algorithm to hypothesize different ways of combining the subparses into a global interpretation of the input sentence. The instantiation of the genetic algorithm is as follows.

    - **Terminals**: Feature structures (FSs) representing the partial parses.
    - **Function**: Single operator that, given a mother FS, a daughter FS and a slot, attempts to insert the given slot into the mother FS with the daughter FS as the value of the slot. Even if the insertion is not successful, the operation is guaranteeed to return a single FS (which means that the smallest FS may have to be discarded). This ensures that the operator can be used compositionally, for at each step the number of remaining partial FSs is reduced.
    - **Fitness function**: Two different fitness functions are presented, one obtained through a hill-climbing search, the other through a genetic search. Both were computed from a corpus of 48 pairs consisting of a sentence and its correct analysis, and both are functions of the number of concepts, the number of repair

actions involved, the statistical goodness of the repair actions, and the percentage of the sentence that is covered by the repair.

2. **Interaction**. After the combination stage returns a set of hypotheses for the extragrammatical utterance, the ROSE system generates a set of queries to the user in order to find the hypothesis that best captures the meaning intended by the user. To minimize the number of interactions, a set of *distinguishing features* from the competing hypotheses is extracted, and then a question is generated from the feature that is *askable* (i.e., it is possible to ask a natural question from it), is *evaluatable* (i.e., it refers to a single repair or to a coöccurring set of repairs), is *in focus* (i.e., it incorporates information shared by all the hypotheses), and is *most informative* (i.e., it is likely to result in the greatest reduction of the search space). ROSE also employs a discourse processor that includes domain-dependent modules (such as the temporal expert program and the calendar program).

The motivation of both the ROSE and GSG systems is the same, namely, to be robust in the face of ungrammaticality, and they also share the philosophy of engaging the user in a repair dialogue (although ROSE does not support mixed-initiative). However, the fundamental difference is that no updates occur in the ROSE system. That is, even after a successful repair, no learning takes place, which means that upon encountering the same extragrammatical sentence again, the same repair steps would have to be taken. This contrasts with the GSG approach of learning rules as a way to remember a learning episode. Also, the repair mechanism in ROSE requires the existence of at least two subparses in the input utterance, whereas GSG has no such limitation.

3.2 Hidden Understanding Models

Hidden understanding models for natural language (as described, for instance, in [Pieraccini and Levin, 1993; Miller et al., 1994; Miller et al., 1995; Minker, 1997]) attempt to carry the success of Hidden Markov Models in speech recognition (see [Rabiner, 1989] for an introduction) over to the field of NLU. In this framework, NLU is seen as a decoding problem: given a string of words, or rather, an acoustic signal, one wants to find the most likely sequence of concepts that gave rise to that signal. In a mathematical formulation using the Bayes rule of conditional probabilites, speech decoding is expressed as

$$P(W|A) = \frac{P(A|W) \cdot P(W)}{P(A)}$$

where W represents a sequence of words and A the acoustic signal. The problem of speech recognition thus becomes that of finding the W that maximizes $P(A|W) \cdot P(W)$.
If semantics are added into the model, the formula can be written as

$$P(W, C|A) = \frac{P(A|W, C) \cdot P(W|C) \cdot P(C)}{P(A)}$$

where C represents a sequence of concepts. If, furthermore, it is assumed that $P(A|W, C) = P(A|W)$ the problem of speech understanding becomes that of finding the W and C that maximize $P(A|W) \cdot P(W|C) \cdot P(C)$. Intuitively, this can be seen as a (coarse) characterization of *acoustics* ($P(A|W)$), *syntax* ($P(W|C)$) and *semantics* ($P(C)$).
Although the particular realizations of hidden understanding models differ (see Chapter 5 of [Minker, 1997] for a survey), they all have the following steps in common.

- **Development**. The precise nature of the encoding of semantic representations into the formalism of an HMM needs to be established, in particular how to represent embedded structures, what the states $\{s_1, ..., s_N\}$ and the observation symbols $\{o_1, ..., o_M\}$ represent, and topology of the HMM, i.e., which of the state transitions $A = (a_{ij})_{N \times N},\ a_{ij} = P(s_j(t)|s_i(t-1))$ are allowed ($a_{ij} > 0$).

- **Training**. Once the topology is in place, the model parameters $\Lambda =< A, B, \Pi >$ have to be estimated, i.e., the state transition probability distribution $A = (a_{ij})_{N \times N},\ a_{ij} = P(s_j(t)|s_i(t-1))$, the observation symbol probability distribution $B = (b_{ik})_{N \times M},\ b_{ik} = P(o_k(t)|s_i(t))$, and the initial state probability distribution $\Pi = (\pi_i)_N,\ \pi_i = P(s_i(t = 1))$. For discrete Markov models the usual method to estimate the model parameters is to simply use the counts obtained from a set of training examples.

- **Decoding**. In the recognition or decoding mode, given the model $\Lambda =< A, B, \Pi >$ and an observation sequence $O =< o(1), ..., o(T) >$ one wants to obtain the sequence of states $S =< s(1), ..., s(T) >$ that is optimal according to some criterion, usually that of a maximum likelihood. The Viterbi algorithm is commonly used to efficiently compute the optimal state sequence.

Statistical frameworks like this one are promising because they allow the optimal model parameters to be automatically learned from data (rather than having to be set by hand). However, defining the semantics and topology of the HMM is not automatic, and, in fact, it is difficult to model the nestedness of natural language. GSG does employ a similar stochastic approach in the form of the Prediction Models (see §6.2.4), but only as a component of the system.

3.3 Acquisition of New Words

The idea of a self-extending NLU system has a (relatively) long history. For instance, [Carbonell, 1979] already reports on incremental learning of new words in the POLITICS system. An even earlier attempt to learn the meaning of new words is made by the FOULUP program [Granger, 1977] in which script expectations drive the meaning acquisition. For example FOULUP induces the meaning of *Rabbit* in *A Rabbit veered off the road and struck a tree* to be a "self-propelled vehicle" because the unknown word *Rabbit* matched the rôle of `vehicle` in an automobile accident script.

In the POLITICS system, new words are learned by projecting contextual expectations (syntactic, semantic, and pragmatic) as constraints on the syntactic category and the semantic rôle of the new word. For instance, if the new word follows "the" it must be a noun, adjective or adverb. The rest of the sentence provides further constraints, although it may not be possible to narrow down the new word to a single part of speech or thematic rôle; in that case all possible options are kept open awaiting more instances of the word to completely disambiguate its syntactic and semantic memberships. In addition, precompiled world-knowledge is also used to infer the meaning of a new word, but this world-knowledge turns out not to be as reliable as the constraints derived from syntax and conceptual-dependency case-frames. Two methods are then proposed to deal with inaccurate inferences: one is to simply not use world-knowledge; the other is to recover from wrong inferences: Once a contradiction (logical inconsistency) is found, the inference rule from which the erroneous conclusion was derived is deleted, as well as any other facts arising from such a conclusion. Finally, a limitation of POLITICS is that it assumes that each word has only one meaning.

In comparison with the POLITICS system, GSG requires much less effort in the encoding of domain knowledge, since all the required domain knowledge is encoded in a context-free grammar. Therefore, even if it takes a few questions to learn the meaning of a new word, GSG is more flexible and easier to port to new domains.

3.4 Adapting to the User's Language

Automated adaptation to the user's language is discussed in [Lehman and Carbonell, 1989] and [Lehman, 1989]. It is understood that total grammar coverage would be ideal but proves an infeasible goal: "Users will invariably provide unparsable input: idiosyncratic phrases, linguistically deviant utterances, or sentences simply beyond the linguistic sophistication of the interface." Therefore mechanisms for grammar adaptation to the user must be provided. In fact, one could also decide to let the users find out for themselves, through trial and error,

what the limitations of the grammar are. But such an approach, instead of leading the users to learn the system's sublanguage, often brings "frustration and disenchantment."
Regarding the nature of such user sublanguages, empirical evidence is provided that suggests the following points.

1. Speed and accuracy of task completion increase significantly as the system adapts to the linguistic usage patterns of individual users.

2. The size of each individual grammar is much smaller than their union, and individual grammars do not overlap greatly.

That is, linguistic usage patterns are *diverse* across users, but *consistent* within users.
Lehman's goal is to create a system that acquires an idiosyncratic grammar with minimal ambiguity and maximal coverage via experience with the user: the CHAMP adaptive parser remembers the recovery actions performed in the presence of an unfamiliar utterance by augmenting the grammar for that particular user with a representation that can parse the same structure directly.
The key concepts of such deviations and recoveries are:

- **Kernel grammar**: the lexicon, caseframe concept definitions, default semantic values, and syntactic forms that are present in the system prior to interaction with any user.

- **System canonical form**: any syntactic form in the kernel grammar.

- **Deviation**: the violation of an expression as embodied in a syntactic form. Four types are possible:

  1. *Deletion*: lack of expected token. E.g., *I have* ϵ *book.*
  2. *Insertion*: presence of unexpected token. E.g., *I have a the book.*
  3. *Substitution*: presence of a token different from the expected one. E.g., *I have a bok* (which in turn is a deletion deviation at the character level).
  4. *Transposition*: presence of a token in an unexpected position. E.g., *I a book have.*

- **Recovery action**: correction of a deviation by its inverse operation, namely:

  1. *Deletion*: recovered by insertion. E.g., *I have* ϵ *book* $\longrightarrow$ *I have a book.*
  2. *Insertion*: recovered by deletion. E.g., *I have a the book* $\longrightarrow$ *I have a book.*

3. *Substitution*: recovered by substitution. E.g., *I have a bok* $\longrightarrow$ *I have a book.*
4. *Transposition*: recovered by transposition. E.g., *I a book have* $\longrightarrow$ *I have a book.*

- **User canonical form**: a new syntactic form created in response to recovery actions.
- **User model**: the union of system and user canonical forms.

It is proven that user canonical forms grow in a *self-bounded* fashion, that is, an individual's language usage converges on a stable subset of the language, as opposed to unbounded growth.

The learning algorithm proceeds as follows: The system begins with the kernel grammar; if the user is known and individual canonical forms have been previously created, those are loaded as well to form that user's complete model. As user inputs are processed, the system attempts to parse them using the possibly augmented kernel grammar. If non-deviant parsing is possible, no further modification is needed. Otherwise, recovery strategies are attempted in a *least-deviant-first manner*, i.e., to obtain minimal-distance parse. If a deviant parse is confirmed as correct by the user, the transformed grammar construct is subjected to a "conservative generalization process" and added to the user model.

GSG has adopted Lehman's philosophy of language acquisition in a goal-oriented environment by augmenting the grammar with new rules that encode the learning episodes. However, whereas Lehman's CHAMP requires a complex caseframe representation to define the actions and objects in the domain that are brought to bear in constraining the parse search, GSG only requires a context-free grammar, from which the domain concepts and their relations are automatically extracted (see §6.2.2).

Also, a major difference between GSG and CHAMP is the stochastic framework of GSG: from the Probabilistic Recursive Transition Networks used to encode the grammar (see §5.3), to the statistical Prediction Models (see §6.2.4), the usage of quantitative information abounds, giving the system more flexibility and the abilility to fine-tune parameters (e.g., skipping a certain word should be more costly than skipping another one, or violating a certain constraint should be penalized less than violating another). This contrasts with CHAMP, where all deviations in its minimal distance parsing are penalized equally.

Another difference is CHAMP's restriction, due to the time complexity of minimal distance parsing (compounded with the performance of computers in 1989), of only being able to learn from sentences that contain at most two deviations from the grammar. GSG has no such limitation.

Finally, there is a plethora of small details that make GSG a worthy descendant of CHAMP. They include the usage of part-of-speech information and shallow syntactic parsing to segment unparsed segments (see §6.2.5), which represents an improvement over CHAMP's Single Segment Assumption, a much clearer distinction between task information (all contained in the grammar) and learning strategies, the mixed-initiative nature of the conversation that GSG engages in (cf. the tight control CHAMP mantains over the dialogue), and, in general, the multiplicity of strategies that GSG pursues (All-top Parsing, Anchor Mother Predictions, Required/Is-a/... Daughter Search, Verbal Head Search, Parser Predictions, Vertical Generalization, Horizontal Generalization), which contrasts with CHAMP's single strategy of minimal distance parsing.

3.5 Linguistic Knowledge Acquisition from Parsing Failures

When a system fails to analyze a certain input sentence, some robust parsers hypothesize errors in the input string in order to recover and achieve a parse. [Kiyono and Tsujii, 1993], however, propose a different set of actions in the presence of an extragrammatical utterance: instead of assuming error on the user's part, assume lack of grammar coverage. Such an approach, which can be carried out both at the word level and at the sentence level, can be expressed in the following principles.

- If a word is not in the dictionary, do not assume that it is a misspelling by trying to apply the recovery operators (as described in §3.4), which in fact define a potentially infinite search space. Instead, assume it is a genuine new word and try to learn as much as possible.

- If a sequence of nonterminals cannot be parsed, again, do not attempt recovery operations but instead treat it as a new rule and try to learn its intended meaning.

In Kiyono and Tsujii's program, when the parse fails to analyze a sentence, the partial parsing results are investigated and all possible modifications to the existing grammar that would produce a complete parse of the input sentence are hypothesized. The formal algorithm is listed in Figure 3.1.

The algorithm starts at the starting symbol S and proceeds to explore each nonterminal in a top-down fashion. If $CatA$ is a failed category, the procedure is called recursively for all daughter categories in any right-hand side of a $CatA$ rewrite rule. ($CatA$ is considered a failed category iff it does not appear in the partial parse tree constructed in the attempt to analyze the input sentence, a notion similar to that of an inactive edge in a chart parser.) Then, if a certain daughter $CatB_{i,j}$ is also a failed category, a new left recursive rule for the preceding category $CatB_{i,j-1}$ is hypothesized, i.e. $CatB_{i,j-1} \longrightarrow CatB_{i,j-1}\ CatR_{j-2}$ ...

```
hypo-gen(CatA)
    begin
        if CatA is a failed category then
            for each i in CatA ⟶ CatB_{i,1} ... CatB_{i,n}
                for each j in CatB_{i,j}
                    call hypo-gen(CatB_{i,j})
                    if CatB_{i,j} is a failed category then
                        hypothesize(left-recursive-rule(CatB_{i,j-1}))
                    end if
                end foreach
                hypothesize(feature-disagreement(CatB_{i,1},...,CatB_{i,n}))
            end foreach
        end if
        if CatA is a non-lexical category then
            hypothesize(CatA ⟶ CatC_1 ... CatC_l)
        else-if CatA is a failed category then
            hypothesize(CatA ⟶ Word)
        end if
    end
```

Figure 3.1: Kiyono and Tsujii's algorithm for parse recovery (from [Kiyono and Tsujii, 1993]).

$CatR_1$ where $CatR_{j-2}$ ... $CatR_1$ are adjacent successful categories next to $CatB_{i,j-1}$. If, on the other hand, all the daughter categories are successful (and yet the mother category failed), a feature disagreement among the daughters is hypothesized. When the procedure has been applied to all the daughters of $CatA$, unless $CatA$ is a lexical category (i.e., a preterminal) the new rule $CatA \longrightarrow CatC_1$ ... $CatC_l$ is hypothesized, where $CatC_1$ ... $CatC_l$ are the adjacent successful categories starting from the word position where $CatA$ is expected. Finally, if $CatA$ is a failed lexical category, the lexical entry $CatA \longrightarrow Word$ is hypothesized, where $Word$ is the word in the input string at the position where $CatA$ is expected. In this way, an unknown word (or a known word in an unexpected position) is learned.

Needless to say, this approach overgenerates, and some way of eliminating redundant or "linguistically nonsensical" hypothesized rules is needed. An example of a linguistically nonsensical rule is the trivial $S \longrightarrow Word_1$ ... $Word_n$ where $Word_1$ ... $Word_n$ form the input sentence. Clearly, adding such a rule will make the grammar accept that sentence, but without any generality. Thus different criteria for disregarding hypothesized rules are proposed, with the underlying assumption that most of the induction processes required in grammar learning have already been done by linguists and embodied in the form of the existing grammar. These criteria include:

- **Priority to the hypotheses of feature disagreement**. Assuming comprehensive coverage of the existing grammar, priority is given to feature disagreement hypotheses because they do not create new rules (they just modify their applicability conditions by lifting or generalizing certain feature agreements). In fact, in the reported implementation of the above algorithm, once a feature disagreement hypothesis is found that restores a category, the recursion is stopped and no more hypotheses are generated.

- **Restrictions on the number of daughter nodes**. The maximum number of daughter nodes that a new rule can have is empirically limited to four, the exposed reasoning being that it does not seem viable for a new rule to collect at once too many constituents into one large constituent. (I.e., once again it is assumed that such a structure would already be present in the grammar if it were linguistically sensible.)

- **Priority to the hypotheses that use generalizations embodied in the existing grammar**. Hypotheses containing sequences of constituents which can be collected into larger constituents by existing rules are discarded as redundant, that is, hypothesized rules containing higher-level nonterminals in their right-hand side are preferred since they make better use of the generalizations already defined in the grammar. For instance, if the following rules are present in the grammar:

  N-head $\longrightarrow$ *Adj N-head*
  NP $\longrightarrow$ *Det N-head*

  the hypotheses whose right-hand side contain *NP* will be preferred over those that contain, say, *Det Adj N-head*.

- **Distinction of preterminals from other nonterminals**. While the general form of context-free grammars (CFGs) does not distinguish lexical categories (preterminals) from other nonterminals, it is proposed to allow hypothesizing new lexical rules only if the mother category (left-hand side) is a preterminal.

- **Distinction of closed vs. open lexical categories**. It is assumed that the existing grammar has a complete list of function words. This means that the left-hand sides of rules for new lexical entries are restricted to members of the predefined set of open lexical categories, such as *Noun*, *Verb*, *Adjective* and *Adverb*.

- **Distinction of closed vs. open categories**. The above reasoning can be extended to other nonterminals; that is, depending on the completeness of the existing grammar one could specify a set of categories as closed and disallow the algorithm to hypothesize new rules whose left-hand sides belong to that set.

- **Use of subcategorization frames**. Before hypothesizing any new rule or lexical entry, it should be checked that such a rule is not redundant via a subcategorization frame checking mechanism. The reason is that, in the grammar formalism presented, a subcategorization frame is embedded in the feature structure of the head category and thus the correspondence with its subcategories does not appear explicitly in the rules.

- **Restrictions on the pattern of new rules**. There could be meta-rules that restrict the form of new rules. For instance, if X-bar theory is followed, only maximal projections should be allowed in the complement position.

- **Prohibition of non-lexical unary rules**. Again, while the general form of CFGs allows unary rules, it is assumed that the existing grammar exhausts all meaningful category conversions (which is what unary rules perform) and thus unary rules are discarded if hypothesized by the algorithm. (However, lexical rules, unary in nature, are of course allowed).

- **Restrictions on lexical rules**. Still, lexical rules can be restricted by taking into account a priori knowledge of the likeliness of multiple lexical category membership. For instance in English it is very common for a word to be both a noun and a verb, but extremely rare for an adverb with the suffix *ly* to be, say, a verb.

What GSG takes from this approach are some of the guidelines for the judicious extension of the grammar, in the form of the learning strategies and licensing constraints as described in §6.3. Also, it is worth noting that, in the above paradigm, as in GSG, the creation of new nonterminals is not allowed.

3.6 Automatic Acquisition of Spoken Language

In [Gorin, 1995] the principles and mechanisms underlying automatic acquisition of spoken language are reviewed. It is again noted that, traditionally, in NLU systems the hierarchy of linguistic symbols and structures has been manually constructed, "involving much labor and leading to fragile systems that are not robust in real environments," and this underlines the necessity of constructing NLU systems that are "trainable, adaptive and robust." Gorin states the following principles of language.

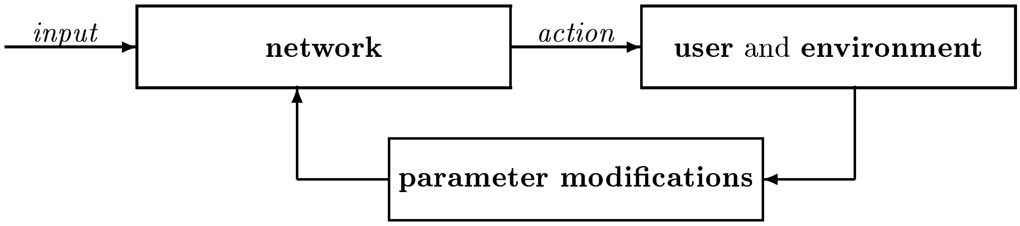


Figure 3.2: The reinforcement learning cycle (from [Gorin, 1995]).

1. The purpose of language is to convey meaning.
2. Language is acquired during interaction with a complex environment: the language-acquiring device receives some input stimuli, responds to that input and receives feedback as to the appropriateness of its response.
3. Meaning is grounded in a device's operational environment.
4. In order to provide rapid learning and generalization, the language-acquiring device must reflect the structure of its input/output periphery and environment.
5. Language acquisition proceeds in developmental stages, from the concrete to the abstract, from the simple to the complex.

Then four architectures are proposed, in accordance with these principles: information-theoretic networks, structural networks, symbols from signals, and grammatical inference, which are summarized next.

3.6.1 Information-theoretic Networks

Within the paradigm of the reinforcement learning cycle (see Figure 3.2) the first architecture reported maps inputs to actions is a connectionist network. The idea is that positive (or negative) reinforcement will strengthen (or weaken) the associations or weights of the connections between input and output. Many methods for learning such weights are known (e.g., backpropagation), but Gorin proposes to define them directly via *mutual information*, with the claim that such an approach has the following theoretical and practical advantages.

- *Theoretical properties.* Given suitable Markovian and independence assumptions on the language of the input sentences, it has been shown that the proposed information-theoretic network algorithm is equivalent to the criterion of classification via minimum description length, where the selected action is the one which provides for the minimum code length of the input sentence, as well as equivalent to a *maximum a posteriori* decision. [Goodman, 1992] observes that in statistical rule-based systems such as expert systems, the strength of a candidate rule can be characterized by the mutual information between the preconditions and the actions of a rule. Furthermore [Tishby et al., 1994] proves a universality theorem for information-theoretic associations: any association measure functionally related to probabilities can be rescaled to mutual information. Although the implications of such equivalences are not yet fully understood, they seem to point out that the traditional debate between connectionist and (statistical) rule-based approaches may not be an issue after all.

- *Practical advantages.* For such an algorithm, weights are computable in a single "epoch" (as opposed to thousands in backpropagation).

The described network is composed of three layers:

- *Input layer*: M nodes (called "word detectors") encode the vocabulary (one node per word).
- *Intermediate layer*: $M \times M$ nodes (called "phrase detectors") encode word pairs. This layer could naturally be extended to word triplets, etc. (at the cost of an exponential explosion in the number of nodes and weights).
- *Output layer*: K nodes (called "semantic nodes") encode the actions (i.e., the "meaning" of the sentence).

The $M \times M \times M$ weights between input and intermediate nodes, and the $M \times M \times K$ weights between intermediate and output nodes are calculated via smoothed relative frequencies.
How the system then works is very simple: Given a sentence, it is run through the network and classified into the action whose output node value is highest.
Gorin reports on the implementation of such a system for an automated call routing application in a department store, where the possible "actions" ($K = 4$) are Furniture, Clothing, Hardware, and a catch-all Unknown. (In fact, the actual network constructed for this experiment had only two layers, input and output.) Another application was built for customer/operator telephone transactions, this time with $K = 21$. In both cases incremental growth of vocabulary and updating of weights is allowed, but no final performance results are reported.
The limitation of such systems is clear: they are equivalent to sorting natural-language sentences into bins or classes, thus providing only a *flat* analysis of the input. In certain applications such as topic identification that may be all that is needed, but of course for

the applications that GSG supports, where the target semantic structures are much richer, such an approach is not sufficient.

3.6.2 Structured Networks

It is precisely the above limitation that motivates structured networks. Two subclasses of structured networks are described:

- *Product networks.* In this case the output is not only a semantic action, but a *parameterized* semantic action. If the parameter values are the same for all actions, the action space is isomorphic to the Cartesian product $K \times N$, where K is again the number of actions (now called "semantic primitives") and N is the number of parameter values. For instance, in the ALMANAC system, a two-dimensional product network was constructed, with $K = 20$ queries about $N = 50$ U.S. states. The system works by building two separate information-theoretic networks (one with K output layer nodes, the other with N output layer nodes). Given an input sentence, it is run through both networks and then the pair of semantic primitive and parameter value that gives highest outer sum of the two output vectors is selected as the final result.

  This approach could be extended to more dimensions, but still, the structure of the output will never be richer than the given number of dimensions, which constrasts with GSG's semantic parse trees, that have no a priori limit on their depth.

- *Sensory primitive subnetworks.* In many cases the appropriate machine response (action) depends not only on the input sentence, but on the state of the environment. A minimal example is the command *Lights, please*, whose correct action is `turn-on-lights` or `turn-off-lights` depending on whether the lights are currently on or off. Of course a solution would be to map such a sentence to a `toggle-lights` action, but still that would only delay the ambiguity, so it is in fact very useful to supply extra-linguistic information to the classification process. This idea is developed in Sankar's BLOCKS WORLD [Sankar and Gorin, 1993], where a learning device with the following innate characteristics is described.

  - It can sense the color and shape of the objects in its visual scene.
  - It is attracted to bright and moving objects.
  - After focusing on some object, it becomes bored, i.e., its attraction to the focus object diminishes over time.
  - It constructs associations between linguistic and visual events that coöccur temporally.

This last characteristic is implemented by sensory primitive subnetworks that learn associations between the linguistic and visual sensory inputs. In this case the learning of a word (i.e., the association between input string and object shape) is made dependent on the environment in the form of two-dimensional focus coördinates.

3.6.3 Symbols from Signals

In the speech recognition context, the task of natural language understanding can be seen as the correct mapping between the speech signal and the desired action, i.e., the notion of word need not be defined a priori; instead one would hope that it emerges spontaneously from the learning process.[1]

It is noted that for humans, an input stimulus evokes memories of associated perceptions and activities. This realization motivates defining the meaning of a word for a particular language-learning device to be the network associations between that word and the device's input/output periphery. Some of these sensory-semantic associations are incorporated in the systems described above. For instance, in the BLOCKS WORLD there are network associations between a word and the visual input periphery, factored through the color and shape sensory primitives. It is conjectured that, as a device's input/output periphery becomes more anthropomorphic, so will its representation of meaning. This can not be tested however until "sufficiently complex" devices are constructed.

Also, the concept of *saliency* of a word or phrase is defined mathematically to capture the information content of that word or phrase for the language-acquisition device, and such a definition is contrasted with the traditional Shannon measure of *entropy* or *information content* understood as the uncertainty about the occurrence of a word or phrase.

3.6.4 Grammatical Inference

All the above experiments were in fact very much tailored to the word level. For instance, the network associations of a word or phrase are context-independent. Grammar, however, serves to modulate the meaning of a word according to its position in the sentence as well as constraining the allowable word sequences.

As a first step it is then suggested to automatically learn grammatical categories, since parts-of-speech are often the preterminals of grammars with a more complex hierarchy of nonterminals. A part-of-speech or preterminal is defined relative to a learning device as the set of words that are strongly associated to some dimension of the device input/output

[1] Taking this idea to the extreme, one may hypothesize a machine that directly maps "brain patterns" representing thoughts and desires into actions, thereby altogether bypassing the usual notion of natural language.

U: `Pittsburgh kara Frankfurt made.`
[From Pittsburgh to Frankfurt.]
S: `Would you like to see the flights from Frankfurt to Pittsburgh?`
U: `Iie, gyaku desu.`
[No, the other way around.]
...
U: `Barcelona kara Hong-Kong made.`
S: `Would you like to see the flights from Barcelona to Hong-Kong?`
U: `Hai.`
[Yes.]

Figure 3.3: Example interaction between the user (U) and the system (S) to illustrate the behavior of embedded information-theoretic networks. (See §3.6.4.)

periphery, and only one preterminal can be learned per such dimension. A method of *saliency thresholding* within an information-theoretic connectionist network is proposed: For each dimension of the device periphery a subnetwork that corresponds to a part-of-speech is defined via saliency thresholding. The resultant subnetwork is then activated only by those words or phrases that are highly salient for its semantic or sensory primitive.

As discussed above, information-theoretic networks are very limited in the output structure they can handle. A first solution to this problem is the design of multi-dimensional networks. Another approach is to construct a network of embedded subnetworks, as reported in [Gertner and Gorin, 1993] (although limited to two levels): a system to analyze queries to the Atis database is built as an information-theoretic network in which each node of the main network is in itself another information-theoretic subnetwork. Thus, the subnetworks correspond to nonterminals for place and object names, thereby adding an extra level of generality in the learned "grammar." Note that, following the developmental nature of language learning, those subnetworks must be learned first. As an example, consider the interaction between an English-understanding, Japanese-speaking user (U) and the system (S) depicted in Figure 3.3.

In contrast to non-embedded information-theoretical networks, the system described is able to abstract the association between `Pittsburgh kara` and `from Pittsburgh` to the general association between `<city-name> kara` and `from <city-name>`, thanks to the embedded subnetworks that act as preterminals for locations.

While agreeing on the underlying principles expressed by Gorin, GSG takes a more symbolic approach as its final objective is to extend a rule-based grammar. However, the Prediction Models (see §6.2.4) can be seen as a generalization of structured networks.

3.7 Concluding Remarks

GSG benefits from the research described in the literature and attempts to incorporate the best features of the systems sketched above (such as stochastic framework, interaction with the user, and rule acquisition and generalization). At the same time, even though GSG is based on grammars and Newell and Simon claim that production systems are central to human cognition [Newell and Simon, 1972], it should be clear that GSG does not intend to model the cognitive processes whereby humans acquire language. (For the interested reader in the cognitive aspect of language acquisition, see, for example, [Brent (ed.), 1997; Baker and McCarthy (eds.), 1981; Bloom (ed.), 1994].)

Chapter 4

Philosophy and Modus Operandi of GSG

This chapter details the motivation for and advantages of GSG. First, it begins with a review of the traditonal approach to grammar development and presents an alternative model. Then, it shows how GSG implements the new paradigm.

4.1 A New Paradigm in Grammar Extension

As mentioned in §2.1, defining the mapping from words onto a semantic representation constitutes one of the critical paths in the development of conversational NLU systems: It takes in the order of months or years of highly-skilled labor (usually computational linguists) to develop a semantic mapping, for example in the form of a semantic grammar, that is comprehensive enough for a given domain. Yet, due to the very nature of human language, such mappings invariably fail to achieve full coverage on unseen data.

If we analyze the process of developing a semantic grammar for a new domain, we find that the following stages are involved.

1. **Data collection**. Naturally-occurring data from the domain at hand are collected. For example, in an e-mail application, one would want to gather the kinds of utterances that a real user of the final application may say, such as *Do I have mail?*, or *Sort messages by sender*. Typically, these data are gathered through a Wizard-of-Oz setting, where a human simulates the system's response.

2. **Design of the domain model**. A hierarchical structuring of the relevant concepts is built in the form of an *ontology* or *domain model*. Through the analysis of the data collected and the knowledge of the functionality that the end-application provides, the relevant concepts are defined. For example, as discussed in §2.1.2, the concepts of `<readMessage>`, or `<index>` are deemed necessary in the E-mail Task domain.

3. **Development of a kernel grammar**. A grammar that covers a small subset of the collected data is constructed. This initial grammar usually contains only one or two ways to express a concept.

4. **Expansion of grammar coverage**. Lengthy, arduous task of developing the grammar to extend its coverage over the collected data and beyond. Typically, it takes a computational linguist many develop-and-test cycles over the course of months to bring coverage on unseen data to minimally acceptable levels. (See Figure 4.1 for an example.)

5. **Deployment**. The grammar is frozen and released in the final application. Usually no mechanisms are in place to collect extragrammatical sentences; and, even if there are, not until the next release will the extragrammatical words and constructions become grammatical.

However, as noted in [Lehman, 1989], this paradigm can be greatly improved: If we allow the deployed system to dynamically extend the underlying grammar, then point 4, the most time-consuming stage, can be eliminated (or, more precisely, it becomes distributed among the users of the end-application).
And this is what the proposed system GSG accomplishes: it changes the traditional paradigm by reordering the last two points, i.e., deployment comes before expansion of grammar coverage. In other words, after the ontology and kernel grammar are defined, the final application can already be launched, because the grammar will be extended, at runtime, as an epiphenomenon of engaging the user in clarification dialogues.

4.1.1 Assumptions

This approach is based on two premises: (*i*) after deployment the domain model is fixed, and (*ii*) the communicative goal of the end-user is expressible in the domain. The justification for these assumptions comes from the fact that they allow a clean delimitation of the problem that this thesis addresses (extension of grammar coverage but not acquisition of new concepts), and also from the realization that they are quite reasonable: Even if the ontology were not fixed and the system were able to learn a new concept, the back-end application would, most likely, not know how to handle it. Also, even though passing the urestricted Turing Test should always be at the back of our minds, it is clear that the current state of the art in language technologies (and artificial intelligence in general) only allows for modeling domains rather narrowly defined by the application at hand. (Still,

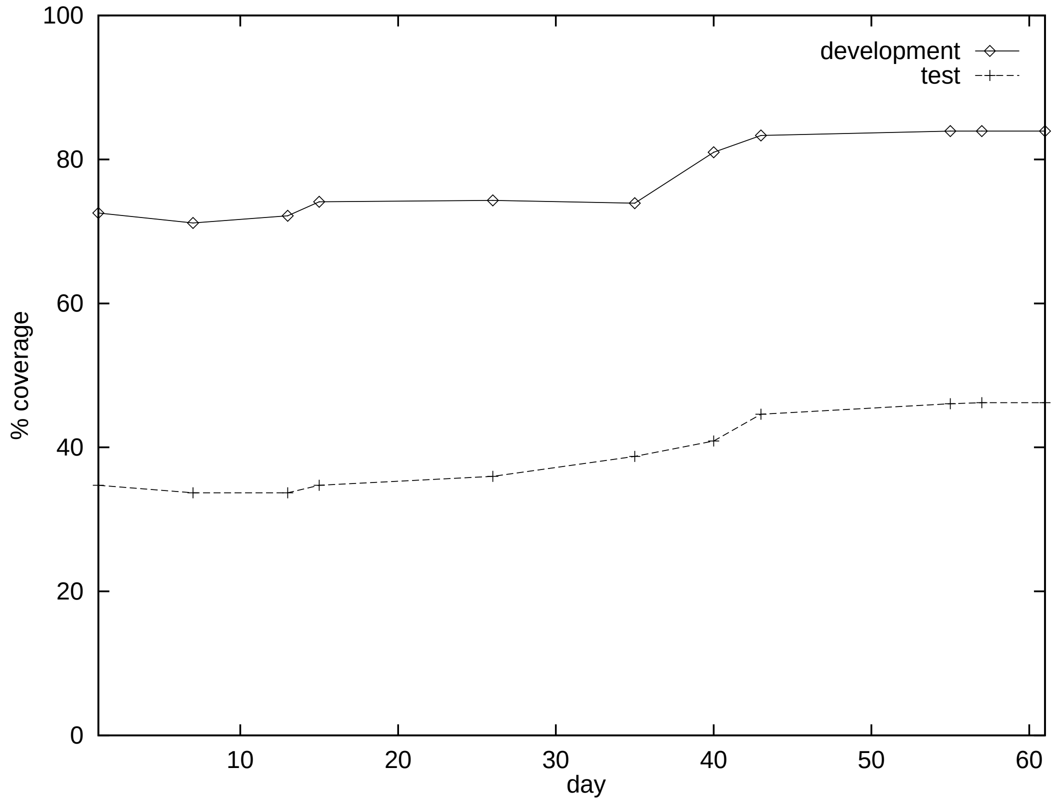


Figure 4.1: Coverage fluctuation over a period of two months as the grammar is extended by a computational linguist. Data is taken from the English Spontanous Speech Scheduling Task (briefly described in [Waibel et al., 1996]). The top line depicts coverage of the development set, the bottom one coverage of an unseen test set.

GSG does incorporate mechanisms to identify utterances that are beyond its capabilities (see §6.4.1.2).

4.1.2 The Philosophy of GSG

Thus, the philosophy of GSG is to treat an extragrammatical sentence that traditionally results in a "parse failure" as an opportunity to learn new semantic mappings. It exploits domain and linguistic knowledge to pose clarification questions and interactively constructs and learns new meaning representations. Therefore one can describe the aim of the GSG learning system as to *judiciously extend a grammar through simple, natural-language only interactions with application end-users.*

The applications of GSG are many, most notably, allowing a very fast development of NLU components in a variety of tasks. GSG enhances the usability of any application that incorporates it, because the end-users are able to easily customize the interface by teaching GSG to understand their individual language patterns.

4.2 Overview of GSG

So how exactly does GSG acquire these individual language patterns? Chapter 5 details the workings of the SOUP parser, the robust parser of CFGs that lies at the core of the system, and Chapter 6 provides the system diagram and a detailed explanation of each of the remaining components, but what follows is a high-level vision of the information flow and learning algorithms that enable GSG to acquire new grammar rules.

4.2.1 GSG's Metagrammar

Since GSG allows for mixed-initiative dialogues and the volunteering of linguistic information, it is necessary to identify utterances that are about the task (say, *send a message to bob*) from "meta-utterances" that are corrections (such as *wrong* or *that's not what i mean*) or that provide linguistic information (such as *noon is a time of day* or *by christmas i'm referring to december twenty-fith*). To that effect GSG always runs the incoming user sentence through its Metagrammar. Table 4.1 explains the top-level concepts of the Metagrammar and §C.1 lists it in full. Note that it is also written in the JSGF formalism.

It could be argued that it is better to keep meta-comments simple (e.g., single-word keywords) so as to guarantee that the metagrammar will never interfere with the task grammar; at the same time, on the other extreme, one could also argue that the metagrammar itself should be learnable. In this particular implementation, GSG takes a middle position: the metagrammar is not restricted to keywords (to obviate special training and keep the conversation natural), but it is not automatically extendible.

In any case, when an incoming utterance can be parsed by the Metagrammar, the appropriate action is triggered. The most important operators, from the language acquisition perspective, are MEANS and ISA, which are two of the fundamental algorithms of GSG.

4.2.2 GSG's Fundamental Algorithms

GSG implements three fundamental algorithms to construct meaning representations for extragrammatical utterances: Interactive Parsing, MEANS Operator and ISA Operator, as well as five other ones dedicated to rule management: Subsumption Detection, Ambiguity Detection, Vertical Generalization, Horizontal Generalization and Rule Merging. They

Name	Example	Effect
`<means>`	*ff is shorthand for fast forward*	Triggers MEANS Operator Algorithm
`<isa>`	*noon is a time of day*	Triggers ISA Operator Algorithm
`<cancel>`	*that's not what i meant*	Cancels current learning episode
`<ignore>`	*never mind*	Ignores current question
`<help>`	*i'm not sure i understand*	Provides help
`<summarize>`	*nutshell*	Provides a summary of the dialogue so far
`<yes>`	*that's right*	Affirmative response to current question
`<no>`	*i don't think so*	Negative response to current question
`<greeting>`	*hi*	Triggers appropriate nicety as reply
`<farewell>`	*bye now*	*Idem*
`<thank>`	*thanks*	*Idem*
`<thankBack>`	*it was a pleasure*	*Idem*

Table 4.1: Main concepts of GSG's Metagrammar, always the first grammar that attempts to interpret the user's utterances to determine whether they are "meta-comments" about the dialogue (`<means>`, `<isa>`, `<cancel>`, `<ignore>`, `<help>` and `<summarize>`), binary answers to the current question (`<yes>` and `<no>`), conversational pleasantries (`<greeting>`, `<farewell>` `<thank>`, and `<thankBack>`) or whether they are sentences addressed directly to the end-application.

constitute the essential machinery that enables GSG to judiciously extend the task grammar upon encountering extragrammatical utterances.

4.2.2.1 Algorithms for Interactive Parse Construction

When an incoming sentence is not dealt with by the Metagrammar it is passed to the task grammar. If the sentence is extragrammatical, i.e., not parsable in SOUP's default mode (see §5.5.1), then the Interactive Parsing Algorithm is applied to construct an analysis for it.

The **Interactive Parsing Algorithm** is introduced in Figure 4.2 (and later expanded in Figures 6.8 and 6.9). Basically, it parses the extragrammatical sentence in all-top mode (see §5.5.3.1) and uses the parse subtrees and unparsed words to establish the overall meaning of the sentence (i.e., the NT at the root of the hypothesized parse tree being constructed) using both internal knowledge sources (the Prediction Models described in §6.2.4) and external knowledge sources (possibly the End-Application Manager, and most importantly, the end-user). Once the root is set, the daughter arguments are searched for by using the evidence, constraints from the Ontology, and guidance from the user offered either in the form of answers to specific questions formulated by the system or in the form of volunteered information (which trigger the MEANS or ISA operators). The intermediate result of this

Input: natural language sentence.

1. **Collect evidence**: Parse sentence with all NTs as top-level, i.e., able to stand at the root of a parse tree. This will result in a sequence of subtrees and unparsed words.

2. **Establish anchor mother**: Apply Prediction Models to hypothesize NT roots from the evidence collected in Step 1. This will result in a ranked list of NTs. Possibly filter or re-rank list with information about the state of the end-application as given by the Backend-application Manager. Pose confirmation or multiple-choice question to the user to establish anchor mother.

3. **Construct daughters**: Apply Daughter Argument Selection strategies (Verbal Head Search, Required/Is-a/... Daughter Search, and Parser Predictions) to hypothesize the structure under the anchor mother.

4. **Update grammar**: Extract rules from the hypothesized tree. Perform vertical and horizontally generalization and add resulting rules to the grammar if they are not already subsumed by existing rules in the grammar, do not increase grammar ambiguity, and do not disrupt previoulsy correct parses.

5. **Update Prediction Models**: Use hypothesized tree to update the Paratactical and Hypotactical Models.

Figure 4.2: Summary of GSG's Interactive Parsing Algorithm for sentences that are parsed neither by the Metagrammar nor by the task grammar (in default, non-all-top mode). Chapter 6 provides a more detailed view of this algorithm (see Figures 6.8 and 6.9).

algorithm (except of course when the learning episode is canceled) is a parse tree that covers the previously extragrammatical sentence with the correct structure. Then, from this parse tree, rules are extracted, generalized and possibly added to the current task grammar. If this rule is not already subsumed by the grammar and it does not introduce ambiguity, it is vertically and horizontally generalized and merged with the existing grammar. (These steps are explained in §4.2.2.2 and §6.3.)

On the other hand, when an incoming sentence is identified by the Metagrammar to be of type MEANS, the MEANS **Operator Algorithm** is invoked. The MEANS operator establishes the meaning equivalence of two sentences, such as *arrange* MEANS *sort* (an operation that can be triggered by the user saying, for example, that *arrange means the same as sort*). The MEANS Operator Algorithm is in fact an instantiation of the Interactive Parsing Algorithm with the further constraint that the hypothesized roots (i.e., candidate

anchor mothers) be present in the trunk[1] of the paraphrase's analysis. That is, given x MEANS y (where x and y are sequences of words) a tree $T(x)$ is hypothesized as the correct interpretation of x with the constraint that the root of $T(x)$ be present in the trunk of $T(y)$.[2] Such construction, however, is not always possible, e.g., when the number of words in x is less than the number of required branches to emulate the structure of $T(y)$, as in *christmas* MEANS *december twenty-fifth*. In that case, the meaning of x cannot be encompassed by a parse tree from which new rules will be acquired but rather the mapping from string to tree has to be learned as a single unit (see, for example, Figures 6.20 and E.3). Note that in the case of such direct mappings, no vertical generalization is performed.
Similarly, when a sentence is parsed by the Metagrammar as an ISA meta-concept, the ISA **Operator Algorithm** is applied. The ISA operator establishes the class of an expression, as in *tuesday* ISA `<dayOfWeek>` (triggered, for example, by the user saying that *tuesday is a day of the week*), or, more generally, establishes a conceptual instance-of or part-of relation, as in `<dayOfWeek>` ISA `<time>`. The ISA Operator Algorithm is also an instantiation of the Interactive Parsing Algorithm with the further constraint that the hypothesized root (anchor mother) is already given. In this case GSG's capability of fuzzily matching NT names (see §6.3.6.4) is employed to establish the anchor mother from the expression given by the user.

4.2.2.2 Algorithms for Rule Management

Once a parse tree has been constructed for the extragrammatical sentence, rule candidates are extracted from the parse tree[3] and run through a series of algorithms to determine the final form that they should take, if any. First, the **Subsumption Detection Algorithm** is applied to establish whether the candidate subRHS is a particular case of an existing one

[1]The trunk of a parse tree is the sequence of NTs starting at the root node and descending as long as the current node has only a single daughter node that is of type NT. For example, the trunk of the parse tree in Figure 5.1 (b) is < `<request>`, `<suggestTime>` >.

[2]This of course requires y to be parsable; if it is not a message is issued and the learning episode concluded.

[3]For each parse tree node that is not a leaf, a rule is created with the left-hand side as the NT in the parse tree node and the right-hand side as the sequence of node's immediate daughters (i.e., in general, a combination of Ts and NTs). For example, from the parse tree in Figure 2.3 the following rule candidates would be extracted:

- `<readMessage>` ⟶ `<_read>` *again the* `<index>` `<_message>` `<_sender>`
- `<_read>` ⟶ *read*
- `<index>` ⟶ *last*
- `<_message>` ⟶ *message*
- `<_sender>` ⟶ *from* `<senderName>`
- `<senderName>` ⟶ *alice*

and thus redundant; then the **Ambiguity Detection Algorithm** is invoked to establish whether adding the candidate subRHS would increase the ambiguity of the grammar. If the subRHS is deemed safe, it is generalized through the **Vertical Generalization Algorithm**, which follows Is-a links in the Ontology, and the **Horizontal Generalization Algorithm**, which makes selected constitutents of the subRHS optional and/or repeatable. Finally, the **Rule Merging Algorithm** is applied to insert the new subRHS into the task grammar in a generalized manner. (See §6.3 for the details.)

4.2.3 Enabling a Natural Dialogue

An aspiration of GSG is to provide a natural feel not only to the interface of the end-application but also to the dialogue that takes place during the learning episodes. Integral requirements for that purpose are (*i*) ability to remember what the interlocutor has said, and (*ii*) ability to keep a stack of attentional topics, that is, the capacity to shift the focus of attention, deal with a subtopic, and then return to the original focus of attention. GSG models (*ii*) through the Dialogue Manager, which contains a focus stack; and (*i*) both via the Interaction History, a repository of the answers the user has given during the session to questions formulated by the system, as well as via the extension of the task grammar through the acquisition of rules.

Also, related to this conversational naturalness, is GSG's ability to dynamically create grammars to parse the user's response to multiple choice questions. For example, given a question such as

```
"get rid of all messages from spamela" is a way to express:
     1.  delete mail, e.g.  "delete"
     2.  reply mail, e.g.  "reply"
     0.  None of the above
```

one can answer with either *1*, *one*, *delete*, or *delete mail* and still select the same choice.

The following Chapter 5 details the workings of the SOUP parser. It is in Chapter 6 where the algorithms outlined in this chapter are revisited in more detail.

Chapter 5

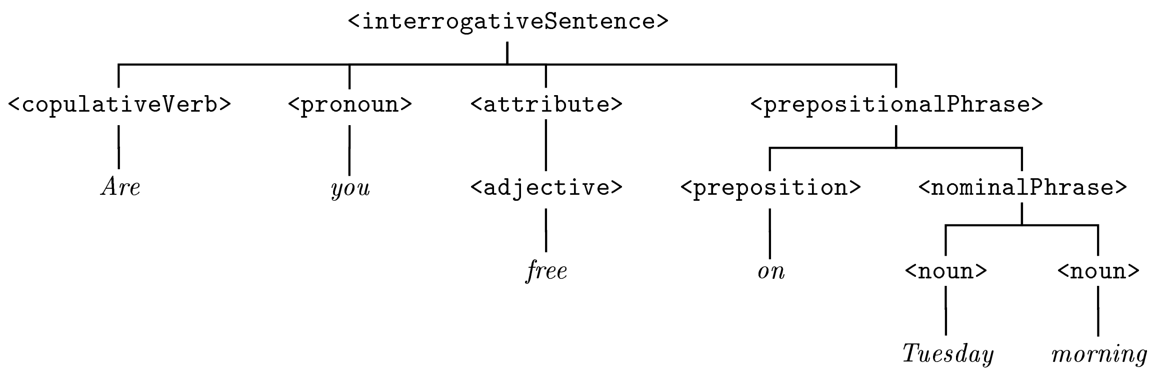
The SOUP Parser

This chapter describes the SOUP parser. It starts with a brief overview of parsing and grammar formalisms, then explains the algorithm and principal features of the SOUP parser, and ends with a note on the graphical user interface GSOUP.

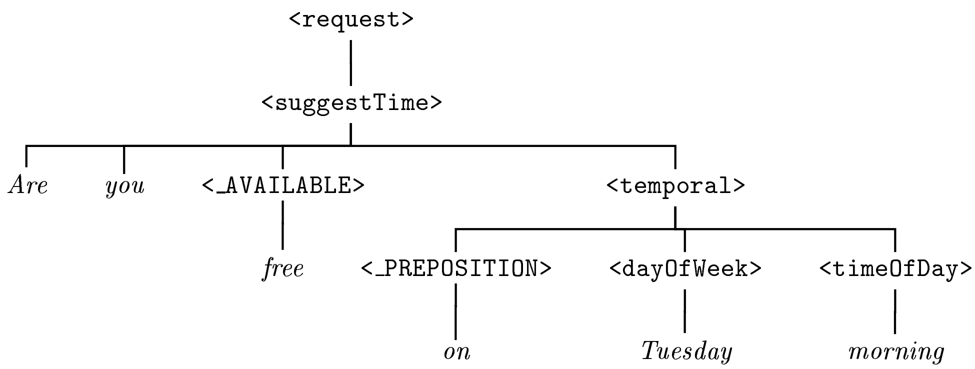
5.1 Parsing

Parsing can be described as the assignment of structure to a sequence of words according to a grammar. A sentence (sequence of words) can obtain very different analyses depending on the grammar it is parsed with. Figure 5.1 illustrates this point by showing the parses for *Are you free on Tuesday morning?* according to two grammars, one syntactic (where the nonterminals correspond to grammatical categries), and the other semantic (where the nonterminals correspond to concepts relevant in a particular domain).

We see therefore that the grammar with which sentences are analyzed is very important, as it determines the parse structures and thus the information that will be extracted from the analysis. From a theoretical standpoint, it is worth mentioning the study by Noam Chomsky [Chomsky, 1956; Chomsky, 1959] of different types of grammar formalisms, in particular the weak generative capacity of different rewrite systems. (See Table 5.1.) The grammars that SOUP handles are context-free, because this class of grammars allows for fast parsing speed and still has sufficient power to cover most of natural language. In fact, although there is evidence that natural languages are not, in general, context-free (see [Culy, 1985; Shieber, 1985]), it has been proposed [Mayfield et al., 1995a; Mayfield et al., 1995b; Woszczyna et al., 1998] that, at least for task-oriented semantic grammars, the advantages in parsing speed and ease of grammar construction of a pure-CFG formalism outweigh the lack of features offered by other formalisms (cf., for example, the [Verbmobil Semantic Specification, 1994]). Parsing then becomes a *search* for the "best" analysis of a sentence according to a grammar. Many parsing algorithms have been proposed in the literature (see [Grune and Jacobs, 1990]



(**a**)



(**b**)

Figure 5.1: Syntactic (**a**) and semantic (**b**) parses for the same sentence.

and [Tomita and Bunt, 1996] for a good sample), but here only SOUP's top-down beam search will be described.

Some of the terminology employed in the explanation that follows may be worth clarifying: a **parse tree** is a tree covering a contiguous subsequence of the input sentence[1]; an **interpretation** is a sequence of non-overlapping parse trees that cover the input sentence (either fully or partially); finally, an **analysis** of the input sentence is a ranked list of interpretations.

[1]Except in the case of intra-concept skipping, where some input words may be omitted.

Grammar Type	**Grammar Productions**	**Language**	**Accepting Machine**
Type 0: Unrestricted, phrase structure	$u \longrightarrow v$ $u \in (V \cup \Sigma)^+$ $v \in (V \cup \Sigma)^*$	Recursively enumerable (Any computable function)	Deterministic, nondeterministic Turing machine
Type 1: Context-sensitive, monotonic	$u \longrightarrow v$ $u \in (V \cup \Sigma)^+$ $v \in (V \cup \Sigma)^*$ $\text{length}(u) \leq \text{length}(v)$	Context-sensitive (E.g. $\mathtt{a}^n\mathtt{b}^n\mathtt{c}^n$)	Linear-bounded automaton
Type 2: Context-free	$A \longrightarrow v$ $A \in V$ $v \in (V \cup \Sigma)^*$	Context-free (E.g. $\mathtt{a}^n\mathtt{b}^n$)	Pushdown automaton
Type 3: Regular, left-linear right-linear	$A \longrightarrow aB$ $A \longrightarrow a$ $A \longrightarrow \lambda$ $A, B \in V$ $a \in \Sigma$	Regular (E.g. $\mathtt{a}^n$)	Deterministic, nondeterministic finite-state machine

Table 5.1: The Chomsky hierarchy of grammars, a classification of grammar families according to their weak generative capacity. V is the set of nonterminal symbols, Σ the set of terminal symbols, and λ represents the empty string. $(V \cup \Sigma)^+$ denotes the set of non-empty strings composed exclusively of symbols from V and/or Σ; $(V \cup \Sigma)^*$ denotes the same set except that the empty string is included.

5.2 Grammars

As shown in Table 5.1, a context-free grammar is composed of a set of terminals Σ (i.e., the vocabulary of the language), a set of nonterminals V, a set of starting nonterminals $S \subset V$, and a set of rules R. Each rule in R is of the form $A \longrightarrow v$ (i.e., symbol A can be rewritten as v), where $A \in V$ (i.e., the left-hand side of the rule is composed of a single nonterminal A), and $v \in (V \cup \Sigma)^*$ (i.e., the right-hand side of the rule is composed of a combination of terminals and nonterminals).

Many different grammar theories and grammar formalisms have been proposed to analyze natural language: Lexical-Functional Grammar (LFG, see [Neidle, 1994; Dalrymple, 1999]), Head-driven Phrase Structure Grammar (HPSG, see [Pollard and Sag, 1994]), Generalized

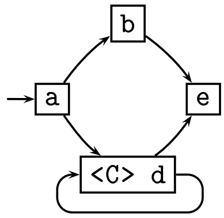


Figure 5.2: Lattice representing the RHS of the JSGF rule `<A> = a ( b | (<C> d)* ) e`. (Cf. Figure 5.3.)

```
<A>         =  a [<_A_aux1>] e;
<_A_aux1>   =  b | <_A_aux2>+;
<_A_aux2>   =  <C> d;
```

Figure 5.3: Rules in the Phoenix formalism equivalent to the single rule `<A> = a ( b | (<C> d)* ) e` in JSGF. Since no grouping of constituents is allowed, two auxiliary NTs (`<_A_aux1>` and `<_A_aux2>`) have to be introduced.

Phrase Structure Grammar, Goverment and Binding (GB, see [Napoli, 1993]), Relational Grammar (see [Pearlmutter, 1983; Pearlmutter and Rosen (eds), 1984]), Link Grammar (see [Sleator and Temperley, 1993]), Categorial Grammar (see [Morrill, 1994; Carpenter, 1998]), and others. SOUP's formalism, however, is only defined in as much as it has to be representable by a context-free grammar. In particular, SOUP accepts grammars written according to the JSGF format (see §1.3 and [JSGF, 1998]).

5.2.1 A Note on the Grouping of RHS Constituents

The JSGF formalism allows to group right-hand side constituents (comprising terminals and/or nonterminals) in order, for example, to repeat or skip them as a whole. This in fact means that a grammar written in the JSGF formalism is a compact representation of a rewrite system that strictly complies with the definition of context-free grammars in Table 5.1.[2] In JSGF, RHSs are no longer linear sequences of symbols but rather constitute *lattices* of symbols. Therefore, if we take a formalism such as Phoenix [Ward, 1994], which does not allow grouping, nonterminals may have to be added. Figures 5.2 and 5.3 exemplify this situation.

[2]And the same holds for the allowance of optional and repeatable constituents.

Arc Type	Explanation
λ_{SEQ}	Beginning of `RuleSequence`
$\lambda_{\mathrm{SEQ}}^{-1}$	End of `RuleSequence`
λ_{ALT}	Beginning of `RuleAlternatives`
$\lambda_{\mathrm{ALT}}^{-1}$	End of `RuleAlternatives`
λ_{CNT}	Beginning of `RuleCount`
$\lambda_{\mathrm{CNT}}^{-1}$	End of `RuleCount`
λ_{TAG}	Beginning of `RuleTag`
$\lambda_{\mathrm{TAG}}^{-1}$	End of `RuleTag`
λ_{FWD}	Forward empty transition
λ_{BWD}	Backward empty transition
λ_{VOID}	`<VOID>` Rule
λ_{NULL}	`<NULL>` Rule
λ_{NT}	Grammar nonterminal (`RuleName`)
λ_{T}	Grammar terminal (`RuleToken`)
λ_{WLD}	Wildcard (to match out-of-vocabulary words)

Table 5.2: Arc types used to encode JSGF grammars as PRTNs.

5.3 Probabilistic Recursive Transition Networks

In SOUP, a context-free grammar, such as a JSGF `RuleGrammar`, is represented via Probabilistic Recursive Transition Networks (PRTNs). Each rule is encoded in a PRTN, where the nodes are either Regular or Final, and the arcs are tuples of <type, ID, probability>.[3] Table 5.2 lists the different types of arcs employed.[4] The probabilities of all the arcs leaving a particular node sum to one. Figure 5.4 shows two PRTNs as examples.

The main advantage of representing a CFG as a collection of PRTNs is the dynamism and flexibility it allows: PRTNs are constructed on the fly as the source text file describing a grammar is being read, and they can be modified at runtime with little effort (which contrasts with the need to recompute the shift-reduce table in some parsers).

In addition, the definition of probabilities at the arc level allows not only to incorporate them into the parse scoring function (see §5.4) but also to generate synthetic data, from which, for instance, a language model can be computed.

[3]In fact, they also contain a boolean to encode activity status.

[4]For expository clarity all λ^{-1} arcs are shown with their subtype (SEQ, ALT, CNT, or TAG) and as a distinct arc, but in the real implementation, there is no need to distinguish, say, between $\lambda_{\mathrm{SEQ}}^{-1}$ and $\lambda_{\mathrm{ALT}}^{-1}$ (a generic λ^{-1} suffices, since the subtype information is obtainable by keeping track of the λ types). Moreover, for efficiency reasons, contiguous λ^{-1} arcs may be collapsed into a single one (with the appropriate marking).

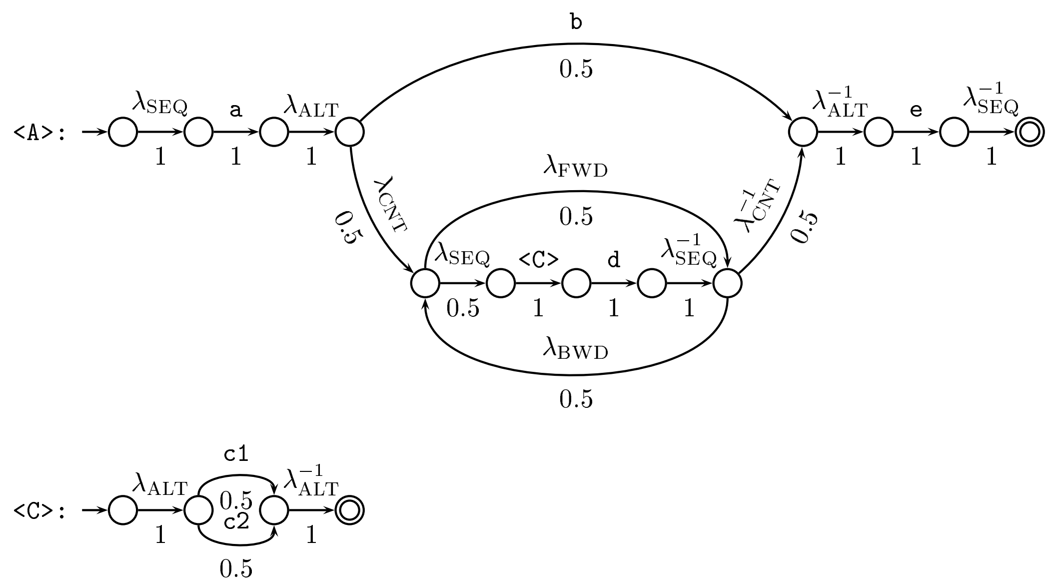


Figure 5.4: Soup's representation of the JSGF rules <A> = a (b | (<C> d)*) e and <C> = c1 | c2 as PRTNs. A PRTN is a directed graph, where the nodes are either Initial (leftmost node), Regular, or Final (denoted by a double circle), and the arcs contain a tuple <type, ID, probability>, where the type is one of the types listed in Table 5.2.

Grammar arc probabilities are initialized to the uniform distribution but can be perturbed by a training corpus of desired (and achievable) parses. Given the direct correspondence between parse trees and grammar arc paths, training the PRTNs is very fast (see §5.8).

5.3.1 Generation from PRTNs

By traversing the PRTNs according to the arc probabilities Soup, can generate not only a corpus of sentences but also a corpus of parse trees (i.e., a parsebank) which can then be used to train Gsg's Prediction Models (see §6.2.4). In the generation function, the following parameters can be selected. (In parentheses are examples of generation from the PRTN in Figure 5.4.)

- **Word vs. subtree generation**: whether the result of the generation is a sequence of terminals (e.g., "a e") or a parse tree (e.g., Figure 5.5).

- **Starting point**: whether to generate (*i*) from the entire grammar, (*ii*) from a given NT, or (*iii*) from a given PRTN node.

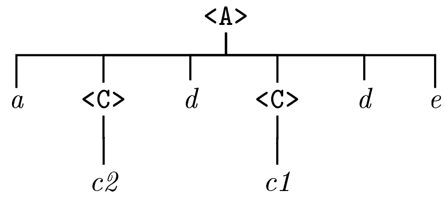


Figure 5.5: Example of a tree generated from the PRTN in Figure 5.4.

- **Treatment of optionals**: whether to always omit optionals (e.g. "`a e`") or make stochastic decisions (e.g. "`a c2 d e`," "`a e`").
- **Treatment of repeatables**: whether to always generate repeatables by a fixed number (e.g., with rep=3, "`a c1 d c2 d c2 d e`") or make stochastic decisions (e.g. "`a c2 d e`").
- **Exhaustive vs. stochastic generation**: whether to generate all possible expansions (e.g., with rep=2, { "`a e`," "`a b e`," "`a c1 d c1 d e`," "`a c1 d c2 d e`," "`a c2 d c1 d e`," "`a c2 d c2 d e`" }) or make stochastic decisions (e.g. "`a b e`," "`a e`," "`a c2 d e`").
- **Number of generations**: how many sentences or trees to generate.

5.4 Parsing Heuristics

As mentioned above, parsing can be seen as a particular instance of a *search* problem. To guide the search, a scoring function is defined to assess the goodness of a particular, possibly partial interpretation. SOUP's scoring function maximizes coverage, minimizes fragmentation, minimizes complexity, minimizes usage of the wildcard, and maximizes arc probabilities. The general rationale behind such heuristics is to favor the simplest, most specific, most informative interpretation, in the technical, information-theoretic sense of *informative*.

Let us look at each of these factors in detail. The highly ambiguous `g3` grammar listed in §A.1 is used to illustrate SOUP's heuristics with concrete examples.

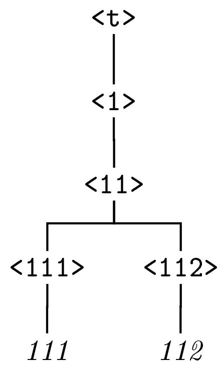


Figure 5.6: Best interpretation of *111 112* according to the `g3` grammar listed in §A.1.

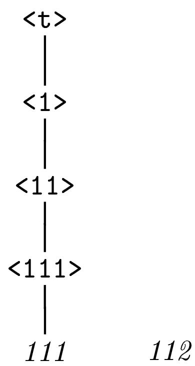


Figure 5.7: Interpretation of *111 112* with less coverage than best in Figure 5.6.

5.4.1 Maximization of Coverage

The most obvious factor is coverage, defined as the ratio of parsed words to the total number of words. For example, the parse of *111 112* with grammar `g3` in Figure 5.6 is preferred to the one in Figure 5.7.

5.4.2 Minimization of Fragmentation

Another factor is fragmentation, defined as the number of parse trees per interpretation. Given two interpretations with the same coverage, the interpretation that consists of fewer non-overlapping parse trees is preferable, as it represents a more succinct explanation of the input sentence.

```
 <t>   <t>
  |     |
 <1>   <1>
  |     |
<11>  <11>
  |     |
<111> <112>
  |     |
 111   112
```

Figure 5.8: Interpretation of *111 112* with same coverage but more fragmentation than best in Figure 5.6.

For example, Figure 5.8 shows another interpretation for *111 112* that has the same coverage as the one in Figure 5.6, but, since it consists of two parse trees, it is more fragmented and therefore dispreferred.

5.4.3 Minimization of Complexity

A third factor is "parse complexity," approximated by the number of parse tree nodes and their branching scores.

5.4.3.1 Minimization of the Number of Nodes

The above factors of coverage and fragmentation being equal, the number of nodes is to be minimized, in this case to attain the simplest explanation. For example, Figure 5.9 shows another interpretation for *111 112* that has the same coverage and fragmentation as the one in Figure 5.6 but has more nodes (six instead of five) and is therefore dispreferred.

5.4.3.2 Maximization of Branching Score

Even when two interpretations have the same coverage, fragmentation and number of nodes, it is still preferrable to select the interpretation with the most specific branching. For example, the parse trees in Figure 5.10 and Figure 5.11 both have the same coverage and number of nodes, but the analysis in Figure 5.10 is preferred because the branching occurs lower in the tree. Formally, the algorithm employed is to compute a *branching score* and then prefer the analysis with highest value. The branching score (bs) for a tree T is defined,

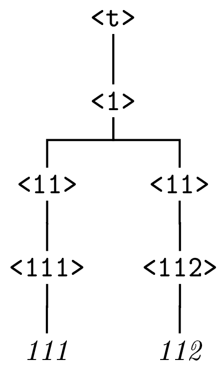


Figure 5.9: Interpretation of *111 112* with same coverage and fragmentation but more nodes than best in Figure 5.6.

recursively, as

$$\mathrm{bs}(T) \quad \stackrel{\text{def}}{=} \quad \left\{ \begin{array}{lcl} 1 & : & Sub(T) = \emptyset \\ \sum_{\forall S : S \in Sub(T)} \left(1 + \left\{ \begin{array}{lcl} 0 & : & \mathrm{bs}(S) \leq 1 \\ \mathrm{bs}(S) & : & \mathrm{bs}(S) > 1 \end{array} \right\} \right) & : & Sub(T) \neq \emptyset \end{array} \right\}$$

where $Sub(T)$ is the set of immediate subtrees of T.

Thus, analyses that group constituents that bond together (i.e., reduce) low in the tree are preferred, as their score will compound up to the root. For example, the branching score for the tree in Figure 5.10 is six and for the one in Figure 5.11 is three.

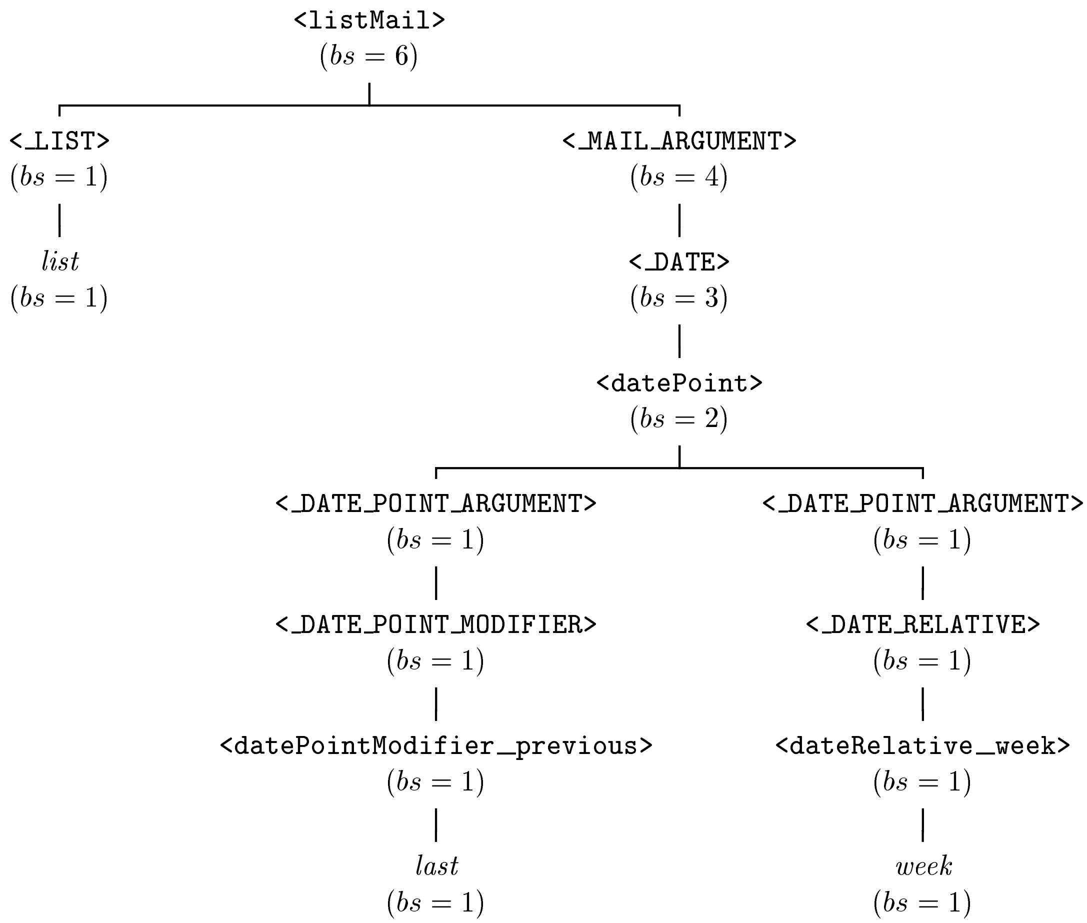


Figure 5.10: Preferred interpretation of *list last week*, as it obtains a branching score (bs) of six (cf. Figure 5.11).

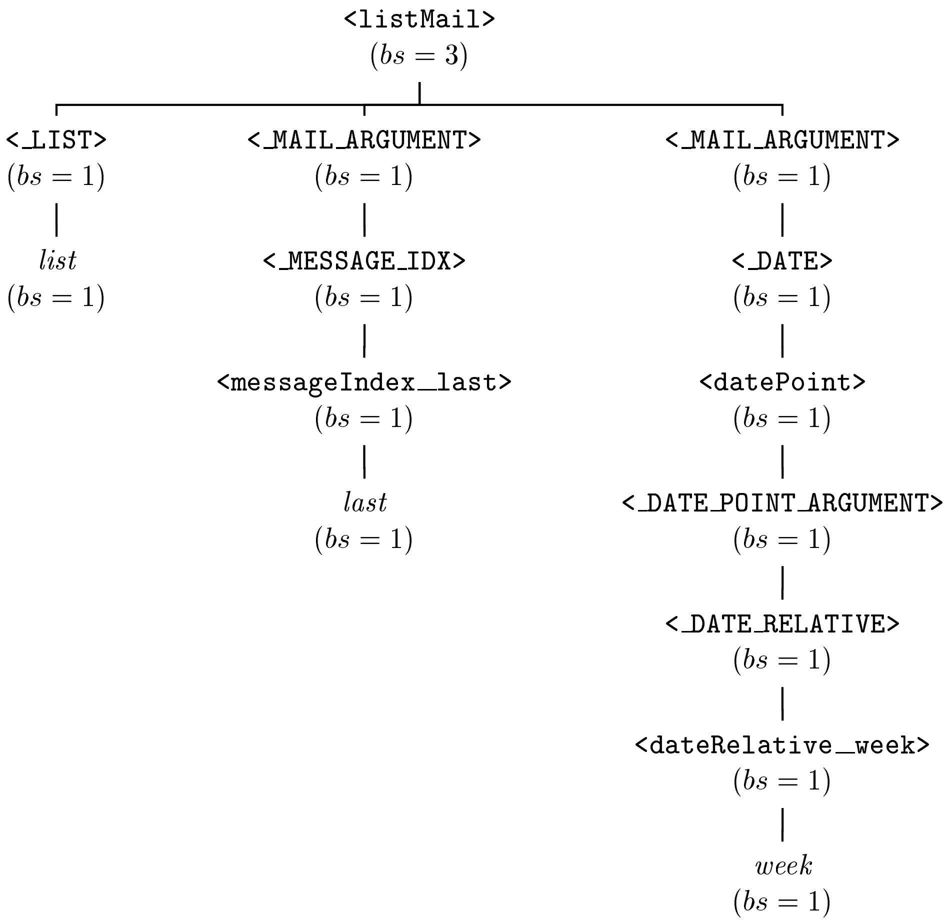


Figure 5.11: Dispreferred interpretation of *list last week*, as it obtains a branching score (bs) of three (cf. Figure 5.10).

```
public <nicety>  =  <greeting> | <farewell>;
<greeting>       =  ciao [<knownPerson> | <unknownPerson>];
<knowPerson>     =  <Maria>;
<Maria>          =  maria;
<unknownPerson>  =  <_WILDCARD>;
```

Figure 5.12: Grammar fragment to illustrate the dispreferral of wildcard usages. See Figures 5.13 and 5.14.

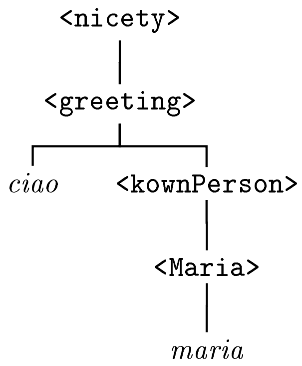


Figure 5.13: Preferred interpretation of *ciao maria* according to the grammar in Figure 5.12 (with a preference for `<greeting>` over `<farewell>`). (Cf. Figure 5.14.)

5.4.4 Minimization of Wildcard Usages

When a parse exists that does not require the usage of the wildcard[5] and has the same coverage, fragmentation, and similar tree complexity as another one that uses the wilcard, the former is preferred. For example, the analysis in Figure 5.13 is preferred over the one in Figure 5.14.

The reasoning behind this heuristic is, again, for parses to be as specific and informative as possible.

5.4.5 Maximization of Arc Probabilities

Finally, the arc probabilities are also taken into account. For example, given the ambiguous grammar in Figure 5.12 and the sentence *ciao*, two parses of identical coverage, fragmentation, branching score and complexity are possible (as depicted in Figures 5.15 and 5.16). If the grammar has not been trained, both trees would also tie in their sum of arc probabili-

[5] A special nonterminal that is able to cover any out-of-vocabulary word (or any in-vocabulary word present in a special set).

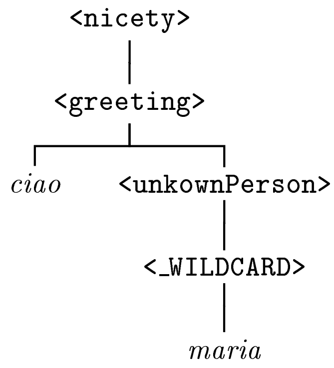


Figure 5.14: Dispreferred interpretation of *ciao maria* according to the grammar in Figure 5.12 and with *maria* present in the set of in-vocabulary words allowed to match the wildcard. (Cf. Figure 5.13.)

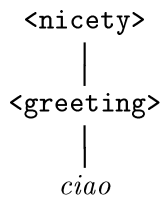


Figure 5.15: One interpretation of *ciao* as `<greeting>` with grammar in Figure 5.12. (Cf. Figure 5.16.)

ties, but if, say, a parsebank with more *ciao*s under `<greeting>` than under `<farewell>` is used to train the grammar, then the parse of *ciao* under `<greeting>` will be preferred.

5.4.6 Combining Heuristics

The heuristics described above are weighted differently in the scoring function. Table 5.3 shows the weight factor for each component in the parse lattice scoring function. These weights were set manually after a short tuning process and are fixed across all grammars. Basically, an order of magnitude separates the weight of each factor, so that they behave hierarchichally (e.g., only if coverage is tied do the rest of the heuristics have an effect).

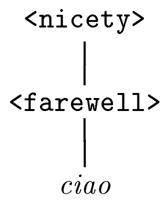


Figure 5.16: Another interpretation of *ciao* as `<farewell>` with grammar in Figure 5.12. (Cf. Figure 5.15.)

Component	Weight
Coverage (number of covered Ts)	+1000
Intra-concept skips (number of skipped Ts)	-100
Wildcards	-20
Fragmentation (number of trees)	-100
Number of principal NTs	-1
Number of auxiliary NTs	-1
Branching score	+1
Sum of arc probabilities	+10

Table 5.3: Weights used in Soup's scoring function of parse lattices. Components with a positive weight will be maximized and components with a negative weight minimized.

5.5 Parsing Modes

The flexibility of SOUP is well exemplified by the variety of parsing modes. Ordinarily, parsing is understood as the assignment of structure to a sequence of words according to a grammar. In the strict sense, a sequence of words constitutes a valid sentence of the language defined by the grammar if and only if there exists a parse tree (sequence of rule rewrites) that, from a starting symbol of the grammar, is able to generate (cover) all the words in the input sequence. Robust parsing however relaxes this constraint by, for example, allowing the input words to be parsed as a sequence of non-overlapping parse trees (i.e., building an interpretation that consists of more than one parse tree), or by allowing the skipping of words. As detailed below, SOUP supports all these different modes, as well as a few truly novel ones, such as the parsing of right-hand sides.

5.5.1 Word-level Default Parsing Mode

The word-level mode is SOUP's typical way of parsing, namely, find an interpretation for a given sequence of terminals.[6] SOUP's robustness adds (*i*) the ability to find interpretations consisting of multiple, non-overlapping parse trees, and (*ii*) the ability to skip input words at any point (i.e., the inter- and intra-concept skipping described in §5.7).

5.5.2 Word-level Constrained Parsing Mode

SOUP's default parsing can be constrained to produce only single-tree interpretations, to make use of speaker side information, and to omit certain NTs.

5.5.2.1 Parsing Constrained to Single-tree Interpretations

In this case, interpretations can only consist of a single tree and the single tree must span (i.e., cover, if no skipping takes place) the entire input sequence. This is the mode specified by JSAPI [JSAPI, 1998]. For example, given the sentence *111 aaa 112* and grammar `g3`, SOUP in default mode would skipp *aaa* and produce the interpretation in Figure 5.17 as its best. In single-tree interpretation mode, however, it would find no interpretation, as there is no legal, single tree that can cover the input sentence.

5.5.2.2 Parsing Constrained to Speaker Side

Nonterminals can be marked to belong to a speaker side only. For example, in a travel reservation domain, there may be a single grammar to parse the utterances of both the

[6] Or multiple interpretations if the sentence is ambiguous.

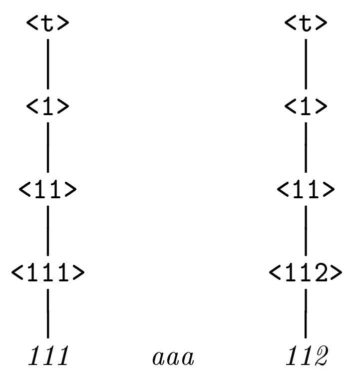


Figure 5.17: Interpretation of *111 aaa 112* allowing for skipping and multiple-tree interpretations.

travel agent and the client, but certain concepts may only make sense for the agent (e.g., `<requestCreditCardInfo>`), or the client (e.g., `<giveRoomPreference>`). Since speaker information (channel source) is readily available at runtime, on a per utterance basis SOUP can disallow the NTs that are exclusive of the other speaker.

5.5.2.3 Parsing Constrained to Given Set of NTs

More generally, the grammar can be partitioned into subsets that can be activated or deactivated for each parse search.

5.5.3 Word-level Augmented Parsing Mode

SOUP's default parsing can be augmented by allowing all NTs to be considered top-level.

5.5.3.1 Parsing Augmented to All-top Mode

In this mode, all NTs are considered top-level, i.e., able to stand at the root of a parse tree. For example, Figure 5.18 shows the result of parsing *ciao maria* under the same conditions as in Figure 5.13 except that in Figure 5.18 all-top parsing mode is set.
Note that the scoring function is the same, therefore coverage maximization, fragmentation and complexity minimization, etc, still apply.
To prevent spurious parses, the wildcard is not allowed to be a root. That is, the special NT `<_WILDCARD>` is only allowed to match an input word if there already exists parsed material to the left of the word in question in the top-level parse lattice currently being pursued. For example, as depicted in Figure 5.19, given the sentence *please retrieve messages from*

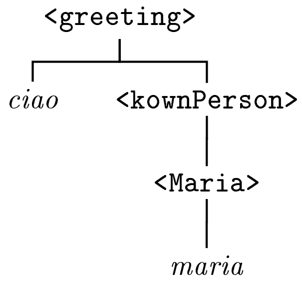


Figure 5.18: Interpretation of *ciao maria* according to the grammar in Figure 5.12 in all-top parsing mode (with a preferece for `<greeting>` over `<farewell>`). Note that the root node is `<greeting>` rather than the more general `<nicety>`.

peter, this restriction has the effect of not parsing out-of-vocabulary word *retrieve* under the wildcard, but doing so for the other out-of-vocabulary word *peter*.

5.5.4 Character-level Parsing Mode

SOUP has the ability to define NTs that operate at the character level. This is useful, for example, for languages with a rich, surface-expressed morphology such as German. Figure 5.20 lists a grammar fragment with rules that operate at the characer level. Figure 5.21 shows an example parse.

Character-level parsing is achieved using the same methods that parse at the word-level. In fact, it is during word-level parsing that character-level parses are spawned by segmenting the current word into its characters and recursively invoking the parse method. The only difference is that, in the search for a character-level parse, the desired root nonterminal is already known and no skipping or multiple-tree interpretations are allowed.

5.5.5 Parsing of Right-hand Sides

Another of SOUP's major parsing modes is the parsing of right-hand sides, that is, of sequences of terminals and non-terminals. This is used to detect the introduction of ambiguity in the grammar, as well as to compute subsumption of right-hand sides. This mode represents a generalization of the usual parsing of terminals, for, in this case, the input vector is constructed as a vector of type–ID pairs (where type is either in-vocabulary terminal (T), out-of-vocabulary terminal (OOV), or nonterminal (NT)). Since parsing in SOUP is finding a path along the PRTNs that best covers the input vector, the algorithm for parsing RHSs is the same as the one for parsing Ts: when the element to match in the input vector is of

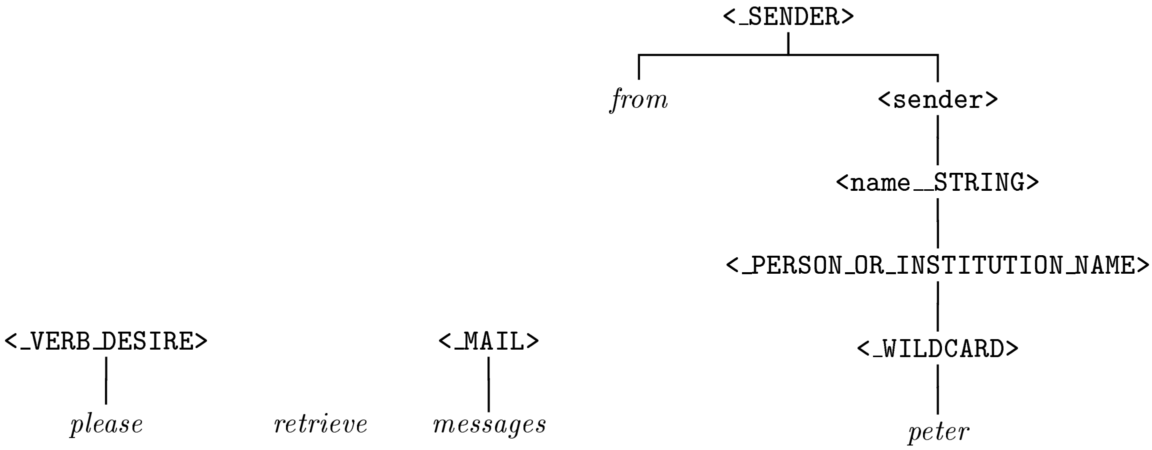


Figure 5.19: Interpretation of *please retrieve messages from peter* according to the E-Mail Task grammar (see §B.3.qq) in all-top parsing mode. Note that in all-top mode the `<_WILDCARD>` is not allowed as the first (leftmost) daughter of a parse tree (cf. unparsed *retrieve* vs. parsed *peter*).

type NT, a corresponding arc of type NT has to be found in the PRTNs, as is the usual case for Ts.

5.5.5.1 Generation and Expansion Sets of Right-hand Sides

An important distinction is made between the *generation* and the *expansion* sets of a right-hand side. Given an RHS R, its generation set $\mathrm{Gen}(R)$ is defined as the sublanguage in Σ^* defined by R, i.e., the exhaustive listing of all sequences of Ts permissible under R via the recursive expansion of NTs until all possible Ts are reached. Obviously, in the case of infinitely-repeatible blocks ('`*`' and '`+`' operators in JSGF) an approximation has to be made, e.g., by repeating them five times. Figure 5.22 shows an example of the generation set for an RHS.

On the other hand, given an RHS R, its expansion set $\mathrm{Exp}(R)$, is defined as the sublanguage in $(\Sigma \cup V)^*$ defined by R, i.e., the exhaustive listing of all sequences of Ts and NTs permissible immediately under R. Figure 5.23 gives an example of the expansion set for the same RHS as the one in Figure 5.22.

5.5.5.2 Detection of Ambiguity Introduction

A key feature of SOUP is its ability to dynamically modify the grammar, as when new rules are acquired. It is important, however, not to disrupt the exisiting grammar when new

```
public <requestRoom>  =  <_VERB_DESIRE> [ <_C_roomQuantity>]
                         <_C_roomFeature>* <_ROOM>;
<_VERB_DESIRE>        = ich ( moechte | will | haette gern);
<_C_roomQuantity>     =  (     ( ein         {1})
                         // | ...
                         )
                         <_c_ENDING>;
<_C_roomFeature>      =  (     ( klein       {small})
                            | ( gemuetlich  {cozy})
                         // | ...
                         )
                         <_c_ENDING>;
<_c_ENDING>           =  <NULL> | er | e | es | em | en;
<_ROOM>               =  zimmer | stube;
```

Figure 5.20: Grammar fragment to illustrate character-level NTs. NTs whose name starts with `_C_` are principal NTs that operate at the character level. NTs whose name starts with `_c_` are auxiliary NTs that operate at the character level. Note the factorization of adjectival endings via `<_c_ENDING>`, thereby preëmpting a six-fold increase in the number of terminals representing adjectives. See Figure 5.21 for an example parse.

rules are added. For example, it is not a good idea to introduce ambiguity. How can the introduction of ambiguity be detected? A first approach would be, given a new right-hand side R, to add it to the grammar, compute $\mathrm{Gen}(R)$, and see whether any $s \in \mathrm{Gen}(R)$ is parsable by the current grammar. If that is the case, it means that the new RHS R does introduce ambiguity. But this algorithm is quite inefficient, as the exhaustive construction of $\mathrm{Gen}(R)$ may give rise to a very large set. But upon the realization that precisely most of the exponential growth of $\mathrm{Gen}(R)$ is due to the expansion of its nonterminals, the time complexity of the ambiguity detection algorithm can be reduced by computing $\mathrm{Exp}(R)$ instead.

Figure 5.24 shows the existence of a parse for `<greeting>` according to the grammar in Figure 5.12, indicating that adding an RHS that has `<greeting>` as a member of its expansion (such as `<greeting> [again]`), would introduce (or increase) the ambiguity in the grammar.

5.5.5.3 Detection of Rule Subsumption

Similarly, Soup's parsing of RHSs can be used to detect rule subsumption, i.e., when the language of one RHS is already contained in the language of an NT. For example, the RHS `a b b` is subsumed by the rule `<r> = c | [a] b+` because $L($`a b b`$) = \{$ "`a b b`" $\}$ is contained in $L($`<r>`$) = L($`c | [a] b+`$) = \{$ "`c`", "`b`", "`a b`", "`b b`", "`a b b`", "`b b b`", "`a b b b`", ...$\}$.

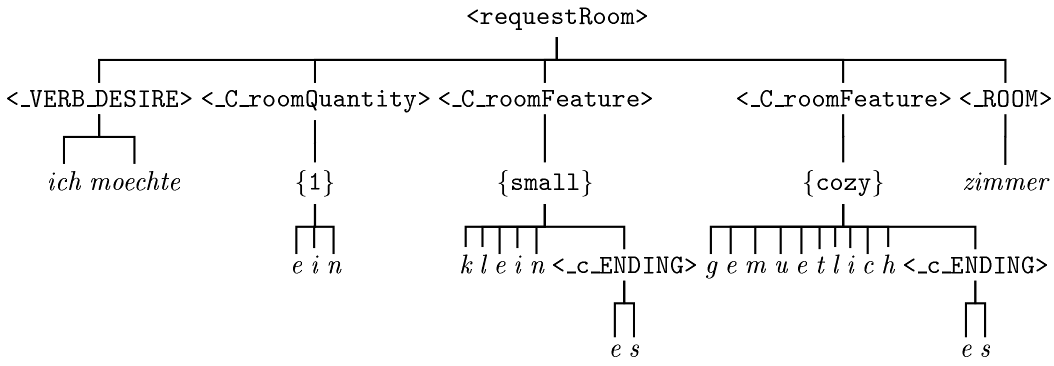


Figure 5.21: Parse of *ich möchte ein kleines gemütliches zimmer* with the grammar in Figure 5.20 to exemplify character-level parsing.

In this case, given RHS R_1 and RHS R_2, R_1 is subsumed by R_2 iff $L(R_1) \subseteq L(R_2)$ iff $\forall s \in$ Exp(R_1) $\exists$P(s,`<r>`), where $P(s,$`<r>`$)$ is a non-empty parse of sequence s under NT `<r>`. In other words, if all expansions of the new RHS obtain a parse under a certain NT, that NT is already covering the language of the new RHS and therefore it is not necessary to add the new RHS to the grammar.

For example, if the grammar already contains the rule `<farewell> = [good] bye+`, a new RHS alternative `bye bye` under `<farewell>` would be redundant.

```
 (1)  a a a a a b1 c d e e e e e
 (2)  a a a a a b1 e e e e e
 (3)  a a a a a c d e e e e e
 (4)  a a a a a e e e e e
 (5)  b1 c d e e e e e
 (6)  b1 e e e e e
 (7)  c d e e e e e
 (8)  e e e e e
 (9)  a a a a a b2 c d e e e e e
(10)  a a a a a b2 e e e e e
(11)  b2 c d e e e e e
(12)  b2 e e e e e
(13)  a a a a a b2 b3 c d e e e e e
(14)  a a a a a b2 b3 e e e e e
(15)  b2 b3 c d e e e e e
(16)  b2 b3 e e e e e
```

Figure 5.22: The Gen(a* [<b>] [c d] e+) set, with <b> = b1 | b2 [b3]. Infinitely repeatable constituents are approximated by five copies. (Cf. Figure 5.23.)

```
(1)  a a a a a <b> c d e e e e e
(2)  a a a a a <b> e e e e e
(3)  a a a a a c d e e e e e
(4)  a a a a a e e e e e
(5)  <b> c d e e e e e
(6)  <b> e e e e e
(7)  c d e e e e e
(8)  e e e e e
```

Figure 5.23: The Exp(a* [<b>] [c d] e+) set. Infinitely repeatable constituents are approximated by five copies. (Cf. Figure 5.22.)

```
 <nicety>
    |
<greeting>
```

Figure 5.24: Example of detection of ambiguity introduction through parsing of right-hand sides: The existence of a parse for <greeting> under <nicety> according to the grammar in Figure 5.12 means that adding, say, <nicety> $\longrightarrow$ <greeting> [again] would increase the ambiguity of the grammar.

```
public <time>  =  <point>+;
<point>        =  <hour> | <minute> | <second>;
<hour>         =  one | two;
<minute>       =  one | two;
<second>       =  one | two;
```

Figure 5.25: Grammar to illustrate ambiguity packing. (See Figure 5.26.)

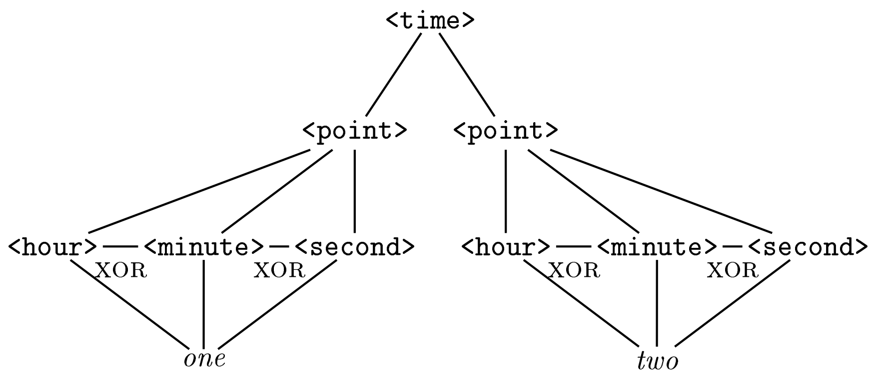


Figure 5.26: Parse lattice for input *one two* according to the grammar in Figure 5.25. It encodes the $3^2 = 9$ different parse trees in Figure 5.27.

5.6 Sketch of the Parsing Algorithm

In Soup, parsing proceeds in the following steps: processing of the input string, population of the chart, and search for the best analysis.

1. *Construction of the input vector.* Given an utterance to be parsed, it is converted into a vector of type–ID pairs.[7] In the usual mode of parsing terminals all elements in the input vector will be of type T (in-vocabulary terminal) or OOV (out-of-vocabulary terminal). In the case of parsing RHSs (see §5.5.5 above) some elements may be of type NT.

   Special terminals `<s>` and `</s>` are added at the beginning and end of an utterance, respectively, so that certain rules only match at those positions. (For example, the rule `<accept> = <s> ok </s>` will only match if *ok* constitutes the entire utterance.)

   Also, global search-and-replace string pairs defined in the grammar[8] are applied, e.g., to expand contractions (as in *I'd like* $\rightarrow$ *I would like*) or to remove punctuation marks.

[7] If the input is an array of tokens, no segmentation is needed; otherwise the input string has to be segmented into tokens.

[8] This is an extension of JSGF: string mappings are defined via keyword `@stringMap` in the `JavaDoc` comment. See grammars in §A, §B, and §C for examples.

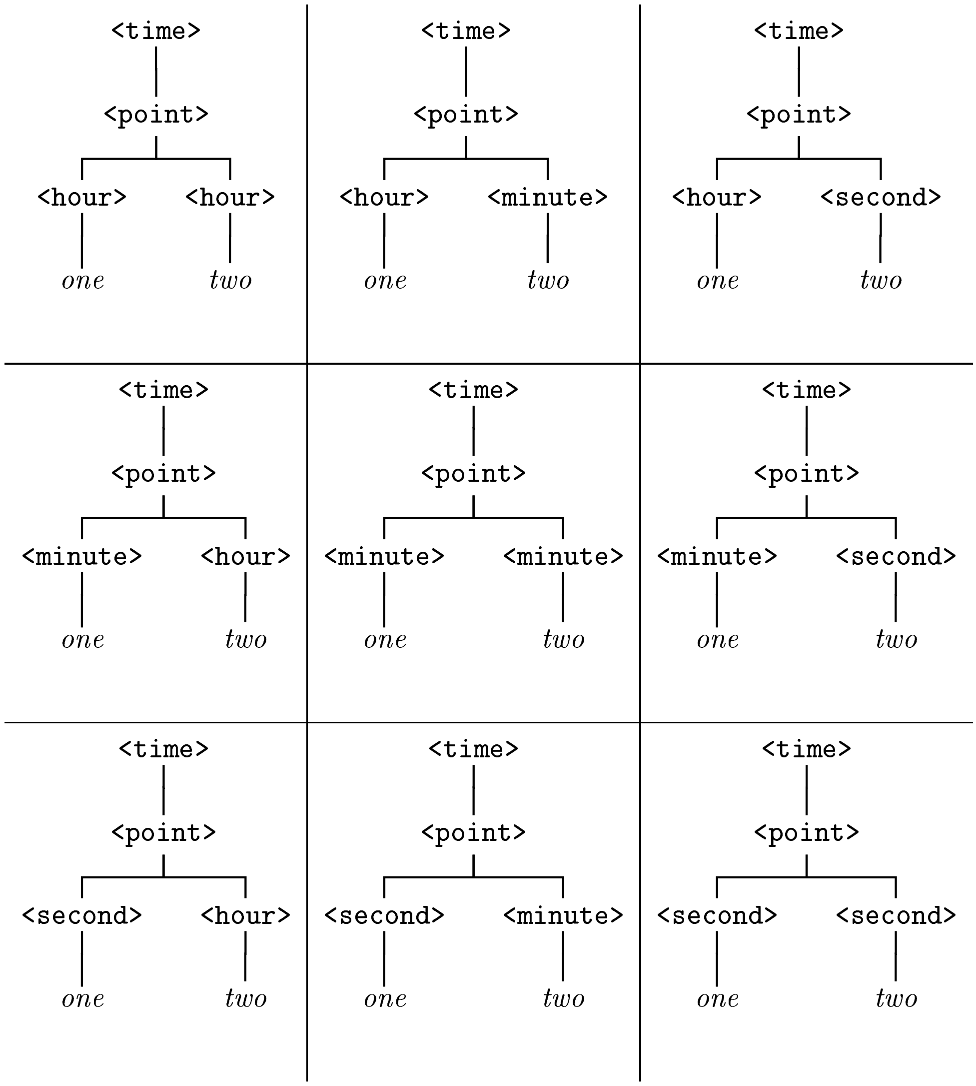


Figure 5.27: Exhaustive listing of the parse trees encoded in the parse lattice in Figure 5.26.

Other settings determine whether out-of-vocabulary words should be removed, and whether the input utterance is case-sensitive.

2. *Population of the chart.* The first search populates the chart (a two-dimensional table indexed by input-word position and nonterminal ID) with parse lattices. A parse lattice is a compact representation of a set of parse trees (similar to Tomita's shared-packed forest [Tomita, 1987]). Figures 5.25, 5.26, and 5.27 provide an example.

   The beam search involves top-down, recursive matching of PRTNs against the input vector. All top-level nonterminals starting at all input vector positions are attempted. The advantage of the chart is that it stores, in an efficient way, all subparse lattices found so far, so that subsequent search episodes can reuse existing subparse lattices. More concretely, for each nonterminal and input word position the chart stores the corresponding list of parse lattices ordered by score (computed from the scoring function defined in §5.4).

To increase the efficiency of the top-down search, the set of terminals with which a nonterminal is allowed to start is precomputed (i.e., the First set), so that many attempts to match a particular nonterminal at a particular input vector position can be preëmpted by the lack of the corresponding terminal in its First set. This top-down filtering technique (also used, for example, in the Tina parser [Seneff, 1992]) typically results in a threefold speedup.

The beam serves to restrict the number of possible subparse lattices under a certain nonterminal and starting at a certain input position, e.g., by only keeping those subparse lattices whose score is at least 30% of the best score. As described in §6.4, the score function is such that (*i*) coverage (number of words parsed), (*ii*) branching score, and (*iii*) sum of arc probabilities are maximized, whereas (*iv*) parse lattice complexity (approximated by number of nonterminals) and (*v*) usages of the wildcard (approximated by the maximal number of arcs encoding the out-of-vocabulary wildcard along the parse lattice) are minimized. Also, pruning of structurally-equal parse lattices is performed, thereby eliminating the redundancy that arises from several right-hand sides matching the same input vector span under the same nonterminal.[9]

3. *Finding the best interpretations.* Once the chart is populated, a second beam search finds the best N interpretations, i.e., the best N sequences of top-level, non-overlapping parse lattices that cover the input vector. Scoring of interpretations adds, to the above scoring function, a sixth factor, namely the minimization of parse fragmentation (number of parse trees per utterance). This search problem can be divided into subproblems (instance of a so-called divide-and-conquer strategy) since both unparsed words and words parsed by a single parse lattice offer a natural boundary to the general problem.[10] For example, Figure 5.28 shows how sentence segments can be covered by more than one top-level NT. Then a beam search is conducted for each subproblem. In this case, the beam limits the number of active sequences of

[9] This can happen, for instance, when the grammar is inherently redundant as in, say, <r> = a [b] | a [c] instead of <r> = a [b | c]. Then, presented with input *a* two seemingly identical parses would be possible. However, with Soup's pruning technique, only one would be kept alive. (Of course, rather than performing this pruning at parse time, Soup's subsumption detection capabilities (see §5.5.5.3) could be used a priori to rid the grammar of such redundancies.)

[10] Note that such boundaries would not exist if ordering of top-level NTs were to be taken into account in the scoring function (e.g., via a bigram of top-level NTs).

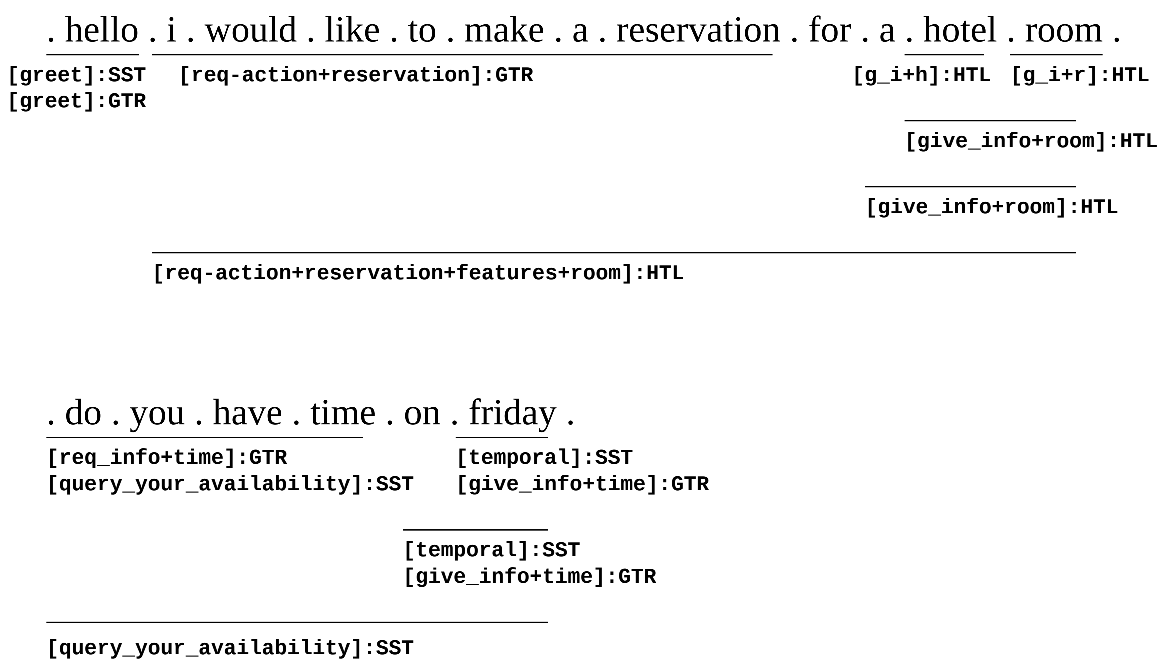


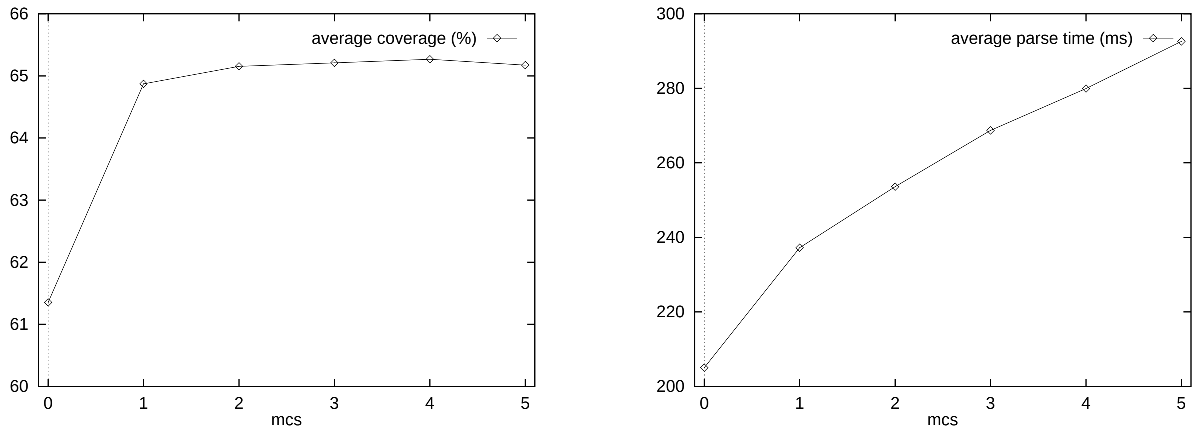
Figure 5.28: Example, in a travel domain, of second order search (see Step 3 in §5.6). For example, the word *room* is covered by four different top-level NTs: three times by `<HTL.give-information+room>` (by itself, as part of *hotel room*, and as part of *a hotel room*), and once by `<HTL.request-action+reservation+features+room>` (as part of *i would like to make a reservation for a hotel room*). The second order search decides which sequence of non-overlapping parse trees best accounts for the input words. In this case it finds `<greet:GTR>`, `<HTL.request-action+reservation+features+room>`, and `<SST.query_your_availability>` as they minimize overall fragmentation.

top-level, non-overlapping parse lattices that form a partial interpretation. Since the single best interpretation is simply the concatenation of the best sequence of each subproblem, even when asked to compute the top N interpretations for $N > 1$, the best interpretation is always computed separately and output immediately so that backend processing can begin without delay.

The final result is a ranked list of N interpretations, where the parse lattices have been expanded into parse trees.

5.7 Skipping

Given the nature of spoken language it is not realistic to assume that a grammar can ever be complete in the sense of covering all possible surface forms. In fact, it turns out that a



(**a**) (**b**)

Figure 5.29: (**a**) Average coverage and (**b**) parse times for different values of *mcs* (maximal number of contiguous words that can be skipped within a nonterminal). Same test set and machine as in Table 5.4 (see §5.8) but with the Travel grammar only.

substantial portion of parse errors comes from unexpected insertions, e.g., adverbs that can appear almost anywhere.

Soup is able to skip words both between top-level nonterminals (inter-concept skipping) and inside any nonterminal (intra-concept skipping). Inter-concept skipping is achieved by the second-level search just described in Step 3 of §5.6 (i.e., the search that finds the best interpretation as a sequence of non-overlapping parse lattices), since an interpretation may naturally contain gaps between top-level parse lattices. Intra-concept skipping, on the other hand, occurs during the first search step, by allowing, with a penalty, insertions of input words at any point in the matching of a PRTN. The resulting exponential growth of parse lattices is contained by the beam search. A word-dependent penalty (e.g., one based on word saliency for the task at hand) can be provided, but the experiments reported in this dissertation use a uniform penalty together with a list of non-skippable words (typically containing, for example, the highly informative adverb *not*). The parameter *mcs* regulates the maximal number of contiguous words that can be skipped within a nonterminal. Figure 5.29 plots coverage and parse times for different values of *mcs*. These results are encouraging as they demonstrate that coverage lost by skipping words is offset (up to *mcs* = 4) by the ability to match longer sequences of words.

	Scheduling	Travel + Scheduling
NTs	600	6,963
Top-level NTs	21	480
Ts	831	9,640
Rules	2,880	25,746
Nodes	9,853	91,264
Arcs	9,866	97,807
Avg. cardinality of FIRST sets (Ts)	44.48	240.31
Grammar creation time (ms)	143	3,731
Training time (ms/tree)	0.452	0.765
Memory (MB)	<2	<14
Avg. parse time (ms)	10.09	228.99
Max. parse time (ms)	53	1070
Avg. coverage	85.52%	88.64%
Avg. fragmentation (trees/utt)	1.53	1.97

Table 5.4: Grammar measurements and performance results of parsing 606 naturally-occurring utterance in a scheduling domain, with an average length of 9.08 words. The first column lists results for the scheduling grammar; the second for the travel grammar (which includes the scheduling grammar). Tests were run on a 266-MHz Pentium II running Linux.

5.8 Performance

A guiding principle in the design and implementation of SOUP is that of efficiency, since real-world, real-time applications are to be supported. Encoding the grammars as PRTNs and conducting the first- and second-order beam searches described above result in very satisfactory parse times.

The upper portion of Table 5.4 lists some parameters that characterize the complexity of two grammars, one for a scheduling domain and the other for a travel domain (plus the same scheduling domain); the lower portion lists performance results of parsing a subset of transcriptions from the English Spontaneous Speech Scheduling Task corpus (briefly described in [Waibel et al., 1996]).

Parsing time increases substantially from a 600-nonterminal, 2,880-rule grammar to a 6,963-nonterminal, 25,746-rule grammar but it is still well under real-time. In addition, as depicted in Figure 5.30, although worst-case complexity for chart parsing is cubic on the number of words, SOUP's parse time appears to increase only linearly. Such behavior, similar to the findings reported in [Slocum, 1981], is due, in part, to SOUP's ability to segment the input utterance into parsable chunks (i.e., finding multiple-tree interpretations) during the search process.

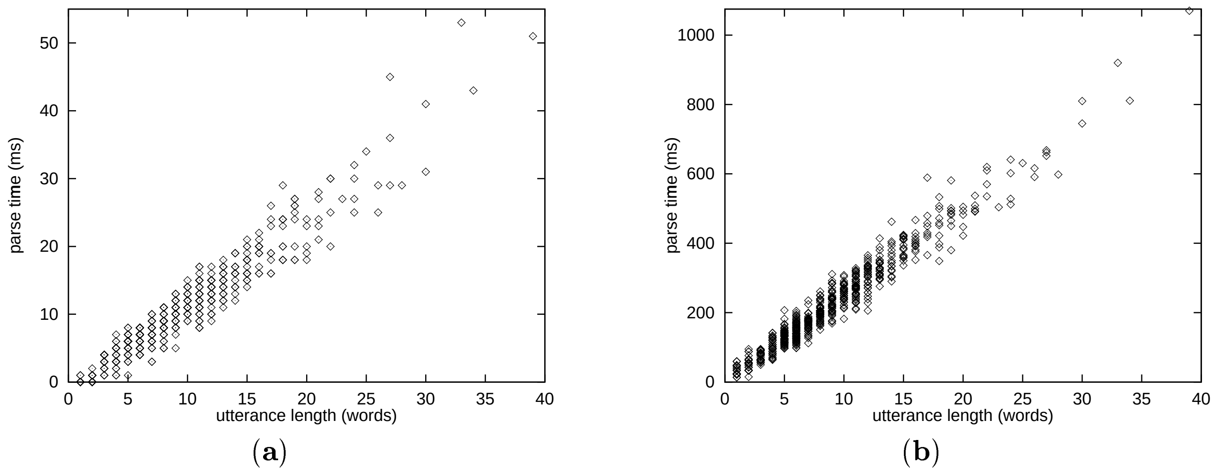


Figure 5.30: Utterance length vs. parse time for (**a**) Scheduling grammar and (**b**) Scheduling plus Travel grammar. Same test and machine as in Table 5.4. Parse time appears to increase only linearly with regards to utterance length.

With these results, even though comparisons of parsers using different grammar formalisms are not well-defined, Soup appears to be faster than other "fast parsers" described in the literature (cf., for example, [Rayner and Carter, 1996] or [Kiefer and Krieger, 1998]).

5.8.1 Comparsion with LCFlex

Still, in order to have a concrete, external reference point, an experiment was conducted to compare parse times of Soup against the LCFlex parser [Rosé and Lavie, 1999] on a few domains.

The LCFlex parser, a descendant of GLR* [Lavie, 1996], is a left-corner parser with many robustness features, such as skipping, insertion of terminals, and flexible feature unification, as well as significant performance improvements over GLR* (see [Rosé and Lavie, 1999] for details).

Table 5.5 quantifies the grammars and test sets used in the comparison, and Table 5.6 presents the results. The grammars were translated into LCFlex's formalism and the LCFlex tests were conducted by Carolyn Penstein Rosé, the designer and implementer of LCFlex, to whom the author is indebted.

With the strong caveat that comparisons of parsers that are designed for different kinds of grammars are never quite commensurable, it is worth noting that Soup is between 10 and 367 times faster in average parse time, and between 79 and 1826 times faster in maximal parse time. It has to be stressed that the grammars used in the comparison were not optimized for LCFlex (e.g., do not use feature unification) and that LCFlex is

	E-Mail	Scheduling	Scheduling and Travel
NTs	382	600	6,963
Ts	469	831	9,640
Rules	947	2,880	25,746
Avg. card. of FIRST sets (Ts)	16.46	44.48	240.31
Utts in test set	318	606	3438*
Avg. length of utts in test set	6.9	9.1	6.9

Table 5.5: Grammars and test sets used in comparison experiment of Table 5.6. *Only first 200 in LCFLEX.

	E-Mail		Scheduling		Scheduling and Travel	
	LCFLEX	SOUP	LCFLEX	SOUP	LCFLEX	SOUP
Avg. parse time (ms)	32	3	570	10	84,150	229
Max. parse time (ms)	5,480	13	4,190	53	1,112,590	1070

Table 5.6: Comparison of LCFLEX and SOUP parse times on grammars and test sets listed in Table 5.5.

implemented in Lisp, whereas SOUP is written in C++ and Java, which may account for an order of magnitude.[11] Still, the results do show that for a pure-CFG formalism SOUP offers a very good performance.

5.9 The Graphical Development Environment GSOUP

Finally, to conclude this chapter on SOUP, what follows is a cursory presentation of GSOUP, a graphical development environment for the construction, editing and testing of grammars also built by the author. GSOUP provides full-fledged editing capabilities, as well as the more advanced features listed below.

- Logical zoom on the grammar.
  - High-level view: Graphical depiction of the NTs and their relations in the form of a graph (see §6.2.2).
  - Low-level view: Graphical depiction of individual grammar rules, down to their probability.
- Logical zoom on the corpus of sentences to parse.

[11] Estimated from the fact that the KANT project's C++ implementation of GLR was about an order of magnitude faster than the Lisp implementation. (Carolyn Penstein Rosé, personal communication.)

  - High-level view: Graphical depiction of parsed/unparsed words as color-coded bullets to quickly spot areas of extragrammaticality.
  - Low-level view: Graphical depiction of parse trees.

- Automatic generation from NTs.
- Automatic detection of ambiguity introduction when a rule is added or modified.
- Automatic detection of changes in a control or regression parsebank when a rule is added or modified.
- Quick switch between initial grammar and extended grammar (initial grammar plus the new rules acquired in the current session) for easy comparison.
- Automatic annotation of rules: New rules are annotated with author, date, and the previously extragrammatical sentence that triggered their construction.

Generally, all graphical objects are clickable and manipulable, e.g., parse tree operations are provided (such as sever branch, attach branch, rename node, etc) for interactive construction of parse trees. Also, changes are dynamically propagated, and feedback is immediate.
Figures 5.31 to 5.35 show screenshots of GSOUP.

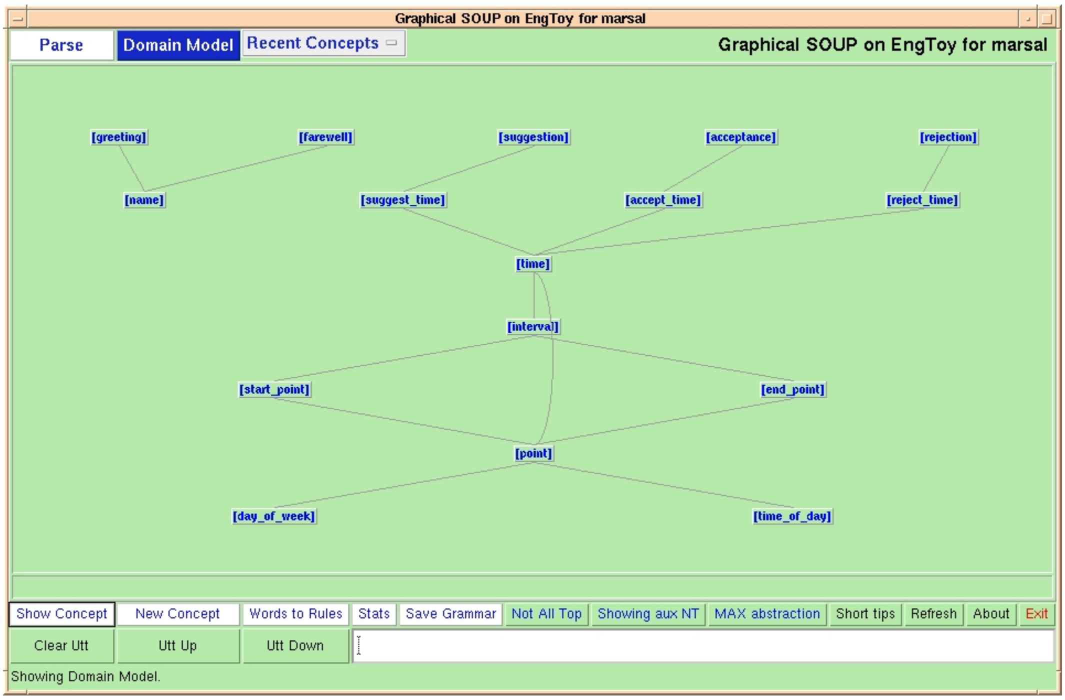


Figure 5.31: Screenshot of GSOUP: Visualization of the ontology.

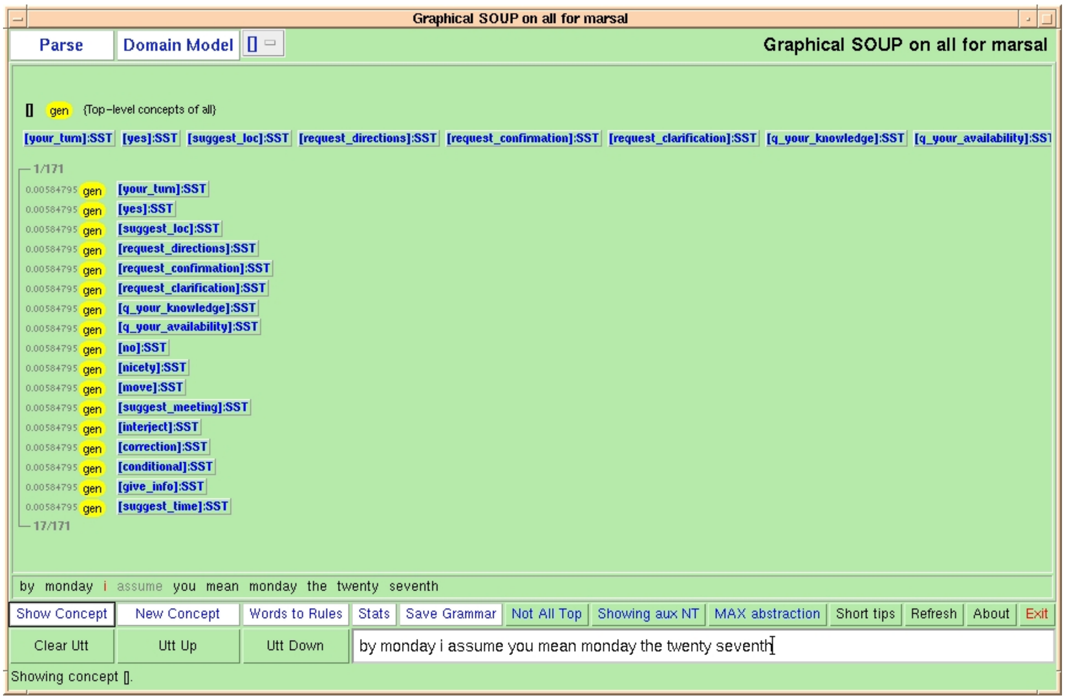


Figure 5.32: Screenshot of GSOUP: Visualization of top-level NTs.

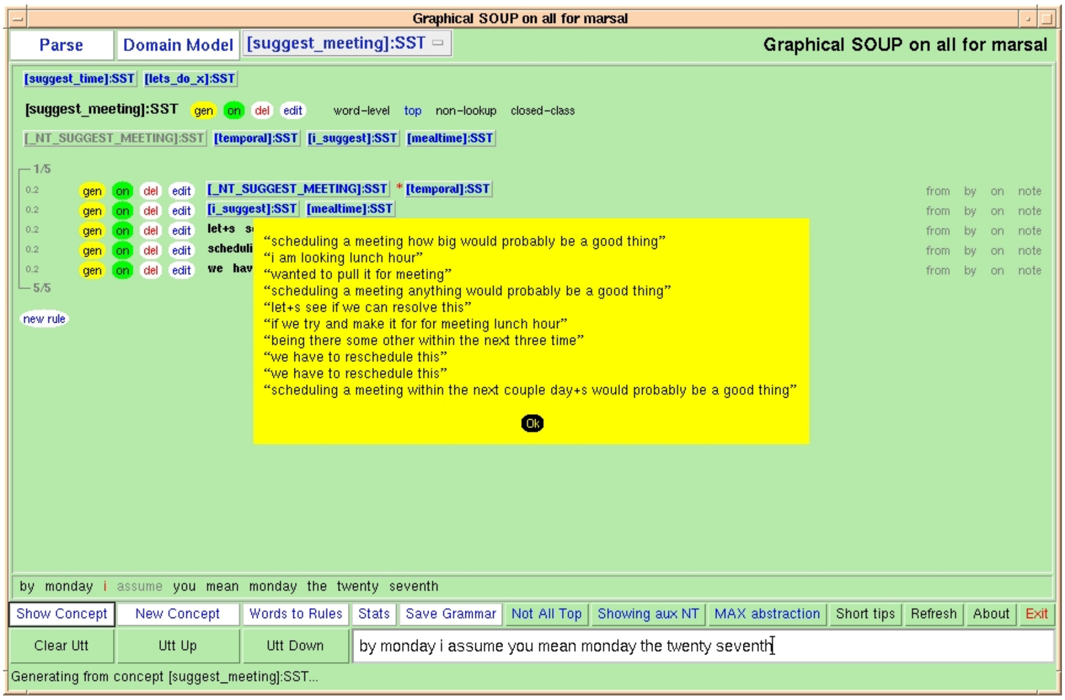


Figure 5.33: Screenshot of GSOUP: Generation from an NT.

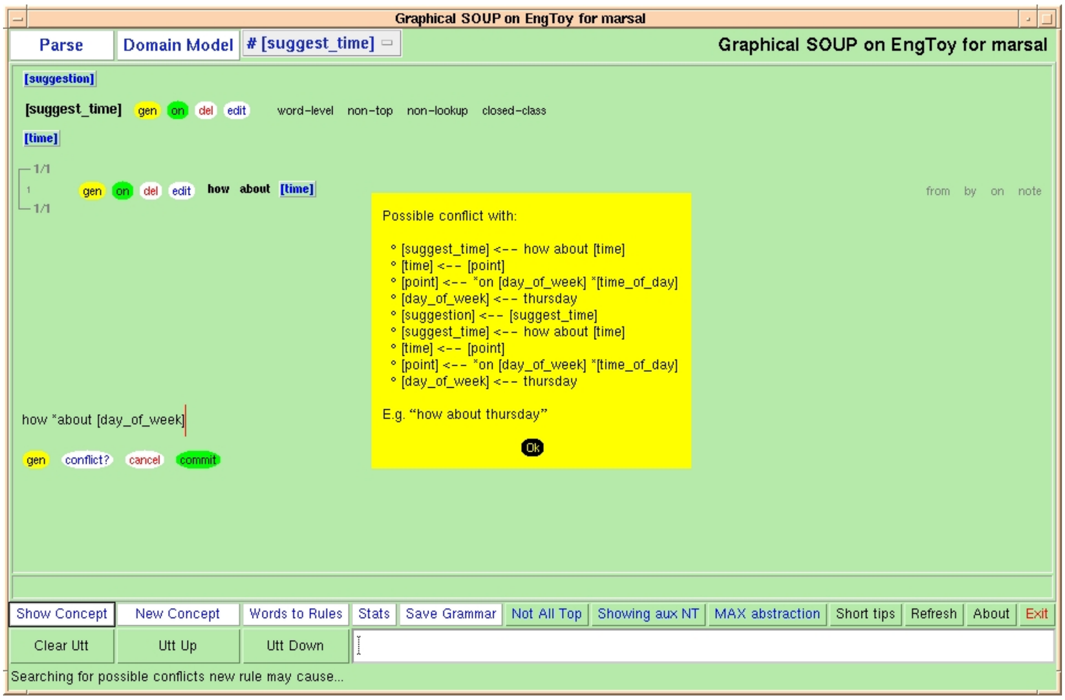


Figure 5.34: Screenshot of GSOUP: Detection of rule conflicts.

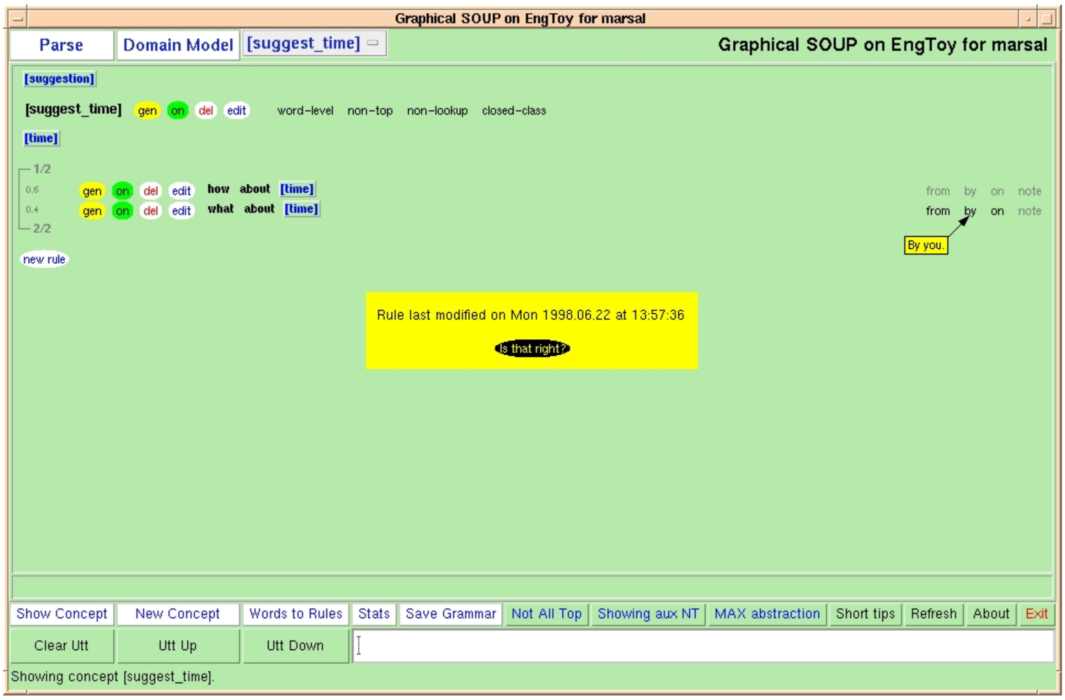


Figure 5.35: Screenshot of GSOUP: Rule annotation.

Chapter 6

The GSG Learning System

This chapter provides a detailed view of all of GSG's components, revisits GSG's fundamental algorithms for interactive parse construction and rule management first outlined in Chapter 4, and ends with a full example of its inner workings.

6.1 System Architecture

Figure 6.1 depicts the overall architecture of the system. The foundation of the system is the SOUP parser, just described in Chapter 5. On top of it lie the core components of the learning machinery: the Dialogue Manager, GSG Engine, and Parse Tree Builder, which are aided in their operation by the principal knowledge sources: the Ontology, Prediction Models and Interaction History, and ultimately driven by the user.

What follows is, first, a detailed explanation of each of the components, including the knowledge sources (§6.2) and the fundamental algorithms (§6.3), and then illustrative examples of GSG in action (§6.4).

6.2 Knowlege Sources

The models, hypotheses and strategies that GSG employs when postulating meanings and formulating clarification questions are based on a variety of knowledge sources that ultimately derive from the Kernel Grammar (i.e., the task grammar in its initial form). The external knowledge sources (ovals outside the dashed line in Figure 6.1) are given to GSG. Note that the only required external knowledge source is the Kernel Grammar;[1] the remaining external knowledge sources (Kernel Parsebank, User Grammar Δ, and Syntactic Grammar) are optional. Finally, the internal or derived knowledge sources are: the Grammar, Ontology, Parsebank, Prediction Models, and Interaction History.

[1] And the Metagrammar, as discussed in §4.2.1.

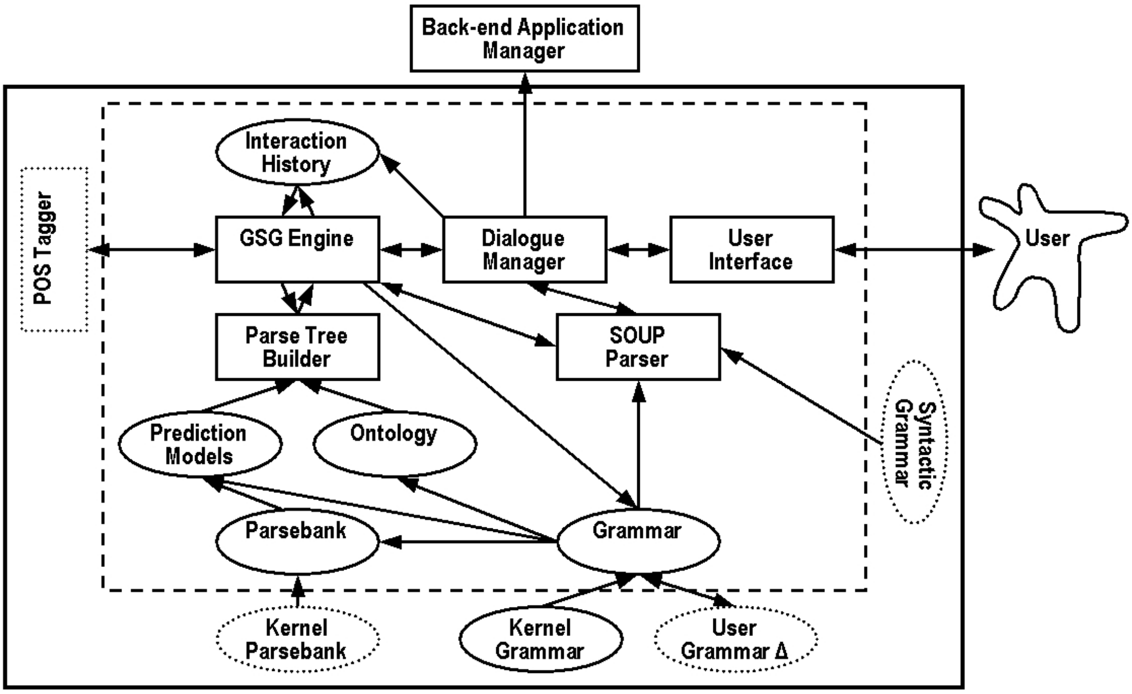


Figure 6.1: GSG's system diagram. Ovals enclose knowledge sources, rectangles enclose modules, and arrows indicate information flow. Dotted components are optional. Note that, optionally, information may also flow from the Back-end Application Manager to the Dialogue Manager, as in the Musicbox Task where information of the current state of the end-application is used by the GSG Engine to filter out hypotheses.

6.2.1 Grammar

As shown in Figure 6.1, the Grammar is the Kernel Grammar possibly augmented with a User Grammar Δ containing user-specific rules acquired in previous sessions. In any case, it is a context-free, semantic grammar such as the ones listed in §C.2 and §C.3.

6.2.2 Ontology

The first knowledge source that is automatically computed from the Grammar is the Ontology. The Ontology encodes the nature and inter-relations of the concepts in the domain, and is often called upon to provide guidance during the learning episodes.

More concretely, the Ontology is a directed acyclic graph in which the nodes correspond to grammar nonterminals and the arcs record immediate dominance relations. That is, the presence of, say, NT_i in the right-hand side of NT_j will result in an arc from NT_i to NT_j. The nature of the ontological nodes is determined as follows.

- **Principal** vs. **Auxiliary**. A distinction is made between principal and auxilary NTs. Principal NTs represent the truly relevant concepts in the domain (such as <composeMail> or <messageIndex> in the E-Mail Task) whereas auxiliary NTs represent useful groupings, that tend to be either syntactic in nature (such as <_VERB_DESIRE>, which covers *I want*, *I wish*, *I would like*, etc), or encode semantic equivalences (such as <_DELETE>, which covers *delete*, *remove*, *destroy*, etc). Auxiliary NTs are typically removed from the final parse tree, e.g., before a feature structure is computed. Note, however, that principal and auxilary NTs can be interleaved (i.e., a principal NT can appear under an auxiliary NT and viceversa) and that the parsing algorithm does not treat them differently. A simple naming convention is used to distinguish auxiliary from principal NTs: an NT is auxiliary if and only if its name[2] starts with an underscore '_'.

- **Top-level** vs. **Non-top** level. Top-level concepts[3] are the starting symbols of the grammar, i.e., the NTs able to stand at the root of a parse tree. A further distinction is made between *topologically* and *logically* top-level nodes. A node is topologically top-level if and only if it never appears as the daughter of another, whereas a node is logically top-level if and only if it is designated as such[4] by the grammar developer. For example an NT <time> could be logically top-level as designated by the grammar writer but at the same time appear under, say, <suggestMeeting>, in which case it would not be topologically top-level. Thus topological top-levelness implies logical top-levelness,[5] but not the other way around.

- **Pre-NT** vs. **Pre-T** vs. **Mixed**. Depending on the nature of the immediate daughters, a node is classified as Pre-NT if all its daughters are NT or all T daughters are Always Optional (see below); as Pre-T if all its daughters are T or all NT daughters are Always Optional; and as Mixed otherwise. This information is used to establish whether an NT can directly license a terminal. More exactly, an NT is deemed able to license a T if it has at least one immediate T daughter that is not Always Optional.

As for the ontological arcs, they are annotated as being:

- **Is-a** vs. **Expresses**. Is-a links denote the typical is-a or instance-of relation in knowledge representation. For example, in the E-Mail Task grammar listed in §C.2 one can extract that <_DATE_OF_WEEK> Is-a <_DATE_POINT_ARGUMENT> Is-a <_DATE_POINT>

[2]Most specific RuleName name in JSGF, e.g. _VERB_DESIRE in <voiceEMail._VERB_DESIRE>.
[3]Public enabled rules in JSGF.
[4]With keyword public in JSGF.
[5]Except when the NT in question is unreachable.

```
<_DATE>      =     <dateRange>
                 | [ <_DATE_POINT_PRE>] <datePoint>;
<dateRange> =      <_DATE_START_PRE> <dateStart>
                 | <dateStart> <_DATE_START_POST>
                 | <_DATE_END_PRE> <dateEnd>
                 | <dateEnd> <_DATE_END_POST>
                 | <_DATE_RANGE_PRE> <dateStart> <_DATE_RANGE_IN> <dateEnd>;
<dateStart> =      <datePoint>;
<dateEnd>    =     <datePoint>;
```

Figure 6.2: Grammar fragment to illustrate computation of dominating Is-a mother on the face of multiple Is-a mothers.

Is-a <_DATE>. All non-Is-a arcs are considered Expresses, e.g., <dateAfter> is a way to express <dateRange>, or <_VERB_DESIRE> a way to (help) express <listMail>.

This distinction is estimated as follows: The relation between NT_i under NT_j is of Is-a type if and only if there exists an expansion[6] of NT_j that contains only NT_i.[7] Note that according to this definition it is possible for an NT to have more than one mother Is-a. If that occurs, Vertical Generalization (see §6.3.4.1) will stop at such juncture. However, if a single one of the multiple Is-a mothers dominates the rest of the Is-a mothers, then that dominating Is-a mother is taken as the next step in the Vertical Generalization. For example, given the grammar fragment in Figure 6.2, we note that <datePoint> has three Is-a mothers (<_DATE>, <dateStart> and <dateEnd>) but since one of them (<_DATE>) dominates the others, it is the one designated as Is-a mother. If none would dominate the others, the generalization would stop at <datePoint>.

- **Always Required** vs. **Always Optional** vs. **Mixed**. In order to conduct Horizontal Generalizations (see §6.3.4.2), it is important to analyze the nature of the presence of the immediate NT daughers under an NT. If all ocurrences of NT_i under NT_j are in a required context,[8] the ontological arc from NT_i to NT_j is labeled as Always Required. If, on the other hand, all occurrences of NT_i under NT_j are in an optional context,[9] the ontological arc from NT_i to NT_j is labeled as Always Optional. And if none of the above cases holds, the arc is labeled as Mixed. For example, in the E-Mail

[6] Recall the definition of expansion set in §5.5.5.1.

[7] I.e., formally, NT_i Is-a NT_j iff $< NT_i > \in \mathrm{Exp}(NT_j)$.

[8] That is, in JSGF, if NT_i never occurs inside a RuleCount with count OPTIONAL (i.e., zero or once) or ZERO_OR_MORE.

[9] That is, in JSGF, if NT_i always occurs inside a RuleCount with count OPTIONAL (i.e., zero or once) or ZERO_OR_MORE.

Task grammar, under `<moveMail>`, `<destinationFolder>` is Always Required and `<_VERB_DESIRE>` is Always Optional.

- **Always Repeatable** vs. **Never Repeatable** vs. **Mixed**. Also for the purpose of performing Horizontal Generalizations, it is useful to record what NTs can be repeatable when immediately under another NT. Specifically, if NT_i under NT_j always appears in a repeatable context,[10] the ontological arc from NT_i to NT_j is labeled as Always Repeatable. If, instead, NT_i under NT_j always appears in a non-repeatable context, the ontological arc from NT_i to NT_j is labeled as Never Repeatable. And if none of the above cases apply, the arc is labeled as Mixed. For example, in the E-Mail Task grammar, under `<moveMail>`, `<_MAIL_ARGUMENT>` is Always Repeatable and `<_MOVE>` is Never Repeatable.

Finally, in addition to establishing the nature of the nodes (Principal vs. Auxiliary, Topologically Top-level vs. Logically Top-level vs. Non-Top-level, and Pre-NT vs. Pre-T vs. Mixed) and the arcs (Is-a vs. Expresses, Always Required vs. Always Optional vs. Mixed, and Always Repeatable vs. Never Repeatable vs. Mixed), a topological sort on the ontological nodes is computed to derive a general-to-specific partial order of the NTs. (This requires that the grammar be acyclic, which, although theoretically limiting, has not proven to be a problem, since well-designed semantic grammars hardly require cyclicity.)
As an example, Figure 6.3 shows a fragment of the ontology derived from the E-Mail Task, Figure 6.4 shows an example parse, and Figure 6.5 highlights the ontological nodes that appear in Figure 6.4.

[10] That is, in JSGF, if NT_i always occurs inside a `RuleCount` with count `ONCE_OR_MORE` or `ZERO_OR_MORE`.

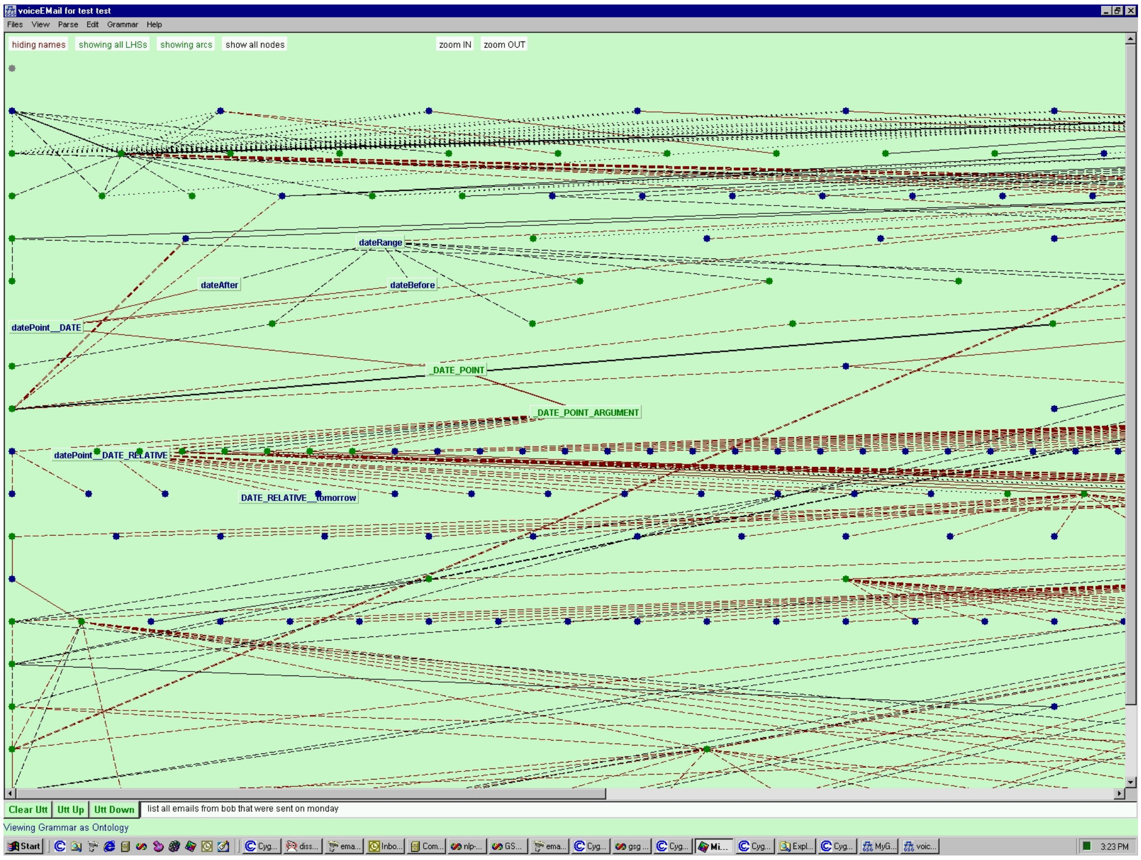


Figure 6.3: Partial vision of the Ontology of the E-Mail Task grammar listed in §B.2. Nodes in gray indicate unreachable NTs, nodes in blue principal NTs, and nodes in green auxiliary NTs. Arcs in red indicate Is-a relations, and black ones indicate Expresses relations. Arcs in solid lines represent Always Required, pointed lines Always Optional, and dashed lines Mixed. Double-thickness indicates Always Repeatable. For example, <_DATE_RELATIVE__tomorrow> is a principal NT and has <datePoint__DATE_RELATIVE> as its Is-a mother.

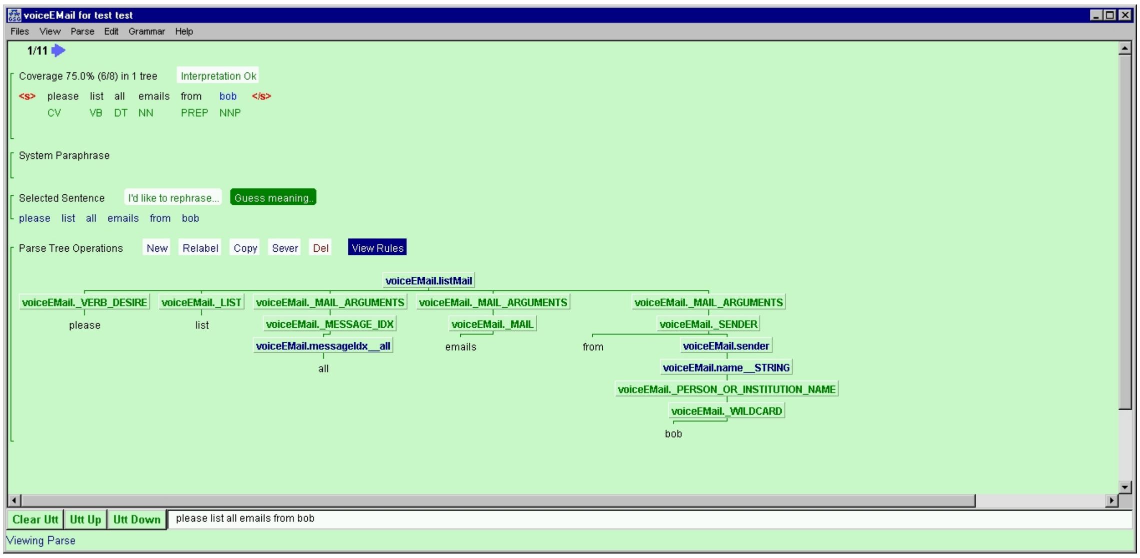


Figure 6.4: Parse of *please list all emails from bob* according to the E-Mail Task grammar in §B.2.

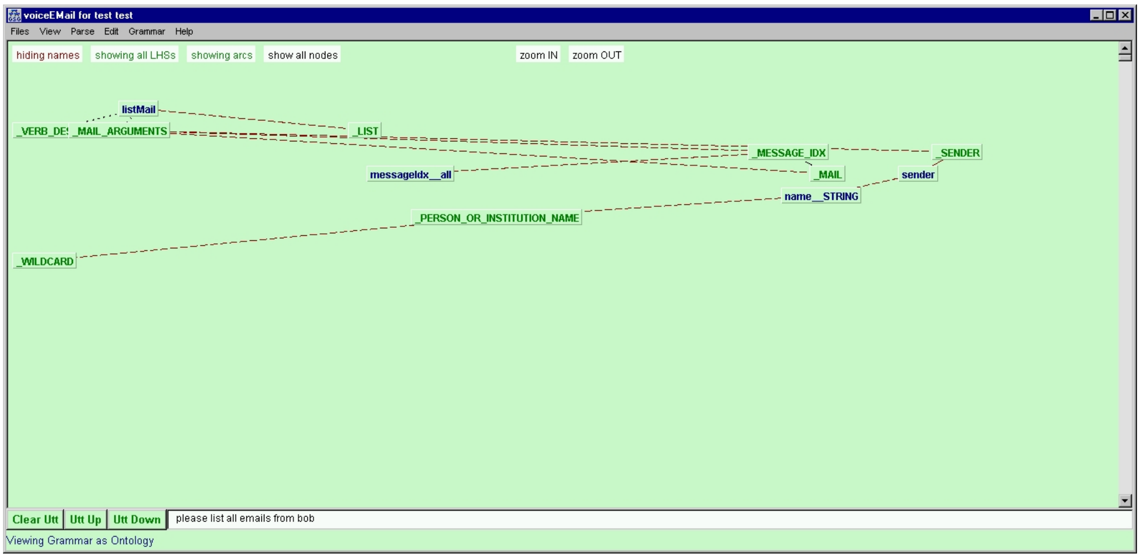


Figure 6.5: Subset of the Ontology in Figure 6.3 comprised by the NTs that are present in the parse in Figure 6.4.

6.2.3 Parsebank

After the Ontology, the next knowledge source that is derived from the Grammar is the Parsebank, a collection of parse trees. The Parsebank in GSG comprises in fact three sets of parse trees: the Kernel Grammar Set, the Training Set, and the Session Set. The Kernel Grammar Set is constructed automatically from the grammar using the generation capabilities of SOUP described in §5.3.1; the Tranining Set is the optional Kernel Parsebank depicted in Figure 6.1, and the Session Set keeps the parse trees collected from the current session that are deemed correct, i.e., the parse trees of grammatical sentences and the parse trees of previously extragrammatical sentences that have been constructed via the Interactive Parsing Algorithm.

Specifically, the Parsebank is seeded with *fixed-point* parse trees only, i.e., trees whose yield, when parsed, result in themselves as the top interpretation. This fixed-point property is needed because, to efficiently determine whether a new rule changes the existing parses, one has to quickly (i.e., in top-interpretation mode) reparse the yields of the parsebank and search for any differences in the resulting parses.

The Kernel Parsebank can optionally be provided to, on the one hand, train the grammar probabilities, and, on the other, ensure that, as the grammar grows, certain parses still obtain.

A typical way of constructing a Kernel Parsebank is by saving a Session Set parsebank at the end of the session, as it will contain the parses that have been produced during the session (and not rejected by the end-user).

The Parsebank can thus be seen as a repository of correct parse trees and a provider, for the Prediction Models, of a tree-based interface to the Grammar.

6.2.4 Prediction Models

The goal of GSG is to go beyond the robust parsing features of SOUP (such as skipping and multiple-tree interpretations) and actually hypothesize the overall meaning of extragramatical utterances. The Prediction Models are a key component for this purpose because, given a sequence of Ts and parse subtrees, they provide a list of NTs ranked by the likelihood of being the anchor mother of the input sequence. For example, in the E-Mail Task, given the sequence `<_MAIL_ARGUMENT>` `<MAIL_ARGUMENT>` `<MAIL_ARGUMENT>` *list* (obtained, for example, from having parsed the extragrammatical sentence *from bob all messages list* in all-top mode) the Prediction Models come up with the values in Table 6.1, i.e., they predict that the overall meaning of the sentence is, in ranked order, `<listMail>`, `<sendMail>` or `<deleteMail>`.

NT Name	Score
`<voiceEMail.listMail>`	0.9
`<voiceEMail.sendMail>`	$-5 \cdot 10^6$
`<voiceEMail.deleteMail>`	$-5 \cdot 10^6$

Table 6.1: Scores of the Prediction Models for the sequence `<_MAIL_ARGUMENT>` `<MAIL_ARGUMENT>` `<MAIL_ARGUMENT>` *list* (which is obtained from having parsed *from bob all messages list* in all-top mode).

In other words, the function of the Prediction Models is to hypothesize a mother for a sequence of Ts and parse subtrees. It is used most notably in the first step of the Interactive Parsing Algorithm (see Figure 6.8 in §6.3) but also during the application of Daughter Argument Selection strategies (see §6.3.2), for example to determine which unparsed segments best match a particular required daughter.
How are these predictions made? After some experimentation with a variety of models and classifiers (mostly HMMs, decision trees and n-gram models), the most useful model found for this specific task of predicting likely mother NTs for a sequence of subtrees and unparsed words is what can be termed *hypotactical* and *paratactical* n-gram models. N-gram models are probabilistic models of word sequences with equivalence classes of order n, that is, the probability of a word following a sequence of words is equated to the probability of that word following the last $n-1$ words of the sequence.[11] The hypotactical and paratactical n-grams encode and predict two kinds of linguistic relations: the hypotactical n-gram models the nestedness of language (e.g., in the E-Mail Task, the fact that *Tuesday* tends to occur under `<dayOfWeek_2>`, which in turn tends to occur under `<_DAY_OF_WEEK>`, itself under `<_DATE_POINT_ARGUMENT>`, etc.), whereas the paratactical n-grams model adjacency (e.g., the fact that the sequence $<$*New*, *Years*$>$ is often followed by *Day*, especially under NT `<dateFixed_jan01>`).
These special n-grams are described in more detailed below.

6.2.4.1 Hypotactical Model

The hypotactical n-gram model (from the Greek *hypotassein*, meaning *to arrange under*) describes the nestedness relation of parse tree nodes. There is one hypotactical model per grammar. Its vocabulary is the union of the terminals and nonterminals of the grammar, i.e., the labels of the nodes of the parse trees contained in its training Parsebank, and its events are the spines of such trees, i.e., all the paths from leaf to root in the parse trees.

[11] Formally, $P(w_i|w_1...w_{i-1}) = P(w_i|w_{i-n-1}...w_{i-1})$; see, for example, [Jelinek, 1997] or [Manning and Schütze, 1999] for an in-depth study.

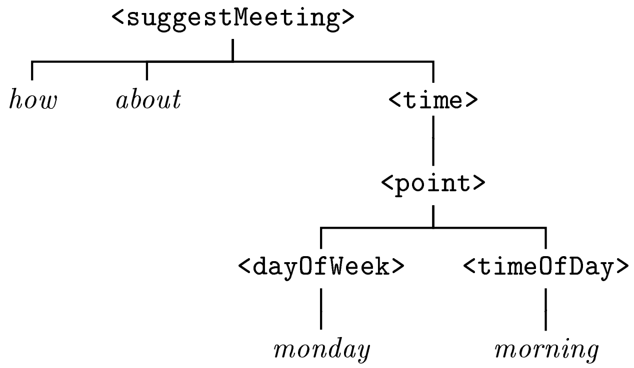


Figure 6.6: Example parse tree to illustrate events for hypotactical and paratactical models.

Again, this serves to model the nestedness or dominance relations of NTs over other NTs and Ts.
For example, from the parse tree in Figure 6.6, the following nestedness relations are extracted and given as input sequences (i.e., events) to the hypotactical n-gram model:

- < *how* , <suggestMeeting> >
- < *about* , <suggestMeeting> >
- < *monday* , <dayOfWeek> , <point> , <time> , <suggestMeeting> >
- < *morning* , <timeOfDay> , <point> , <time> , <suggestMeeting> >

6.2.4.2 Paratactical Models

The paratactical n-gram model (from the Greek *paratassein*, meaning *to arrange side by side*) describes the distribution of NTs and Ts within an NT. There is one paratactical model for each NT in the grammar. Their vocabulary is also the union of the terminals and nonterminals of the grammar, and the events are ordered daughters occurring immediately under their namesake nonterminal.
For example, from the same parse tree in Figure 6.6, the following events are extracted:

- For <suggestMeeting>'s paratactical model:
  - < *how* , *about* , <time> >
- For <time>'s paratactical model:

  - $<$ `<point>` $>$

- For `<point>`'s paratactical model:
  - $<$ `<dayOfWeek>` , `<timeOfDay>` $>$
- For `<dayOfWeek?`'s paratactical model:
  - $<$ *monday* $>$
- For `<timeOfDay>`'s paratactical model:
  - $<$ *morning* $>$

6.2.4.3 Combination of Hypotactical and Paratactical Models

After training the hypotactical and paratactical models from the events extracted from the parse trees contained in the Parsebank, the models are ready to be used.
At runtime, given a piece of evidence in the form of a sequence of unparsed terminals and parse subtrees, the models are queried to hypothesize an overall mother. More exactly, the procedure is as follows: Given the evidence $\mathbf{e} =< e_1 ... e_m >$ (sequence of Ts and subtrees), compute the score of the hypotactical n-gram model for each NT, defined as

$$P_{\text{Hypo}}(NT_i|\mathbf{e}) \quad \stackrel{\text{def}}{=} \quad \sum_{j=1}^{m} P_{\text{Hypo}}(NT_i|e_j)$$

This results in a vector of reals of dimension $|NT|$ (the number of grammar nonterminals), hereafter referred to as $\mathbf{v}_{\text{Hypo}}$. Also compute the score of $\mathbf{e}$ for each paratactical n-gram model, defined as

$$P_{\text{Para}}(NT_i|\mathbf{e}) \quad \stackrel{\text{def}}{=} \quad \sum_{j=1}^{m} P_{\text{Para},NT_i}(r(e_j)|r(e_{j-(n-1)})...r(e_{j-1}))$$

where

$$r(e_j) \quad \stackrel{\text{def}}{=} \quad \left\{ \begin{array}{lcl} e_j & : & e_j \text{ is a terminal} \\ \text{root}(e_j) & : & e_j \text{ is a tree} \end{array} \right\}$$

that is, the sum of the probability of the overlapping windows of order n over the terminals and the roots of the subtrees present in the evidence. This again results in a vector of reals of dimension $|NT|$, hereafter referred to as $\mathbf{v}_{\text{Para}}$. Finally, interpolate $\mathbf{v}_{\text{Hypo}}$ and $\mathbf{v}_{\text{Para}}$ via a linear combination

$$\mathbf{v}(i) \quad \stackrel{\text{def}}{=} \quad \lambda \cdot \mathbf{v}_{\text{Hypo}}[i] + (1-\lambda) \cdot \mathbf{v}_{\text{Para}}[i] \quad \forall\, i : 1 \leq i \leq |NT|$$

where $0 \leq \lambda \leq 1$, and sort $\mathbf{v}$ descendingly, i.e., generate a ranked list of mother nonterminals.

To automatically test the goodness of the hypotactical and paratactical models, and to empirically find a suitable λ as interpolation weight between the hypotactical and paratactical scores, the following experiment was conducted.

First a *parse tree distance metric* was devised to capture the degree of similarity between two parse trees. The main factors in the distance metric are a penalty for NTs present in one tree but not in the other, penalty for NT and T mismatches under the same NT, a depth decay factor (agreement at the root more important that at the leaves), and, finally, a normalization factor on the number of nodes. Then, a test parsebank was obtained by parsing test utterances with a fully-developed grammar. Subsequently, the grammar was *decimated* (RHS alternatives were removed at random), and the hypothesized trees predicted by the models from the evidence obtained by parsing with the decimated grammar was compared (via the distance metric) against the correct trees in the test parsebank.

Tables 6.2–6.5 list the scores for an experiment with a Scheduling grammar, consisting of about 600 NTs and 3000 RHS alternatives. The test parsebank contained 457 parse trees. There are four tables, each with a different degree of decimation. The first number indicates the actual portion of rules removed from the grammar, e.g., 45% (50%:1) means that from a target decimation of 50% but with the constraint of leaving at least one RHS alternative per NT, the real decimation was 45%.

In each table, the first column lists the value of λ_{Hypo}, i.e., the weight of the hypotactical score (with the weight of the paratactical score, λ_{Para}, being $1 - \lambda_{\text{Hypo}}$). The other three main columns list the average tree distance over the test parsebank of 457 examples and the number of cases in which the distance is exactly zero (i.e., a perfect match). **Baseline** refers to a baseline hypothesis where no prediction models are employed. In that case, a tree with a dummy root and the evidence obtained from all-top parsing directly attached to it is used as the hypothesized tree. **GSG-0** refers to using the Prediction Models statically, i.e., without dynamic updates. **GSG-1** refers to using the Prediction Models dynamically, i.e., incrementally adding the new rules hypothesized by the system into the grammar and into the models themselves. Still, GSG-1 is conservative in the sense that it does not add rules that introduce ambiguity or change previous parses.

Note that in all cases, GSG-1 is better (average distance smaller) than GSG-0, which in turn is better than the baseline. As for the effect of λ_{Hypo}, it is interesting to observe that, in the case of low decimation (Table 6.2) as well as high decimation (Table 6.5), the best results are obtained with $\lambda_{\text{Hypo}} = 0$, i.e., using the paratactical models only. However, in medium decimation (Tables 6.3 and 6.4) the optimum is achieved with $\lambda_{\text{Hypo}} \simeq 0.5$, which is the value taken for the rest of the experiments reported in this dissertation.

	Baseline		GSG-0		GSG-1	
λ_{Hypo}	Avg. d	|d=0|	Avg. d	|d=0|	Avg. d	|d=0|
0.0	1.4364	148	0.8579	149	0.6933	153
0.2	1.4364	148	0.8692	149	0.8192	160
0.4	1.4364	148	0.8817	149	0.8612	156
0.6	1.4364	148	0.9190	149	0.8940	156
0.8	1.4364	148	0.9351	150	0.9149	161
1.0	1.4364	148	1.3926	148	1.3172	166

Table 6.2: Experiment to determine optimal weight (λ_{Hypo}) for linear interpolation of hypotactical and paratactial models. Results for grammar decimation 17% (25%:1), i.e., of a target removal of 25% of RHS alternatives but leaving at least RHS alternative per LHS, an actual decimation of 17% was achieved. Hypotactical model weight λ_{Hypo} ranges from 0.0 to 1.0, paratactical model weight is defined as 1 - λ_{Hypo}. *Avg. d* refers to the average parse distance between the canonical parse trees (obtained with no grammar decimation) and the decimated grammar. |d=0| refers to the number of trees that obtain a distance of zero, i.e., are identical to the canonical ones. Baseline indicates the baseline model, i.e., without Prediction Models. GSG-0 indicates the usage of *static* Prediction Models. GSG-1 indicates the usage of *dynamic* Prediction Models.

	Baseline		GSG-0		GSG-1	
λ_{Hypo}	Avg. d	|d=0|	Avg. d	|d=0|	Avg. d	|d=0|
0.0	1.9934	68	1.2164	71	1.1857	70
0.2	1.9934	68	1.2165	71	1.1447	81
0.4	1.9934	68	1.2040	71	1.1788	79
0.6	1.9934	68	1.1994	71	1.1640	74
0.8	1.9934	68	1.2502	71	1.1917	80
1.0	1.9934	68	2.0577	68	2.0400	83

Table 6.3: Results with decimation 45% (50%:1). (See 6.2 for details.)

6.2.4.4 Model Adjustments

As an extra step, the evidence can be "jump-started," that is, a sequence of unparsed Ts is substituted by its most likely mother according to the hypotactical and paratactical models, and then the new evidence is considered.

In addition, a "boost" is given to those NTs that are top-level, or can reach a top-level NT through Is-a links, to accommodate *extension* of coverage of an NT (rather than postulating another NT). For example, *list messages of today* parses *list messages* as `<listMail>` which

λ_{Hypo}	Baseline Avg. d	Baseline |d=0|	GSG-0 Avg. d	GSG-0 |d=0|	GSG-1 Avg. d	GSG-1 |d=0|
0.0	2.2756	31	1.7939	32	1.7629	38
0.2	2.2756	31	1.8023	32	1.7415	37
0.4	2.2756	31	1.7871	32	1.7260	39
0.6	2.2756	31	1.8450	32	1.7092	40
0.8	2.2756	31	1.8102	32	1.7597	38
1.0	2.2756	31	3.1511	31	3.1412	42

Table 6.4: Decimation 66% (75%:1). (See 6.2 for details.)

λ_{Hypo}	Baseline Avg. d	Baseline |d=0|	GSG-0 Avg. d	GSG-0 |d=0|	GSG-1 Avg. d	GSG-1 |d=0|
0.0	2.3601	9	2.0865	11	1.9319	13
0.2	2.3601	9	2.0754	11	1.9356	11
0.4	2.3601	9	2.1629	11	1.9425	12
0.6	2.3601	9	2.1583	11	1.9681	12
0.8	2.3601	9	2.1682	11	2.0117	12
1.0	2.3601	9	4.0438	9	4.0444	10

Table 6.5: Decimation 79% (100%:1). (See 6.2 for details.)

is in fact the correct anchor mother. Thus, when gathering evidence, `<listMail>` is found and is given a predefined score of 0.9 so that it surfaces as the top option.[12]

Note that the paratactical models are a generalization of Seneff's TINA parsing [Seneff, 1992], where bigrams of NTs are used to find most likely parses.[13] Also, the hypotactical models appear to be a better modeling of natural language compared to that of Minker's [Minker, 1997], where each different parse tree spine (chain from leaf to root NTs) is considered a different token.

All the experiments reported in this thesis used a language model order of $n = 4$, for the hypotactical models, and $n = 2$, for the paratactical models, with a back-off interpolation mechanism for unseen events.

[12] Should there be more than one top-level NT present in the evidence, then they would all get a score of 0.9.

[13] However, in the actual runs of GSG described in this thesis, the n-grams for the paratactical models are of order 2 and thus equivalent to TINA's.

6.2.5 Part-of-Speech Tagging and Shallow Syntactic Parsing

GSG's philosophy is to require as little knowledge as possible for the development and launching of new applications. However, this does not prevent the exploitation of domain-independent linguistic knowledge, such as morphosyntaxis. GSG incorporates, for English, Brill's part-of-speech tagger [Brill, 1994], with modifications from Klaus Zechner and the author, as well as a shallow syntactic grammar written by Klaus Zechner. The set of part-of-speech (POS) tags and the shallow syntactic grammar are listed in §B.1 and §B.3 respectively, and an example of a POS-tagged sentence and its syntactic parse can be seen in Figure 6.7.

The purpose of obtaining the POS tags for an incoming utterance is twofold: on the one hand, it is a required step for the syntactic analysis (since the vocabulary of the syntactic grammar is comprised of POS tags); on the other, it is useful to know the syntactical category of the extragrammatical words to determine whether they should be learned at all or simply skipped. For example, in the current implementation of GSG, unparsed conjunctions, determiners and pronouns that constitute a single-word segment (see below) are ignored in the Daughter Argument Selection phase of the Interacive Parse Algorithm (see §6.3.2).

As for the shallow syntactic parse, its main utility is in helping the segmention of contiguous sequences of unparsed words: As an improvement over [Lehman, 1989]'s Single Segment Assumption, which considers a sequence of contiguous unparsed tokens a single segment, GSG takes a different approach and considers each word to form a different segment unless it belongs to the same noun-phrase bracket.

For example, the sentence *obtain my sister's messages*, where the only known words to the semantic grammar are *'s* and *messages*, the Single Segment Assumption would consider *obtain my sister* as one segment that performs a single function in the utterance, whereas GSG is able to identify *obtain* and *my sister* as separate segments to learn. In particular, GSG uses the POS tagger to obtain the sequence `VB PRP$ NN NNP NNS` which is then parsed with the syntactic grammar to obtain the parse in Figure 6.7.

Since *obtain* is on a syntactic tree of its own, it is considered a single segment. But given that *my* and *sister* belong to the same syntactic root `<NP>` they are grouped.[14]

In a few cases this approach over-segments (see, for example, learning episodes LE e.5.1 (*can you check*), LE e.7.2 (*did i receive*) and LE e.10.3–4 (*get rid of*) in §E.1) but, in general, it

[14]This also illustrates that if a noun phrase bracket covers a pronoun, the the pronoun is not ignored (as it would if it stood in a segment by itself).

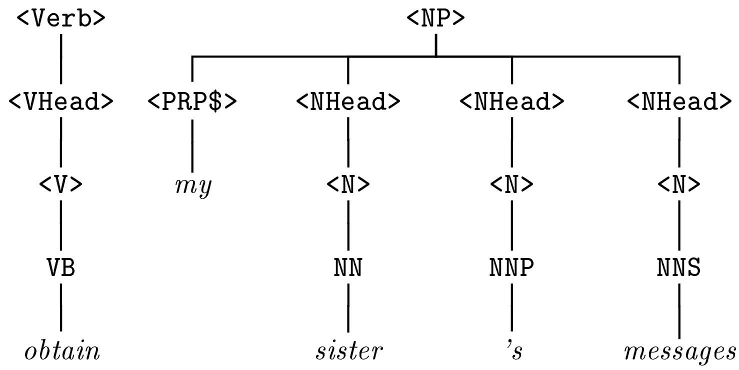


Figure 6.7: Syntactic analysis of *obtain my sister's messages*. First, Brill's part-of-speech tagger [Brill, 1994] is run to obtain the sequence of part-of-speech tags `VB PRP$ NN NNP NNS` (see §B.1 for an explanation); then, the tags are parsed by the sytantactic grammar (listed in §B.3), which yields the above parse tree.

performs better than assuming all contiguous unparsed words constitute a single functional segment.

6.3 Learning Strategies

The aim of Gsg is to postulate meanings for extragrammatical sentences and subsequently update the grammar as a way to remember and generalize such learning episodes. As first outlined in §4.2.2, the fundamental algorithms of Gsg are Interactive Parsing, Means Operator and Isa Operator (for the construction of meaning representations for extragrammatical utterances), as well as Subsumption Detection, Ambiguity Detection, Vertical Generalization, Horizontal Generalization and Rule Merging (for rule management). In this section a more detailed explanation is presented. Figures 6.8 and 6.9 refine the Interactive Parsing Algorithm first sketched in Figure 4.2, and Table 6.6 summarizes the strategies employed by Gsg, which are described below.

Input: Natural language utterance as sequence of words.

Step 1. **Parse in default mode**: Parse input utterance in SOUP's word-level default mode (see §5.5.1, i.e., with only logically top-level NTs able to stand at the root of a parse tree). If no parse is found, or the end-user is not satisfied with the resulting interpretation (as evidenced by rejecting a paraphrase of such interpretation), go to Step 2, else exit algorithm.

Step 2. **Collect evidence**: Parse sentence in SOUP's all-top mode (see (§)5.5.3, i.e., with all grammar NTs able to stand at the root of a parse tree). This will result in a sequence of subtrees and unparsed words, called evidence. For all subtrees, extend trunk following ontological Is-a links, i.e., perform Vertical Generalization on subtree roots.

Step 3. **Establish anchor mother (automatic part)**: Apply Prediction Models to hypothesize NT roots from the evidence collected in Step 2 as described in §6.2.4. Possibly re-rank list with information about the state of the end-application as given by the Back-end Application Manager.

Step 4. **Establish anchor mother (interactive part)**: Pose confirmation or multiple-choice question to the user to establish anchor mother. The user may

- (*i*). Select one of the options proposed by the system, in which case that becomes the anchor mother.
- (*ii*). Volunteer linguistic information that triggers ISA or MEANS Operator, in which case a sub-learning episode is launched through the invocation of the ISA or MEANS Operator Algorithm, at the completion of which, unless the sub-learning episode has been canceled, this algorithm goes back to Step 1 (as new rules may have been introduced).
- (*iii*). Cancel learning episode (metagrammar NT `<cancel>`), in which case this algorithm exits.

Figure 6.8: GSG's Interactive Parsing Algorithm (Part I).

Step 5. **Construct Daughters (automatic part)**: While unparsed segments remain, apply Daughter Argument Selection strategies (Verbal Head Search, Required/Is-a/... Daughter Search, and Parser Predictions) to hypothesize a daughter argument under the anchor mother.

Step 6. **Construct Daughters (interactive part)**: Pose confirmation or multiple-choice questions to the user to establish daughter arguments. The user may

(*i*). Select one of the options proposed by the system, in which case that becomes the anchor mother.

(*ii*). Volunteer linguistic information that triggers ISA or MEANS Operator, in which case a sub-learning episode is launched through the invocation of the ISA or MEANS Operator Algorithm, at the completion of which, unless the sub-learning episode has been canceled, this algorithm goes back to Step 1 (as new rules may have been introduced).

(*iii*). Cancel learning episode (metagrammar NT `<cancel>`), in which case this algorithm exits.

Step 7. **Construct parse tree**: Build a complete parse tree for the extragrammatical sentence from the anchor mother obtained in Step 3 and the daughter subtrees obtained in Step 4. (If the anchor mother is not logically top-level, follow ontological Is-a links to a top-level node. This sequence becomes the trunk of the parse tree.)

Step 8. **Update grammar**: Extract all rules form the parse tree (see §4.2.2.2). For each new grammar rule: discard if already subsumed by the grammar (see §6.3.5.1); discard if it introduces ambiguity (see §6.3.5.2); discard if it disrupts existing parse trees (see §6.3.5.3); generalize via Vertical and Horizontal Generalization Algorithms (see §6.3.4); add to the grammar via Rule Merging Algorithm (see §6.3.5.4).

Step 9. **Update hypotactical and paratactical models**: For each new rule added to the grammar in Step 8: generate a few or all distinct parse trees rooted at its left-hand side nonterminal. Add all parse trees to the current Parsebank. Update hypotactical and paratactical models using the new data. Exit algorithm.

Figure 6.9: GSG's Interactive Parsing Algorithm (Part II).

Strategy	Explanation	Knowledge source
All-top Parsing	All NTs treated as starting symbols of the grammar to collect evidence for overall meaning of sentence	Grammar
Anchor Mother Predictions	Bottom-up hypothesis of most likely NT to cover evidence	Prediction Models
Required/Is-a/... Daughter Search	Top-down search of most likely daughter to fulfill required/Is-a/... rôle	Ontology
Verbal Head Search	Top-down search of most likely daughter to be verbal head	POS Tagger, Ontology
Parser Predictions	Bottom-up/top-down search of most likely NT to follow parsed segment	Grammar
Vertical Generalization	Bottom-up generalization along ontological Is-a links	Ontology
Horizontal Generalization	Generalization by making RHS constituents optional and/or repeatable	Ontology

Table 6.6: Summary of GSG's prediction, learning and generalization strategies.

6.3.1 Establishing the Anchor Mother

As explained in Steps 2 and 3 of Figure 6.8, the techniques employed to establish the anchor mother of the hypothesized parse tree for an ungrammatical sentence are All-top Parsing, to collect the evidence, and the Prediction Models, to hypothesize the anchor mother.

6.3.1.1 Collectig Evidence

To collect evidence for the Prediction Models, the extragrammatical utterance is parsed in SOUP's all-top mode (see §5.5.3). Since all NTs are considered top-level it is much more likely to find parses for segments of the utterance.

SOUP's heuristics (see §5.4) imply that in all-top parsing mode the most specific trees will be preferred. For example, the parse tree root of *tuesday* would be `<dayOfWeek>` instead of `<time>`. However, this specialization is overcome by applying Vertical Generalization to the trees in the evidence.

6.3.1.2 Predicting Anchor Mother

After collecting the evidence, the next step is to establish the overall meaning of an utterance, i.e., find its "anchor mother" or root node of the desired analysis. To that effect the Prediction Models are employed as described in §6.2.4.3, i.e., given the evidence as a vector of unparsed words and parse trees, the scores of the Hypotactical and Paratactical Models are combined to provide a ranked list of the most likely anchor mothers. The NTs whose likelihood falls above a certain threshold are presented to the end-user.
To cushion the end-user from the actual names of the NTs, an automatically generated example is presented, e.g. `<trackTitle>` becomes "`track title, e.g.  "my all"`." Also, in order not to overwhelm the end-user (especially if the options are conveyed by speech) the suggested NTs are grouped in batches of three.

6.3.2 Daughter Argument Selection

Once the anchor mother has been established, it is time to build the rest of the parse tree as the meaning representation for the extragrammatical sentence. For that purpose, three types of strategies are used: Required/Is-a/... Daughter Search, Verbal Head Search, and Parser Predictions.

6.3.2.1 Required/Is-a/... Daughter Search

In this strategy, the Ontology is used to find potential daughters for the anchor mother: the semantic rôles (daughter NTs) licensed by the anchor mother according to the Ontology are considered candidates to cover unparsed segments of the evidence. The subcategorization candidates of the anchor mother are ranked by the following, manually-set scores:

- **Required daughter**: Score 1.0. Search for most likely unparsed segment to give rise to a Required ontological daughter of the anchor mother.

- **Is-a daughter**: Score 0.9. Search for most likely unparsed segment to give rise to a Reversed Is-a ontological daughter of the anchor mother (i.e., a daughter that Is-a anchor mother).

- **Non-optional daughter**: Score 0.5. Search for most likely unparsed segment to give rise to any Non-optional ontological daughter of the anchor mother. This in fact means a Mixed daughter, since Required daughters have already been considered.

- **Any other daughter**: Score 0.1. Search for most likely unparsed segment to give rise to any remaining ontological daughter of the anchor mother.

For example, given an Ontology in which NT `<m>` has NT `<d1>` and `<d2>` as Required daughters and NT `<d3>` as a Reversed Is-a daughter, and an evidence $u_1 ... u_n$ $u_{n+1} ... u_m$ `<d2>` $u_p ... u_q$ (where u_i denote unparsed words)[15] and anchor mother `<m>`, the Required/Is-a/... Daughter Search would proceed as follows. First, possible daughters of `<m>` would be gathered from the Ontology and ranked according to their nature (e.g., preference for Required ones). That would give `<d1>` and `<d2>`, followed by `<d3>`. But `<d2>` would be discarded since its rôle is already fulfilled in the evidence.[16] Thus, `<d1>` would be a candidate to cover unparsed segments $u_1 ... u_n$, $u_{n+1} ... u_m$, or $u_p ... u_q$. The Prediction Models would be used to establish which unparsed segment should be considered first, and a question would be posed to the user (e.g., a formulation to the effect of *By* $u_p ... u_q$ *do you mean* `<d1>`*?*). (See §6.3.6.3 for details on the formulation of questions).

6.3.2.2 Verbal Head Search

Since most utterances contain a verb,[17] it is useful to locate the principal or head verb in an utterance. Similar to the Required/Is-a/... Daughter Search, if the Ontology licenses a daughter that is mostly verbal in nature, a search for an unparsed verb is launched. The "verbness" degree of each NT is established by computing a "verbness ratio" defined as the proportion of words tagged as verbs in a set of automatically generated sentences[18] from the NT in question. For example, the NT `<_LIST>` of the E-Mail Task grammar (§C.1) obtains a verbness ratio of about[19] 0.8, for most of its generations are verbs (e.g., *list*, *get*, *search*, *search for*).[20]

In this strategy, the POS of the evidence is also employed to find the unparsed segment[21] that is most likely to be a verb.

[15] Note that unparsed segments may be contiguous: as described in §6.2.5, unparsed words are only grouped into segments if they belong to the same noun phrase.

[16] Unless it were Repeatable.

[17] In fact, all complete sentences in the traditional, grammatical sense of the term *sentence* must contain a verb or predicate.

[18] Now in the sense of a sequence of words.

[19] Depending on the stochastic generation.

[20] The fact that POS membership may be ambiguous for many words is ignored; only the top POS tag, as given by Brill's tagger, is used.

[21] Most probably containing a single word, since no verb would be part of a shallow noun phrase.

6.3.2.3 Parser Predictions

Another strategy in the selection of daughter arguments is to use the grammar as a predictor of what can come after a parsed sequence. That is, given the context in which an unparsed sequence occurs, the grammar is traversed to find likely continuations of the context. It is both a bottom-up and top-down strategy: if the nonterminal predicted does not license terminals (see §6.2.2), its daughters are followed top-down, collecting the NTs that allow for immediate Ts.
This is exemplified by the case of *christmas* in the detailed example presented in §6.4.2, or by the learning episode LE e.4.2 in §E.1.4.2, where the class (NT) of *recency* is deduced by appearing immediately after *by*.

6.3.2.4 Strategy Ordering

The actual order in which the strategies are applied is fixed for all the experiments described in this dissertation, and is listed below.

1. Required Daughter Search.
2. Verbal Head Search.
3. Parser Predictions (only applied if anchor mother does not license Ts).
4. Is-a Daughter Search.
5. Non-optional Daughter Search (only applied if anchor mother does not license Ts).
6. Any Daughter Search (only applied if anchor mother does not license Ts).
7. Any NT Search (only applied if anchor mother does not license Ts).

It is of course conceivable that a dynamic ordering of the strategies may perform better, e.g., by reducing the average number of choices. One could train, say, a decision tree to find the most efficient ordering of the strategies given a certain context, but this second-order search lies beyond the scope of this work.

6.3.3 Licensing Constraints

Similar in spirit to the strategies employed by [Kiyono and Tsujii, 1993] (see §3.5), the above strategies are subject to the following constraint: Ts and principal NTs must be licensed by the Ontology. That is, an NT can have a T as immediate daughter only if it does so in the Ontology (i.e., ultimately, in the Kernel Grammar). Also, principal NTs must be licensed by the Ontology, i.e., an NT can have another principal NT as its immediate daughter only if such dominance relation occurs in the Ontology. (An auxiliary NT, on the other hand, has no such constraints.) The reasoning behind these constraints is the need to preserve the topology of the principal NTs in the Ontology, as they define the backbone of the feature structure definition that the Back-end Application Manager interprets and executes. These constraints guarantee that all resulting feature structures are structurally correct, for they will not contain unknown combinations of principal NTs (i.e., attributes), although they may contain unexpected (atomic) values.

6.3.4 Rule Generalization

Upon completing a successful learning episode, one would like the system to not only learn the meaning of that particular sentence but also be able to generalize to account for similar constructions. This is accomplished in GSG via the Vertical and Horizontal Generalization algorithms.[22]

6.3.4.1 Vertical Generalization

The Vertical Generalization algorithm takes a sequence of Ts and NTs (e.g., a candidate subRHS extracted from a validated parse tree) and follows ontological Is-a links for all the NTs in the sequence. This serves the purpose of making the sequence more general, for example, given the E-Mail Task grammar in §C.2, the sequence *in the morning*, is parsed in all-top mode as the tree in Figure 6.10, which gives `<_TIME_OF_DAY>` as the root NT. But, upon applying Vertical Generalization, this NT becomes `<_MAIL_ARGUMENT>` (through the `<_TIME_OF_DAY>` —Is-a→ `<_DATE_POINT_ARGUMENT>` —Is-a→ `<datePoint>` —Is-a→ `<_DATE>` —Is-a→ `<_MAIL_ARGUMENT>` chain).[23]

6.3.4.2 Horizontal Generalization

The Horizontal Generalization algorithm relaxes a candidate rule by attempting to make some of the constituents in its RHS optional and/or repeatable. It uses the information in the Ontology (specifically, the nature of the arcs that go from RHS constituents of the given rule to its LHS) to determine whether the Ts and NTs present in the RHS can be made optional and/or repeatable. In addition, the presence of an NT in two consecutive positions makes it repeatable, e.g., `<a> <a>` becomes `<a>+`.

[22] As well as via the Rule Merging algorithm as described in §6.3.5.4.

[23] If this generalization appears too extreme, it is only because of the high degree of orthogonality that the E-Mail Task grammar possesses. One could choose not to group all the mail arguments under the single NT `<_MAIL_ARGUMENT>`, in which case the Vertical Generalization of *in the morning* would stop at `<_DATE>` (see §6.3.7 for further discussion).

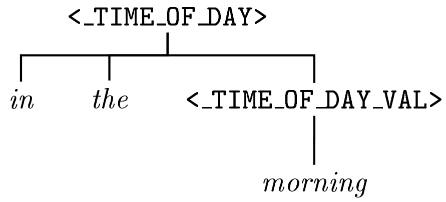


Figure 6.10: All-top parse of *in the morning* according to the E-Mail Task grammar. Upon applying the Vertical Generalization the root `<_TIME_OF_DAY>` is generalized to `<_MAIL_ARGUMENT>`.

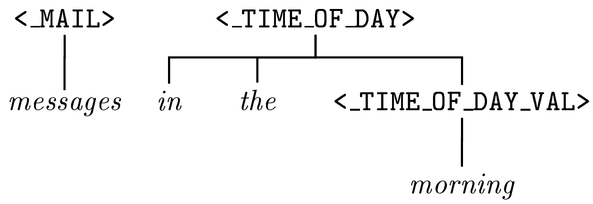


Figure 6.11: All-top parse of *messages in the morning* according to the E-Mail Task grammar. Upon applying the Horizontal Generalization algorithm after Vertical Generalization, the root sequence `<_MAIL> <_TIME_OF_DAY>` is generalized to `<_MAIL_ARGUMENT>+`.

For example, the sentence *messages in the morning*, is parsed in all-top mode as the analysis in Figure 6.11. The sequence of roots would be first vertically generalized to `<_MAIL_ARGUMENT> <_MAIL_ARGUMENT>` and then horizontally generalized to `<_MAIL_ARGUMENT>+`. Similarly, a *please* parsed as `<_VERB_DESIRE>` under, say, `<listMail>` would be horizontally generalized to `[<_VERB_DESIRE>]`, i.e., made optional, since, according to the Ontology, the NT `<_VERB_DESIRE>` is Always Optional under `<listMail>`.

6.3.5 Rule Management

The aim of GSG is to extend the Kernel Grammar, but in a judicious manner. A concern about allowing the end-user to (indirectly) modify a grammar is that the grammar may grow untamed and become filled with new rules that disrupt the original grammar. To prevent this, besides the careful construction of interpretations via the strategies described above, GSG employs three safety mechanisms: before a rule is added to the grammar, it is checked whether it is redundant, whether it introduces ambiguity to the grammar, and whether it disrupts existing, correct interpretations.

6.3.5.1 Detection of Rule Subsumption

As explained in §5.5.5.3, rule subsumption or redundancy detection is accomplished by using SOUP to parse the candidate subRHS (the set of expansions of the subRHS, to be precise). If all expansions can be parsed under the candidate's LHS (i.e., by the current RHS), then the new rule is *subsumed* by the existing RHS and can therefore be discarded since it is redundant (adding it to the grammar would not change the language that the grammar defines).

6.3.5.2 Detection of Ambiguity Introduction

If the candidate rule is not subsumed, SOUP is again employed to detect introduction of ambiguitity (see §5.5.5.2). In that case, the existence of a parse for an expansion of the candidate subRHS, under any NT in the grammar, indicates ambiguity. If a rule introduces ambiguity it is typically discarded, although, depending on the user settings (advanced vs. naïve) the ambiguity may be presented to the user and the decision delegated.

6.3.5.3 Parsebank Disruption Check

Finally, before adding a rule, a reparse of all or a random sample of the Parsebank is performed[24] to ensure that the new rule does not disrupt correct parses.

6.3.5.4 Rule Merging

Once it has been determined that a subRHS needs to be added to the grammar, the rule merging algorithm is invoked. A naïve algorithm would just add the new subRHS as an alternative to the current RHS. For example, given initial rule `<S> = [please] sort`

[24] Recall that SOUP can typically parse on the order of 100 utterances per second for a 600-NT grammar (see §5.8).

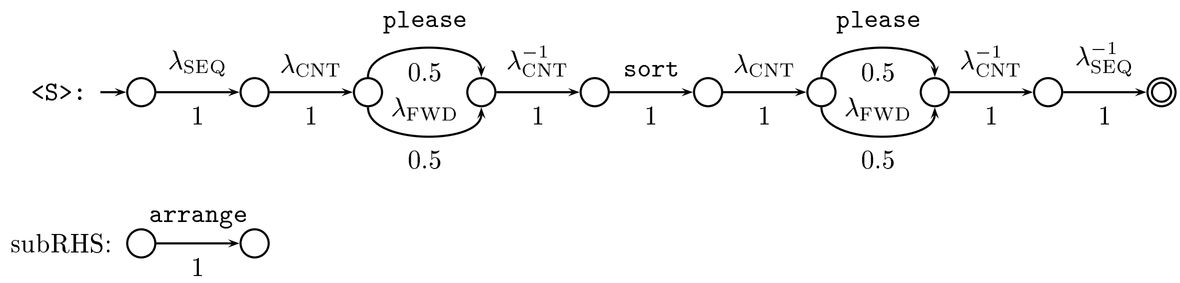


Figure 6.12: PRTN for initial rule <S> = [please] sort [please] and PRTN fragment for subRHS arrange.

[please] and subRHS arrange, a naïve merge would yield the rule <S> = ([please] sort [please]) | arrange (see Figures 6.12 and 6.13).

However, GSG implements a more advanced algorithm: Add the new subRHS as an alternative to the segment between the first non-skippable constituent and the last non-skippable constituent in the current RHS. That is, the insertion start point is found by skipping initial optional constituents and the insertion end point by skipping final optional constituents.

In the case of the example, the resulting rule is <S> = [please] (sort | arrange) [please] (see Figure 6.14), which generalizes better, as it covers {*sort*, *arrange*, *please sort*, *please arrange*, *sort please*, *arrange please*, *please sort please*, *please arrange please* }, as opposed to the naïve merge, whose resulting rule only covers {*sort*, *arrange*, *please sort*, *sort please*, *please sort please*}.

In some cases this algorithm does not yield the most compact RHS for the language it defines (e.g., cf. a b* b* vs. a b*, or see all Learning Episodes in §E whose score ends in a minus sign, such as LE e.1.1). But that is not a problem because what matters is coverage, not the succinctness of the rule. In any case, as elaborated in §8.3.7, a specialized algorithm could be applied to find the most compact representation of a rule after merging.

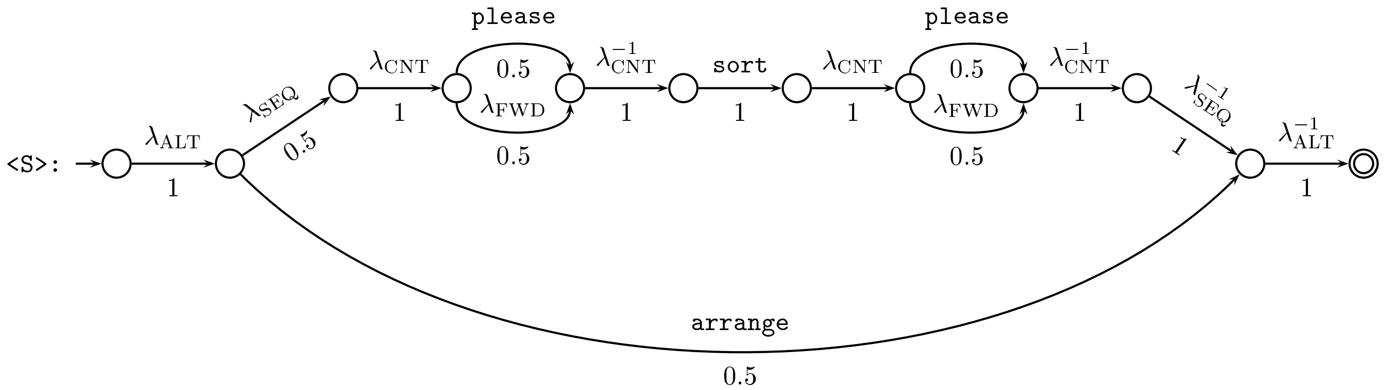


Figure 6.13: Naïve merging of rule `<S> = [please] sort [please]` and subRHS `arrange`, resulting in `<S> = [please] sort [please] | arrange`.

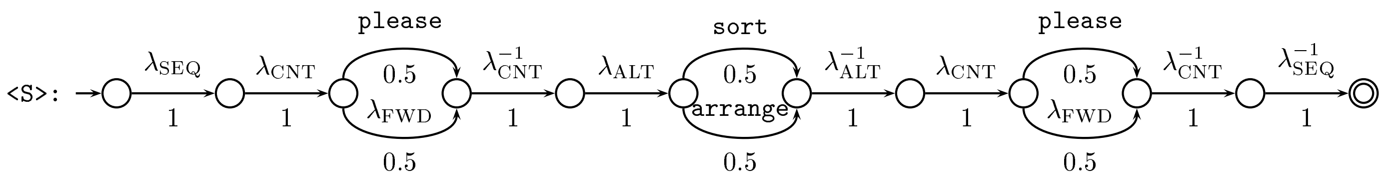


Figure 6.14: Gsg's merging of rule `<S> = [please] sort [please]` and subRHS `arrange`, resulting in `<S> = [please] (sort | arrange) [please]`. Note that the resulting rule is more general than that of Figure 6.13.

6.3.6 Enabling a Natural Dialgoue

As mentioned in §4.2.3, GSG is able to both remember what the user has said as well as shift the focus of attention. This behavior is modeled by the Dialogue Manager and the Interaction History (pictured in Figure 6.1). In addition, to achieve a naturally-sounding dialogue, it is important to minimize the number of choices presented to the user, and be flexible in understanding the concepts the user may be referring to. Finally, mechanisms should be in place to cope with the ever-present ambiguity of natural language. All of these points are discussed below.

6.3.6.1 Dialogue Manager

The Dialogue Manager mantains a stack of learning epiosodes, which allows for the mixed-initiative nature of the conversation: if the user volunteers some information during a learning episode, a new Parse Tree Builder object is pushed onto the stack and becomes the current focus of attention. Once that information is processed, resulting in, say, a new semantic mapping, the previous Parse Tree Builder is reëvaluated and regains the focus of attention.

6.3.6.2 Interaction History

The Interaction History is a database of the sentence-meaning pairs given by the user during the current session. This serves as a memory bank and thus preëmpts the system from asking a question that the user has already answered. Note that it stores both positive associations (e.g., *christmas* is a `<timePoint>`) as well as negative ones (e.g., *retrieve* is not a `<_SEND>` or `<_FORWARD>` (since, for example, they were not selected by the user in a multiple-choice question)).

However, in some cases, a question, although present in the database with a positive association, is not considered as an answer. This is the case if the system is specializing. For example, given *sing* MEANS *play*, and assuming `<play>` has been established as the anchor mother, when pursuing strategy Is-a Daughter Search `<_PLAY>` is found as a candidate (because the Ontology records that `<_PLAY>` Is-a `<play>`). But `<_PLAY>` should not be filtered out just because it does not match the answer that the history records for the word *play* (namely, `<play>`). Therefore, when the suggestion is less abstract than the current answer provided by the user, it is allowed as a choice. In other words, the history mantains the most specific answer given so far for a particular phrase.

6.3.6.3 Formulating Questions

A key feature of GSG is the *interactive* nature of the construction of an interpretation for an extragrammatical utterance. Why interactive? Rather than exposing the system to diminishing success as (erroneous) decisions are compounded, the user is invoked at certain decision points (e.g., in establishing the anchor mother) to secure pivotal elements in the working interpretation. Because, as [Manning and Schütze, 1999] note, what makes semantic analysis hard is the nested nature of its result (parse tree): One has to make many consecutive, dependent correct decisions in order to succeed, and the overall success rate is basically the n^{th} power of the success rate of the individual decisions, a number that easily becomes small.

Therefore GSG poses questions to the user at different stages of the construction of the interpretation. For example, when the system is hypothesizing daughter NTs under the anchor mother to cover an unparsed segment of the evidence (see §6.3.2.1), two different strategies are possible: One is to gather all NTs under the anchor mother that license a T and ask a single multiple-choice question to the user (with typically many choices); the other is to directly engage the user in the top-down traversal of the Ontology, from the anchor mother, to the correct, pre-terminal NT, i.e., by asking the user a few multiple-choice questions (with typically very few choices). The advantage of the first strategy is that it only requires a single question to the user. However, the number of choices, in this multiple-choice question, is, roughly, exponential on the ontological distance between the anchor mother and the leaf nodes. On the other hand, the second option, does require, roughly, a number of questions equal to the ontological distance between the anchor mother and the leaf nodes, but each of the questions will have no more than the number of ontological daughters of the node at hand. Thus, the second strategy is preferred: Even if it involves more than one round of questions, the search is directed by the end-user, thereby avoiding the compounding of errors.

6.3.6.4 Fuzzy Matching of NTs

To increase flexibility, GSG allows the end-user to refer to NTs in a "fuzzy" way, e.g. *send mail* will match `<sendMail>`, and *send* alone will match, in ranked order, `<_SEND>` and `<sendMail>`. This is achieved by mantaining an n-gram model (specifically, a bigram) for each NT. The training events for the NT-dependent name n-gram are the segmented words comprising the NT name in question. For example, for `<sendMail>` the training corpus for its name n-gram consists of <*send*, *mail*>.

6.3.6.5 Coping with Ambiguity

Finally, ambiguity, this omnipresent and difficult problem for any branch of Natural Language Processing, is dealt with in GSG in a way that is consistent with the rest of the decision making processes, namely, by a combination of stochastic models and direct interaction with the user. For example, in the following interaction, two cases of ambiguity (*a* and *to bob*) are found, and asked to be disambiguated by the user.

Interaction 6.3.6.5

```
> create a new message to bob
I don't understand right away what you mean but let me guess...
"create a new message to bob" is a way to express:
     1.  move mail, e.g.  "move to <folderName>"
     2.  sort mail, e.g.  "sort"
     3.  count mail, e.g.  "count mail"
     0.  None of the above
> 0
"create a new message to bob" is a way to express:
     1.  forward mail, e.g.  "forward"
     2.  read mail, e.g.  "read"
     0.  None of the above
> 0
"create a new message to bob" is a way to express:
     1.  list mail, e.g.  "list"
     2.  compose mail, e.g.  "compose"
     3.  send mail, e.g.  "send"
     0.  None of the above
> 2
By "a" do you mean:
     1.  article, e.g.  "a"
     2.  integer cardinal 1, e.g.  "1"
     3.  string a, e.g.  "a"
> 1
By "to bob" do you mean:
     1.  destination folder, e.g.  "to <folderName>"
     2.  recipient, e.g.  "<personName>"
> 2
Is "create" a way to express compose, e.g.  "compose"?
     1.  Yes
     2.  No
> 1
Thanks for teaching me the meaning of "create a new message to bob"!
I understand "create a new message to bob"
```

Obviously, some distinctions are more important than others. For example, distinguishing <destinationFolder> from <recipient> is more important than differentiating between <_ARTICLE>, <INTEGER_CARDINAL__1> and <_STRING__a>. However, the current system finds all the ambiguities [25] and cannot model differences in importance.

[25] To be precise, all the ambiguities that cover the same segment in the evidence.

At the same time, if the grammar had been less orthogonal and encoded the fact that, under `<composeMail>`, `<destinationFolder>` does not make sense, the ambiguity would never have arisen.

This last consideration begs an important question: How sensitive is GSG to the nature of the Kernel Grammar?

6.3.7 The Effect of Grammar Design on GSG's Learning Abilities

As Lehman writes in [Lehman, 1989], "Any representation of the grammar biases what is learned from a single event," and a potential pitfall of GSG lies in its heavy reliance on the grammar. On the one hand GSG's approach greatly simplifies the creation and maintenance of an application, because all the domain knowledge is represented by the grammar; but, on the other hand, it makes a good design of the grammar crucial for the correct behavior of the system, as all of its domain-dependent knowledge sources (such as the Ontology and Parsebank) are derived from it. Experience, however, indicates that this is not a real problem because GSG works well with any kind of well-structured grammar (see Chapter 7). Nonetheless, it is the case that the heuristics that determine the nature of a NT–NT relationship may not always be correct or may sometimes be overly sensitive.

For example, Figure 6.15 shows a grammar in which `<_BY>` is present in all expansions of `<sortBy>`. This prevents the sorting criteria (`<sortBy_date>`, `<sortBy_subject>`, `<sortBy_sender>`, etc) from being considered Is-a daughters of `<sortBy>` and may impact the kind of rule that will be acquired. For example, given the utterance *sort by ascending size means sort by size ascending*, a MEANS operation *sort by ascending size* MEANS *sort by size ascending* is triggered. Then, the evidence for the extragrammatical sentence *sort by ascending size* is found to be `<_SORT> <_BY> <sortMode_ascending> <sortBy_size>`, and, upon establishing `<sortMail>` as the anchor mother, the final subRHS acquired, after Vertical Generalization, is `<_SORT> <_BY> <sortMode> <sortBy_size>`. That is, the fact that the sorting mode can be inserted between *by* and the sorting criterion has not been generalized to all sorting criteria but only to `<sortBy_size>`.

In contrast, Figure 6.16 shows a grammar fragment very similar to that in Figure 6.15, except that an additional, explicit grouping of the sorting criteria under `<_SORT_BY>` is present. In this case, the sorting criteria are considered Is-a daughters of `<_SORT_BY>` and thus, when it comes to vertically generalizing evidence `<_SORT> <_BY> <sortMode_ascending> <sortBy_size>` under `<sortMail>`, not only is `<sortMode_ascending>` generalized to `<sortMode>`, as in the previous case, but also `<sortBy_size>` is generalized to `<_SORT_BY>`.

```
<sortBy> =
        <_BY> [the]
        ( <sortBy__date>
        | <sortBy__subject>
        | <sortBy__sender>
        | <sortBy__recipient>
        | <sortBy__size>
        )
        [of [the] <_MAIL>];
<_BY> =
          by
        | according [to]
        | following;
<sortMode> =
        [in] ( <sortMode__ascending> | <sortMode__descending>) [mode];
<sortMode__ascending>  =  ascending  | increasing;
<sortMode__descending> =  descending | decreasing;
```

Figure 6.15: First grammar fragment to illustrate effect on GSG's generalization abilities. Note that sorting criteria (`<sortBy__date>`, `<sortBy__subject>`, etc.) are not grouped. (Cf. Figure 6.16.)

Therefore, since the final subRHS acquired is `<_SORT> <_BY> <sortMode> <_SORT_BY>`, a subsequent sentence such as *sort by descending date* would be correctly understood.
An easy solution to such sensitivity would be to allow the grammar writer to override the default heuristics and explicitly set the nature of the ontological arc. For example, one could add `/* @ontArc <sortBy__size> Is-a <sortBy> */` to the grammar in Figure 6.15 and achieve the same generalization as with the grammar in Figure 6.16.

6.4 GSG in Action

Having described the components and algorithms of GSG, let us now turn to their actual behavior when the system is fed with real-life grammars and sentences.
Chapter 7 explains a formal evaluation of GSG, but here a brief typology of the interactions is presented, followed by a detailed example of a learning episode.

6.4.1 Typology of User Interactions

The effect of user interactions can be classified into two basic categories: learning episodes and detections of conflicting information. The first case is by far the most common, and the one that results in the acquisiton of rules. (See also §7.3 and §E for more examples of interactions.)

```
<sortBy> =
          <_BY> [ the] <_SORT\_BY> [of [the] <_MAIL>];
<_SORT_BY> =
          <sortBy__date>
        | <sortBy__subject>
        | <sortBy__sender>
        | <sortBy__recipient>
        | <sortBy__size>;
<_BY> =
          by
        | according [to]
        | following;
<sortMode> =
        [in] ( <sortMode__ascending> | <sortMode__descending>) [mode];
<sortMode__ascending>  =  ascending  | increasing;
<sortMode__descending> =  descending | decreasing;
```

Figure 6.16: Second grammar fragment to illustrate effect on GSG's generalization abilities. Note that sorting criteria are grouped under `<_SORT_BY>`. (Cf. Figure 6.15.)

6.4.1.1 Learning Episode

An interaction with the user typically results in a learning episode, as in the following example.

Interaction 6.4.1.1a

```
> toss thunder road into cart
I understand "thunder road into cart"
> wrong
"toss thunder road into cart" is a way to express:
     1.  object, e.g.  "great"
     2.  to cart, e.g.  "cart"
     3.  add to cart, e.g.  "add"
     0.  None of the above
> 3
Thanks for teaching me the meaning of "toss thunder road into cart"!
I understand "toss thunder road into cart"
Thanks for teaching me the meaning of "drop thunder road from cart"!
```

In this case, the subRHS *toss* is acquired under `<shoppingcart.addToCart>`, i.e., as a synonym of the existing RHS *add*.

Sometimes what is needed is an extension of the coverage for an NT, as in this example.

Interaction 6.4.1.1b

```
> play again
I understand "play"
> wrong
"play again" is a way to express:
     1.  play, e.g.  "i want hear"
     2.  softer, e.g.  "softer"
     0.  None of the above
> 1
Is "play again" a way to express play, e.g.  "play"?
     1.  Yes
     2.  No
> 1 Thanks for teaching me the meaning of "play again"!
I understand "play again"
```

Here, the subRHS *play again* is acquired under `<shoppingcart.play>`.

6.4.1.2 Detection of Conflicting Information

In some instances GSG is able to detect a conflict or inconsistency with its current knowlege, as in the following two examples.

Interaction 6.4.1.2a

```
> play
I understand "play"
> wrong
Is "play" a way to express play, e.g.  "i want hear"?
     1.  Yes
     2.  No
> 2
"play" is a way to express:
     1.  reset, e.g.  "reset"
     2.  list all songs, e.g.  "what do you have"
     3.  get information, e.g.  "list information"
     0.  None of the above
> 3
Sorry, I can't learn what you mean as it conflicts with my existing
knowledge.
```

In this case, since the utterance is already fully parsed under an anchor mother (`<musicbox.play>`) which is different from the one that the user suggests (`<musicbox.getInformation>`), GSG refuses to learn, as adopting this meaning equivalence may be too disruptive to the Kernel Grammar.

Interaction 6.4.1.2b

```
> send means forward
Learning from MEANS...
"send" is a way to express:
     1.  forward mail, e.g.  "forward"
     2.  forward, e.g.  "forward"
     0.  None of the above
> 1
Sorry, I can't learn what you mean as it conflicts with my existing
knowledge.
```

Here the same situation arises since the left argument of the *send* MEANS *forward* operation is not extragrammatical and its anchor mother (`<sendMail>`) differs from the one for the right argument (`<forwardMail>`).

It should be noted that, if the interpretation of the right argument of a MEANS or ISA operation consists of more than one top-level NT (as in, say, *mail means compose and send*), the system is unable to learn and an appropriate message is issued. Also, if the system exhausts candidates for anchor mother, it determines that the extragrammatical utterance is out of domain (see §7.3.2.2 for an example).

6.4.2 Detailed Example

Finally, to illustrate the inner workings of GSG in action, what follows is the detailed analysis of an interaction drawn from the E-Mail Task.[26]

Figure 6.17 shows the example dialogue and Figure 6.18 presents the relevant fragment of grammar used to analyze the input.

Let us proceed step by step. The user inputs (U1) (*retrieve all messages from bob that were sent after christmas*, see Figure 6.17) and, as it turns out, neither *retrieve*, nor *bob*, nor *christmas* are in the grammar, and no full parse of (U1) can be found. Rather than responding with *I don't understand — please rephrase*, GSG sets its predictive machinery in motion with the goal of constructing a full interpretation for (U1).[27]

The first step is to reparse (U1) in all-top mode, i.e., with all NTs able to stand at the root of a parse tree. This produces a set of still unparsed words, such as *retrieve*, and parse subtrees, such as *from bob* being parsed under `<_SENDER>` (via grammar rules (R6) to (R10)) and, ambiguously, under `<sourceFolder>` as well (via grammar rules (R11) and (R12)).

Then, the full content of the chart is taken as evidence by the Prediction Models to postulate the overall meaning of the original utterance.

[26]In a slightly different, earlier form from the one included in §C.2.

[27]The hypothesized interpretation assumes that a single parse tree will cover the entire utterance.

(U1) U: retrieve all messages from bob that were sent after christmas

(U2) S: I don't understand right away what you mean but let me guess...
"retrieve all messages from bob that were sent after christmas" is a way to express:

1. send mail
2. reply mail
3. read mail
0. None of the above

(U3) U: none

(U4) S: "retrieve all messages from bob that were sent after christmas" is a way to express:

1. move mail
2. list mail
0. None of the above

(U5) U: list

(U6) S: By "from bob" do you mean:

1. source folder
2. sender

(U7) U: sender

(U8) S "christmas" is a way to express:

1. yesterday (date relative yesterday)
2. tomorrow (date relative tomorrow)
3. today (date relative today)
0. None of the above

(U9) U: by christmas i'm referring to december twenty fifth

(U10) S: Thanks for teaching me the meaning of "christmas"!
Thanks for teaching me the meaning of "retrieve all messages from bob that were sent after christmas"!
I understand "retrieve all messages from bob that were sent after december twenty fifth"

(U11) U: retrieve last email to mary before christmas

(U12) S: I understand "retrieve last email to mary before december twenty fifth"

Figure 6.17: Example dialogue between a user (U) and the system (S) in the E-Mail Task. (See full explanation in §6.4.2.)

<table>
<tr><td>(R1)</td><td><listMail> = [<_VERB_DESIRE>] <_LIST> [<_TO_FOR_ME>] <_MAIL_ARGUMENTS>+;</td></tr>
<tr><td>(R2)</td><td><moveMail> = [<_VERB_DESIRE] <_MOVE> <_MAIL_ARGUMENTS>* [<sourceFolder>] <destinationFolder>;</td></tr>
<tr><td>(R3)</td><td><_LIST> = list | get;</td></tr>
<tr><td>(R4)</td><td><_MOVE> = move;</td></tr>
<tr><td>(R5)</td><td><_MAIL_ARGUMENTS> = <_SENDER> | <_RECIPIENT> | <_SUBJECT> | <_DATE> | <_MESSAGE_IDX> | ...;</td></tr>
<tr><td>(R6)</td><td><_SENDER> = [<_SENDER_PRE>] <sender>;</td></tr>
<tr><td>(R7)</td><td><_SENDER_PRE> = from | by;</td></tr>
<tr><td>(R8)</td><td><sender> = <name__STRING> | <emailAddress__STRING>;</td></tr>
<tr><td>(R9)</td><td><name__STRING> = <_PERSON_OR_INSTITUTION_NAME> | <_MAILING_LIST_NAME>;</td></tr>
<tr><td>(R10)</td><td><_PERSON_OR_INSTITUTION_NAME> = <_WILDCARD>+;</td></tr>
<tr><td>(R11)</td><td><sourceFolder> = from <folderName__STRING> [<_FOLDER>];</td></tr>
<tr><td>(R12)</td><td><folderName__STRING> = <_WILDCARD>;</td></tr>
<tr><td>(R13)</td><td><dateRange> = (<_DATE_AFTER_PRE> <dateAfter>) | (<_DATE_BEFORE_PRE> <dateBefore>) | ...;</td></tr>
<tr><td>(R14)</td><td><_DATE_AFTER_PRE> = after | from | since;</td></tr>
<tr><td>(R15)</td><td><dateAfter> = <datePoint__DATE>;</td></tr>
<tr><td>(R16)</td><td><datePoint__DATE> = <_DATE_POINT_ARGUMENT>+;</td></tr>
<tr><td>(R17)</td><td><_DATE_POINT_ARGUMENT> = <datePoint__DATE_RELATIVE> | <datePoint__DATE_FIXED> | ...;</td></tr>
<tr><td>(R18)</td><td><datePoint__DATE_RELATIVE> = <DATE_RELATIVE__yesterday> | <DATE_RELATIVE__tomorrow> | ...;</td></tr>
<tr><td>(R19)</td><td><DATE_RELATIVE__yesterday> = yesterday;</td></tr>
</table>

Figure 6.18: Grammar fragment relevant to the dialogue in Figure 6.17. (See full explanation in §6.4.2.)

The suggestions of the Prediction Models (see (U2) to (U5)[28]) are, in this case, not particularly accurate (the correct choice is presented only in fifth place), but, considering that the head verb (*retrieve*) is not even in the grammar, such a response to (U1) is definitely better than simply rejecting the extragrammatical input.

The effect of (U5) is to select `<listMail>` as (U1)'s anchor mother (the logical root of the overall interpretation). But, in order to complete the parse tree, a few details still need to be filled in. To that effect (U6) is generated to disambiguate *from bob* and (U8) to find the right mapping for *christmas*.

The reasoning behind the seemingly puzzling choices offered by (U8) comes from applying the Parser Predictions strategy: given the context in which an unparsed sequence (in this case, single word) *christmas* appears, i.e., the subtree `<_DATE_AFTER_PRE>` covering *after* (via (R14)), the grammar is traversed to find likely continuations of the context (left context only in this case). Since `<_DATE_AFTER_PRE>` can be immediately followed by `<dateAfter>` (see (R13)) that makes `<dateAfter>` a candidate to cover the unparsed word *christmas*. However, since, according to the Ontology, `<dateAfter>` does not license terminals, a search is performed to find NTs under `<dateAfter>` that permit terminals as immediate daughters. In this case (via (R15) to (R19)) it suggests *yesterday*, *tomorrow*, etc.[29] The user, however, realizing that the system does not directly understand *christmas*, volunteers (U9),[30] from which the mapping in Figure 6.20 is learned.

At this point one may wonder about the fate of the unparsed word *retrieve*, since no question was asked about it. The answer is that GSG need not ask about every single prediction, if the confidence value is high enough. In this case, as soon as `<listMail>` was established (in (U5)) as the anchor mother, a Verbal Head Search strategy was launched to see whether, among the unparsed words, a verb was found that could be placed in a mostly-verb NT[31] directly under `<listMail>`. The result was highly positive and led to the acquisition of the RHS alternative in Figure 6.19.

It is worth recalling (see §4.2.2.1) that there are two kinds of mappings that GSG learns: RHS alternatives and subtree mappings. Learning new RHS alternatives is the preferred way because not only is the new linguistic knowledge better generalized, but also it is incorporated into the Parsebank, and, in turn, into the Prediction Models. This is, precisely,

[28]The options presented in (U2) and (U4) are generated at the same time; the only reason they are split is to prevent overwhelming the end-user, who may be hearing the choices spoken over the telephone.

[29]In fact it suggests `<DATE_RELATIVE__yesterday>`, `<DATE_RELATIVE__tomorrow>`, etc, but it presents an example automatically generated from such NTs.

[30]Obviously "the meaning of Christmas" (cf. cheerful (U10)) may be much more profound than a short-hand for December 25 — but, alas, conveying that is well beyond the simple grammar presented here.

[31]Recall that a "verbness ratio" is automatically computed for each candidate NT by running the POS tagger on automatically generated sentences from the NT in question.

`<_LIST> = list | get` $\boxed{\texttt{| retrieve}}$;

Figure 6.19: Resulting rule for NT `<_LIST>` after the dialogue in Figure 6.17. Boxed alternative highlights the new subRHS acquired (cf. (R3) in Figure 6.18). (See full explanation in §6.4.2.)

the effect of acquiring the subRHS in Figure 6.19: Since the Parsebank and the Prediction Models are updated on-line, the presence of the word *retrieve* in subsequent utterances becomes a strong indicator of `<_LIST>` and, associatively, of `<listMail>`. However, when the source expression can not be mapped into the desired target structure via grammar rules, as in Figure 6.20, the only solution is to remember the equivalence. This type of learning, although definitely useful since the meaning of the source expression will be henceforth remembered, cannot be incorporated into the Prediction Models.

Note also that the rule `<listMail> = <_LIST> <_MAIL_ARGUMENTS>+`, extracted from the final interpretation of (U1), was a candidate too, but it was discarded when the system detected its subsumption by existing rule (R1).

Right after (U9), (U1) is considered fully understood and the interpretation is automatically mapped into the feature structure (FS1)[32] in Figure 6.21, which is then sent to the Back-end Application Manager.

Finally, when (U11) comes in, a correct analysis is produced thanks to the mappings just learned from (U1), and (FS2) in Figure 6.21 is generated.

[32]As sketched in §2.1.1 (footnote 4), the mapping from parse tree to feature structure consists of recursively converting principal NT parse nodes into feature structure attributes. Some additional processing is involved to extract the value of certain attributes, such as dates, numbers and strings. For example, the tree in Figure 6.20 becomes the feature substructure under `<datePoint>` in (FS1).

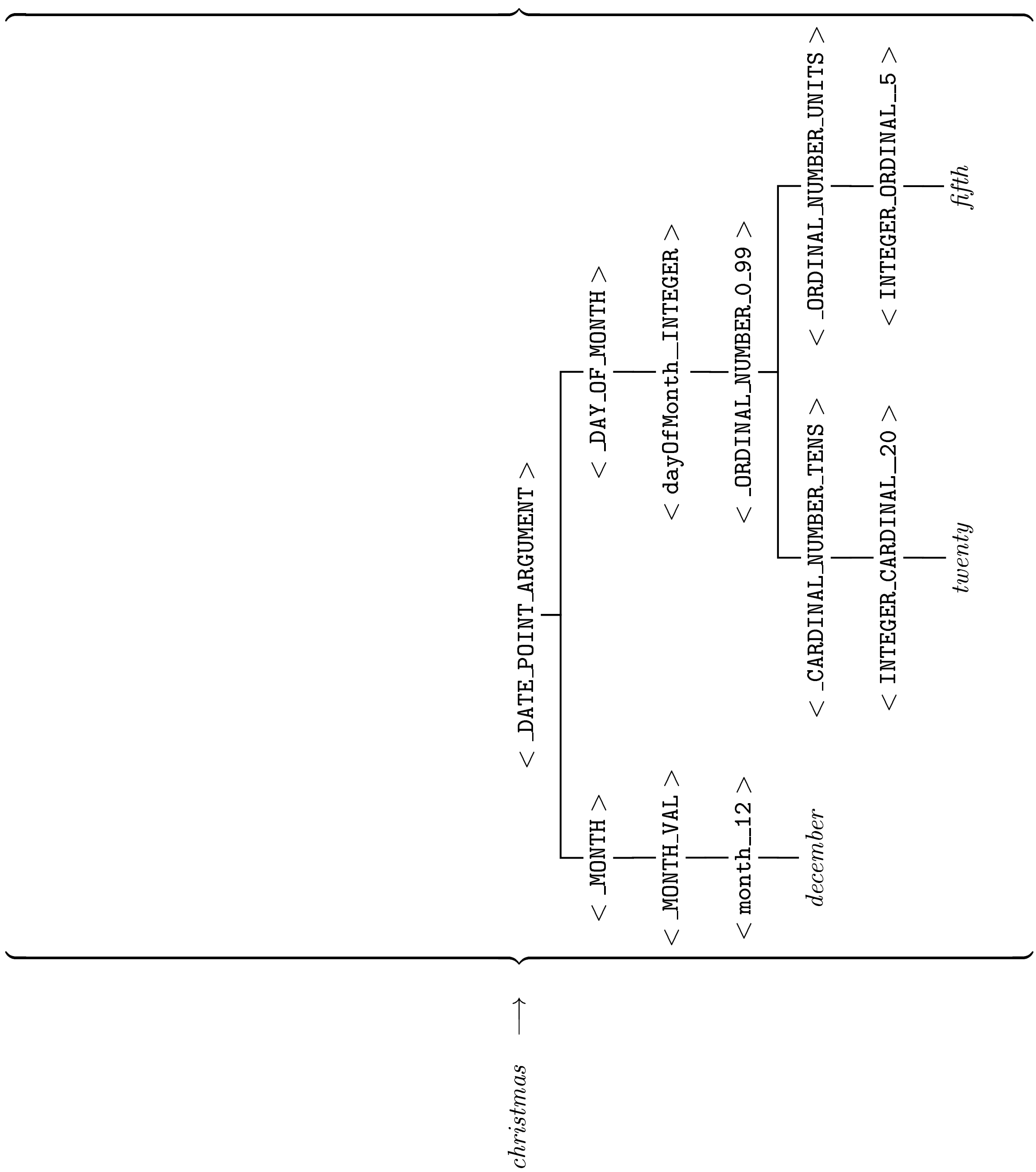


Figure 6.20: Direct mapping learned from the dialogue in Figure 6.17. (See full explanation in §6.4.2.)

```
[ listMail
      messageIndex:  all
      sender
            name bob
      dateRange
            dateAfter
                  datePoint
                        month:  12
                        dayOfMonth:  25 ]
```

(FS1)

```
[ listMail
      messageIndex:  last
      recipient
            name:  mary
      dateRange
            dateBefore
                  datePoint
                        month 12
                        dayOfMonth:  25 ]
```

(FS2)

Figure 6.21: Feature structures sent to the Back-end Application Manager after (U10) and (U12) in Figure 6.17. (See full explanation in §6.4.2.)

Chapter 7

Experiments and Results

This chapter describes the different experiments conducted to evaluate GSG and presents a summary of the results obtained. Appendix D (Utterances from User Sessions) and Appendix E (Learning Episodes) exhaustively list all the user interactions.

7.1 Introduction

The principal claim of this thesis is that it is possible to extract enough information from a simple context-free grammar and be able to correctly extend it, as a side-effect of conversing with the users of the end-application.

To test the validity of this claim and the effectiveness of the learning methods and strategies described in the preceding chapters, a study was conducted in which non-expert[1] end-users interacted with GSG in two different domains.

As we shall see, GSG's overall performance can be qualified as very satisfactory, as it is indeed able to learn, generalize and acquire very reasonable rules through interactions conducted in natural language.

The two domains tested were an email client task, where users can check their e-mail, and a "musicbox" task, where users can listen to and buy songs. The users were undergraduates and staff at Carnegie Mellon University, on a first-come, first-serve basis without any screening.[2]

In each case, a *union grammar* was created as the union of all user grammars, i.e., the kernel grammar plus the subRHSs acquired from the users. In the case of a discrepancy (for example, *show*, to some users, meant `<listMail>`, whereas to others it meant `<readMail>`), a majority vote was taken (and broken at random if tied).[3]

[1] I.e., not versed in computational linguistics.
[2] Except to disallow members of the Language Technologies Institute.
[3] Specifically, such inter-user ambiguities occurred for *check*, *create*, *draft*, *see* and *show*.

Nonterminal	Example
`<countMail>`	*i'd like to know how many messages i received yesterday*
`<listMail>`	*please list all messages about the site visit sent last month*
`<readMail>`	*read bob's latest message*
`<composeMail>`	*compose a new message to bob about the party*
`<sendMail>`	*send mail to bob*
`<forwardMail>`	*forward bob's message to alice*
`<replyMail>`	*reply to cynthia*
`<deleteMail>`	*delete all messages from last year*
`<sortMail>`	*sort mail by date descendingly*

Table 7.1: Main concepts (i.e., top-level NTs) of the E-Mail Task grammar, and examples thereof.

In the E-Mail Task, 56 rules were learned, out of which 48 were unique (i.e., in 8 cases different users triggered the creation of the same rule); in the Musicbox Task, 19 rules, all unique, were acquired. And, as detailed below, the performance of the union grammars on independent test sets shows an increase in semantic accuracy between 20 and 32 absolute percentage points.

7.2 Experiments

To establish the generality of GSG's learning methods it is important to test GSG in a variety of environments. In this case the two applications (E-Mail and Musicbox) and their corresponding grammars present very different approaches, the only commonality being that both grammars are written in the JSGF format.

7.2.1 E-Mail Task

The first task is an interface to an e-mail client. The full grammar is listed in §C.2, but the main concepts of the domain are summarized in Table 7.1. Two example parses in this domain are given in Figures 7.1 and 7.2.

As can be seen from the example parse trees, the E-Mail Task grammar is extremely orthogonal in its design. That is, it basically allows free ordering of all the arguments. It is well known that such a design overgeneralizes and cannot model certain dependencies. For example, in a flight reservation task, the meaning of the following sentences *Is the earliest flight going to Boston on a 747?* and *Is the earliest flight on a 747 going to Boston?* is not the same, yet, an orthogonal, free-order argument design is not able to model the difference because all the arguments (*earliest*, *going to Boston*, and *on a 747*) are at the same level and their order not taken into account. Nonetheless, for restricted domains with rather coarse semantics, this kind of approach is advantageous because the resulting grammar is easier to design and mantain.

As for the experiment itself, Figure 7.3 shows the instructions given to the testers, and Figure 7.4 presents a screenshot of a session.

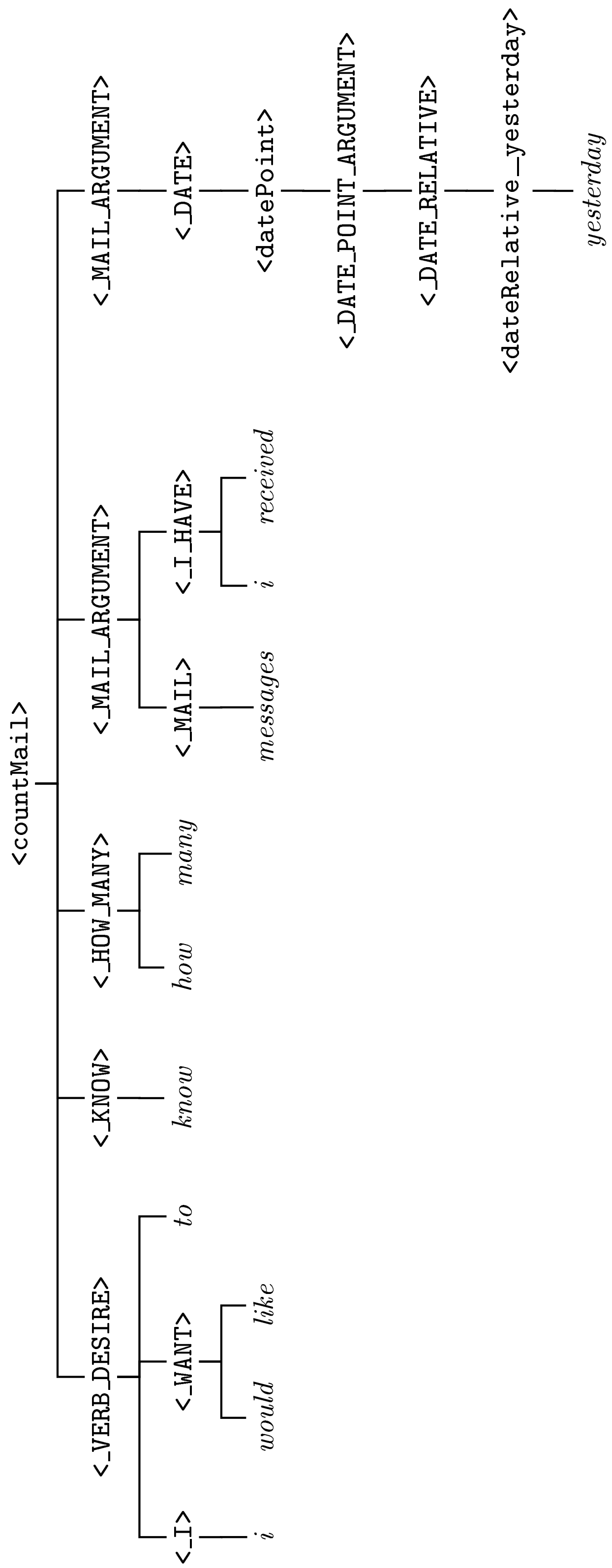


Figure 7.1: Parse of *i'd like to know how many messages i received yesterday* according to the E-Mail Task grammar listed in §C.2.

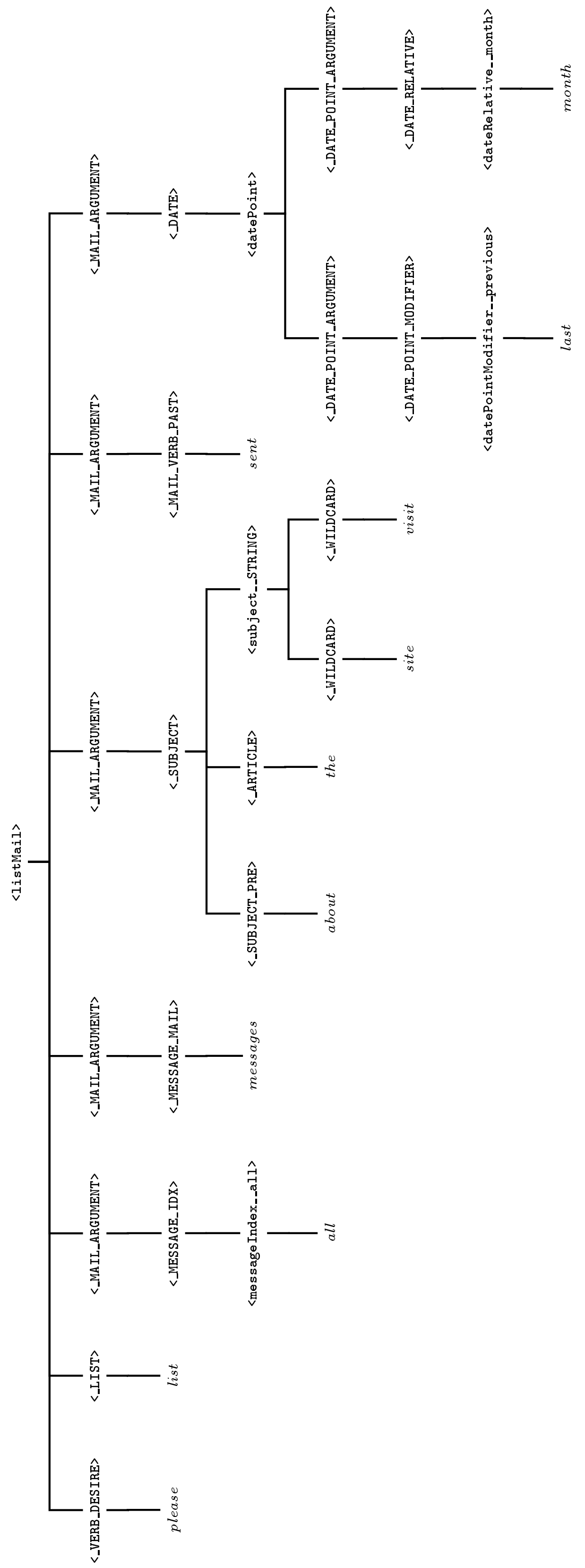


Figure 7.2: Parse of *please list all messages about the site visit sent last month* according to the E-Mail Task grammar listed in §C.2

What you'll have to do is very simple: think about what you'd like to say to your e-mail client (that is, an email reader such as Pine, Mulberry, Eudora, Outlook, etc), type it on the text box and see whether the system understands you. Example sentences:

- *do i have any mail*
- *please show me all messages about the meeting*
- *reply to the last message from peter*
- *sort messages by size*
- *...*

Note that along the way the system may ask you some clarification questions, as it attempts to understand what you mean. Also, please be aware of the following "power commands":

- **wrong**: Tells the system it didn't (fully) understand what you meant.
- **cancel**: Cancels the current learning attempt.
- **ignore**: Ignores the current clarification question.
- *X* **means** *Y*: Teaches the system a meaning equivalence. E.g., *christmas means december twenty fifth.*
- *X* **is a** *Y*: Teaches the system the category of a word or phrase. E.g., *my sister is a person.*

During your session please make sure that, at least, you:

- Check your mail.
- Sort all your messages by date.
- Get rid of some messages.
- Reply to Cynthia.
- Create a new message to Bob.

Figure 7.3: Instructions given to the users for the E-Mail Task.

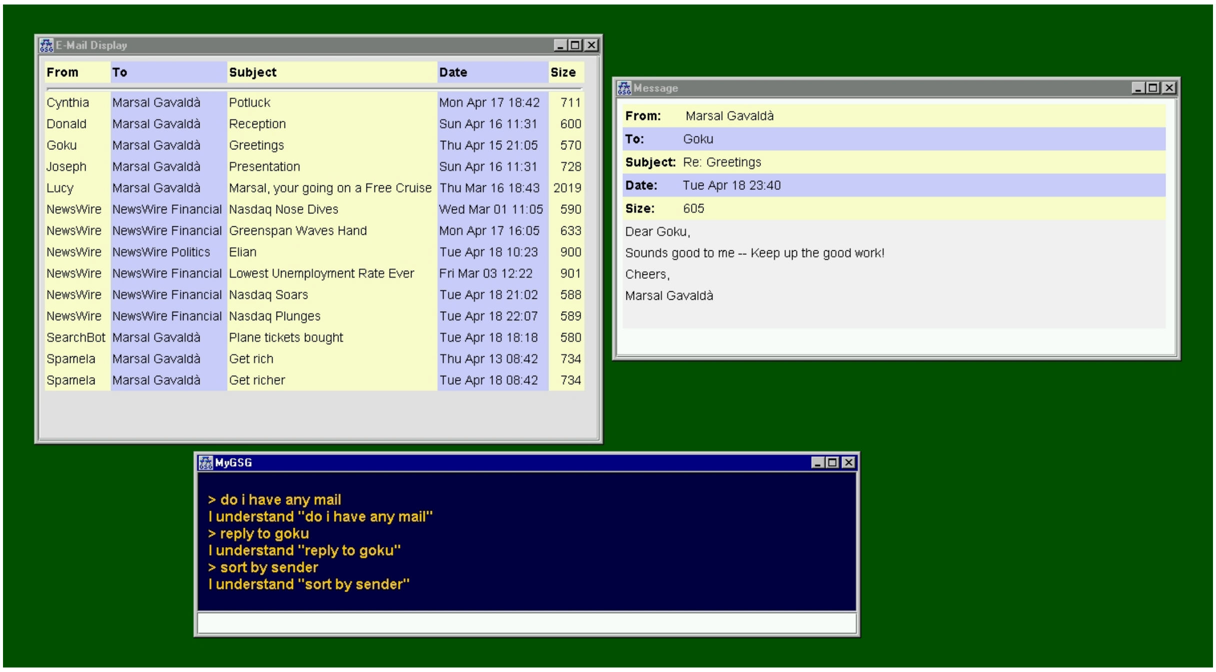


Figure 7.4: E-Mail Task screenshot. The lower window is the GSG interface; user input is preceded by a '>'. The upper left window shows the e-mail inbox. The upper right window popped up as a result of the sentence *reply to goku*.

7.2.2 Musicbox Task

The second task is an interface to a music database, MP3 player, and shopping cart. The approach to grammar design in this second application is very different from the one in the E-Mail Task. The Musicbox Task was developed by Matthias Denecke using the principles of typed feature structures and the grammar contains many independent top-level concepts that are then unified to determine the satisfiability of dialogue goals. A dialogue goal can be, on the one hand, either *enabled* or *disabled*, and, on the other, if it is enabled, it can be either *neutral* (no information present), or *selected* (current information is compatible with goal), or *deselected* (current information is not compatible with goal), or *determined* (current information leaves only one goal compatible), or *finalized* (all information requirements of uniquely determined goal are met and action associated with the goal fires). Then, at runtime, history accumulates from multi-tree interpretations that give rise to multiple typed feature structures and it also accumulates over utterances, until a goal is finalized,[4] at which point the history is reset.

For a full account of Denecke's approach see [Denecke, 1997], but let the brief explanation above serve to illustrate that GSG's strategies are general enough to support a variety of domain and grammar development philosophies.

Another point worth noting is that in this application GSG was able to take Denecke's notion of goal activity into account as an additional external knowledge source employed to rank hypotheses: Goals were mapped back to NTs, and NTs whose goals were disabled in the current state of the application (say, `<continue>` when the player is playing) were filtered out. Of course, this particular scheme only works if there is a one-to-one correspondence between application goals and grammar NTs, but it serves to illustrate that GSG can incorporate external knowledge sources, and in the case where such one-to-one mappings would not exist, a stochastic model could be built so that NT probabilities would depend on the dialogue state (see §8.3.2 for a more formal proposal).

The static parts of the grammars[5] for the Musicbox Task are listed in §C.3, but, at runtime, the Musicbox grammar is dynamically augmented with entries from a live database containing the song titles, artist, album, country and language information, as well as pointers to the audio files. The main concepts of the domain are summarized in Table 7.2 and an example parse is shown in Figure 7.5. Note that, given the way that the grammars are structured, it is very common to find cross-grammar, multiple-tree interpretations.

[4]Or all goals become deselected, which triggers an error message.

[5]Note GSG's ability (following the specifications of the JSAPI) to take the multiple JSGF grammars and operate on all of them at once. In this case, the application consists of three grammars: Generic, Musicbox, and Shopping Cart.

Nonterminal	**Example**
`<listAllSongs>`	*please show me all available songs*
`<getInformation>`	*give me some info*
`<play>`	*please play*
`<pause>`	*pause*
`<stop>`	*please stop*
`<continue>`	*go on please*
`<fastForward>`	*fast forward*
`<fastBackward>`	*fast backward*
`<rewind>`	*please rewind*
`<louder>`	*play louder*
`<softer>`	*please play softer*
`<mute>`	*mute*
`<object>`	*a groovy song*
`<addToCart>`	*buy*
`<removeFromCart>`	*remove*
`<checkout>`	*checkout please*
`<balance>`	*what's my balance*
`<toCart>`	*into my shopping cart*
`<fromCart>`	*from my cart*

Table 7.2: Main concepts (i.e., top-level NTs) of the Musicbox Task grammar, and examples thereof.

As for the experiment, Figure 7.6 shows the instructions given to the testers, and Figure 7.7 presents a screenshot of a session.

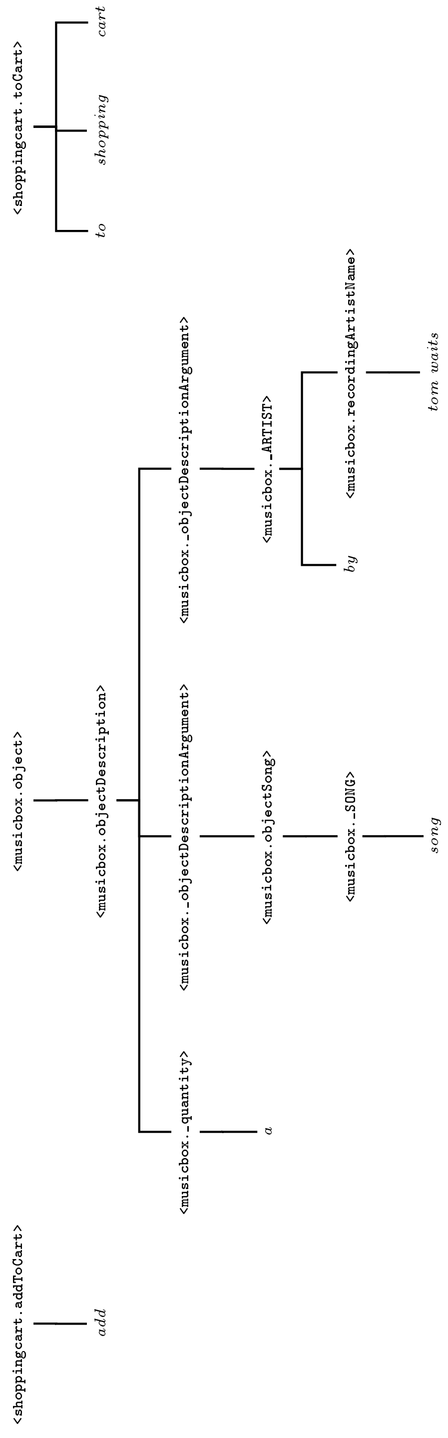


Figure 7.5: Parse of *add a song by tom waits to shopping cart* according to the Musicbox Task grammar in §C.3.

What you'll have to do is very simple: imagine that you have \$20 dollars to spend in music and you want to buy a few songs from the musicbox. Example sentences:

- *what do you have*
- *play my all*
- *add my all to shopping cart*
- *what's my current balance*
- *i want to check out now*
- *...*

Note that along the way the system may ask you some clarification questions, as it attempts to understand what you mean. Also, please be aware of the following "power commands":

- **wrong**: Tells the system it didn't (fully) understand what you meant.
- **cancel**: Cancels the current learning attempt.
- **ignore**: Ignores the current clarification question.
- X **means** Y: Teaches the system a meaning equivalence. E.g., *track means song.*
- X **is a** Y: Teaches the system the category of a word or phrase. E.g., *French is a language.*

Figure 7.6: Instructions given to the users for the Musicbox Task.

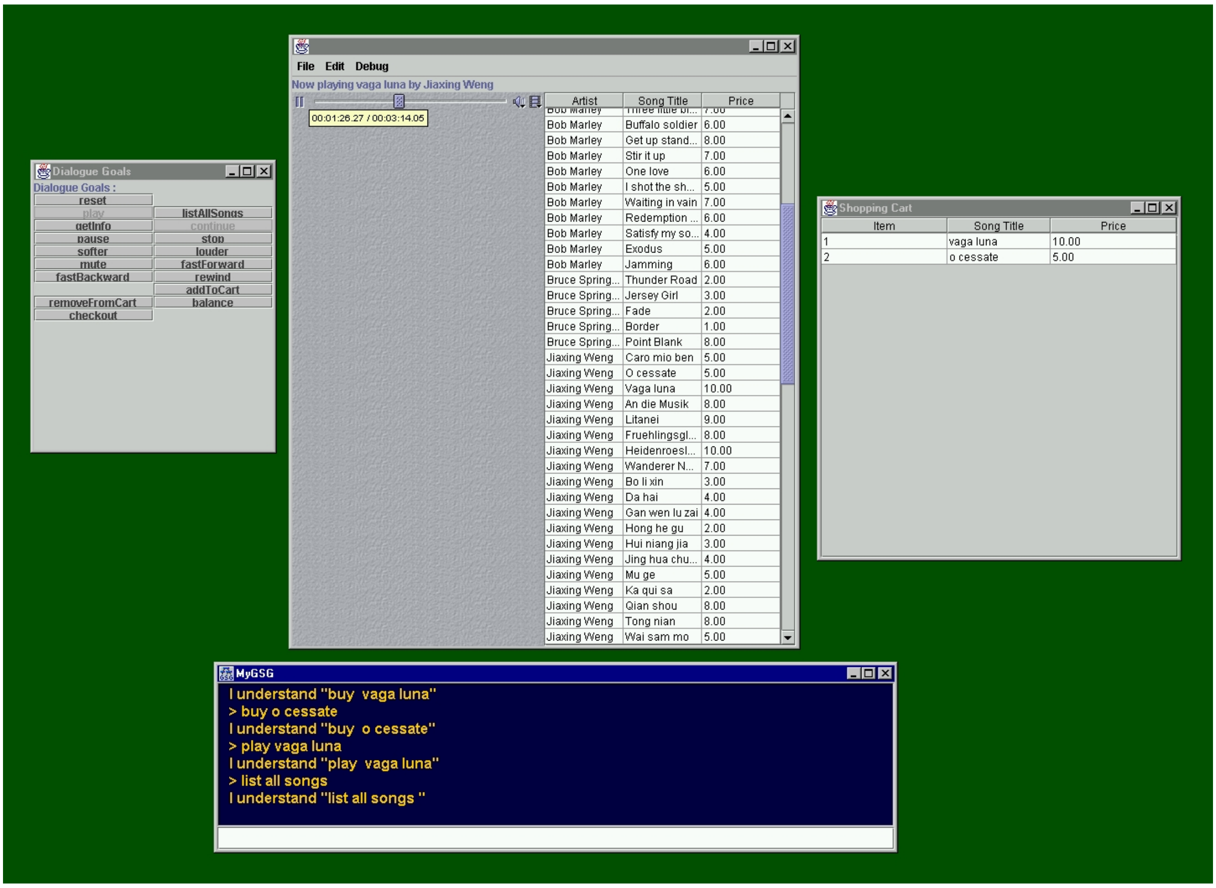


Figure 7.7: Musicbox Task screenshot. Again, the lower window is the interface to GSG, with user input preceded by '>'. The upper left window depicts the current dialogue goals: enabled goals in black and disabled goals in gray (in this case `<play>` and `<continue>`, since the application is already playing). The upper middle window lists available songs and shows playing status. The upper right window displays the contents of the shopping cart.

7.3 Example Interactions

Listing all the interactions that were recorded in the experiments would take too much space and not be as informative as the learning episodes (which are presented in full detail in Appendix E). But, in order to have a sense of the kind of "conversation" that the users engaged in with GSG, what follows is a sample of the types of interactions that took place in the user sessions.

7.3.1 Successful

Most of the interactions in the user studies can be classified as successful, for they trigger the acquisition of correct rules. This is exemplified by the two interactions below.

Interaction 7.3.1a

```
> see email from goku
I don't understand right away what you mean but let me guess...
"see email from goku" is a way to express:
     1.  count mail, e.g.  "count"
     2.  list mail, e.g.  "list"
     3.  send mail, e.g.  "send"
     0.  None of the above
> read email from goku
By "read email from goku" are you referring to:
     1.  read, e.g.  "read"
     2.  read mail, e.g.  "read"
     3.  email address part string, e.g.  "a"
     0.  None of the above
> 2
Is "see" a way to express read, e.g.  "read"?
     1.  Yes
     2.  No
> 1
Thanks for teaching me the meaning of "see email from goku"!
I understand "see email from goku"
```

As detailed in learning epiosode LE e.2.1 (see §E.1.2.2) this interaction triggered the acquisition of the subRHS *see* under NT `<voiceEMail._READ>`.

Interaction 7.3.1b

```
> get rid of all messages from spamela
I understand "get"
> wrong
"get rid of all messages from spamela" is a way to express:
     1.  list mail, e.g.  "list"
     2.  count mail, e.g.  "count"
     3.  sort mail, e.g.  "sort"
     0.  None of the above
> 0
"get rid of all messages from spamela" is a way to express:
     1.  delete mail, e.g.  "delete"
     2.  reply mail, e.g.  "reply"
     0.  None of the above
> 1
Is "rid" a way to express delete, e.g.  "delete"?
     1.  Yes
     2.  No
> get rid of means delete
Learning from MEANS...
Is "get rid of" a way to express delete, e.g.  "delete"?
     1.  Yes
     2.  No
> 1
Thanks for teaching me the meaning of "get rid of"!
I understand "get rid of"
Thanks for teaching me the meaning of "get rid of all messages from
spamela"!
I understand "get rid of all messages from spamela"
```

In this case the user volunteered the correct segmentation and the entire expression *get rid of* was acquired under NT <voiceEMail._DELETE>. (See LE e.10.4 in §E.1.10.2 for more details.)

7.3.2 Neutral

In some occasions an interaction does not lead to the creation of a new rule. This can happen for a few reasons: the utterance is out of domain (i.e., not supported by the end-application), the utterance is partially parsed but with the same meaning, or the utterance conflicts with existing knowledge.

7.3.2.1 User Cannot Map Own Intention onto Task Concept

Sometimes users are not able to match their intentions to the existing concepts defined for the task, as in the following example.

Interaction 7.3.2.1

```
> check messages
I don't understand right away what you mean but let me guess...
"check messages" is a way to express:
     1.  list mail, e.g.  "list"
     2.  count mail, e.g.  "count"
     0.  None of the above
> 1
"check" is a way to express:
     1.  ask to check, e.g.  "do i have"
     2.  list, e.g.  "list"
     0.  None of the above
> cancel
Ok, back to square one
```

In this case the user's "world view" did not allow her to equate *check* to either *ask to check* or *list* and decided to terminate the entire learning episode.

7.3.2.2 Utterance is Out of Domain

Other times, the user's intention may truly lie beyond the application's (and, correspondingly, the grammar's) ability, as in the example below.

Interaction 7.3.2.2a

```
> delete message from lucy
I understand "delete message from lucy"
Message deleted.
> undelete message from lucy
I don't understand right away what you mean but let me guess...
"undelete message from lucy" is a way to express:
     1.  count mail, e.g.  "count"
     2.  list mail, e.g.  "list"
     3.  reply mail, e.g.  "reply"
     0.  None of the above
> 0
"undelete message from lucy" is a way to express:
     1.  read mail, e.g.  "read"
     2.  compose mail, e.g.  "compose"
     3.  send mail, e.g.  "send"
     0.  None of the above
> 0
"undelete message from lucy" is a way to express:
     1.  forward mail, e.g.  "forward"
     2.  delete mail, e.g.  "delete"
     3.  sort mail, e.g.  "sort"
     0.  None of the above
> 0
Sorry, I can't learn what you mean.  Maybe what you are trying to say is
beyond my abilities.
```

The user was trying to undo her last action but it turned out that such a command was not supported by the end application. After many attempts, the system was able to detect that the utterance was in fact out of domain, as no anchor mother could be found.

Interestingly enough, the self-assessed limitations of the system may sometimes be overridden by an eager user, as in the following example.

Interaction 7.3.2.2b

```
> put song in my shopping basket
I understand "song"
> wrong
"put song in my shopping basket" is a way to express:
     1.  to cart, e.g.  "cart"
     2.  track title, e.g.  "my all"
     3.  from cart, e.g.  "from cart"
     0.  None of the above
> 1
Sorry, I can't learn what you mean as it is not supported by the end
application.
> shopping basket means shopping cart
Learning from MEANS...
Is "shopping basket" a way to express to cart, e.g.  "cart"?
     1.  Yes
     2.  No
> 1
Thanks for teaching me the meaning of "shopping basket"!
I understand "shopping basket"
> put song in my shopping basket
I understand "song in my shopping basket"
> wrong
"put song in my shopping basket" is a way to express:
     1.  object, e.g.  "great"
     2.  to cart, e.g.  "cart"
     3.  add to cart, e.g.  "add"
     0.  None of the above
> 3
Thanks for teaching me the meaning of "put song in my shopping basket"!
I understand "put song in my shopping basket"
```

7.3.2.3 Utterance is Partially Matched

Sometimes, even though the system is not able to parse the entire sentence, what it understands (i.e., the partial analysis) is close enough to the meaning of the entire sentence. If the user accepts it (e.g., by not saying *wrong*) no learning episode is triggered. Here is an example.

Interaction 7.3.2.3

```
> a little louder
I understand "louder"
```

Of course, GSG could automatically acquire subRHS *a little louder* under `<musicbox.louder>` but, in that case, (*i*) it may have to bother the user with clarification questions as it determines the substructure of the analysis, and (*ii*), more importantly, henceforth it would give the false impression that it understands *a little louder* as well as *louder*, i.e., that it is making a nuanced distinction that is in fact not supported by the end application (since the execution of `<louder>` raises the volume by a fixed amount).

7.3.2.4 Utterance Conflicts with Exisiting Knowledge

As explained in §6.4.1.2, in order to prevent not only catastrophic results from malicious users (such as *list means delete*) but also slight conflicts inherent in the carving up of the world that a particular application and grammar entail, GSG rejects new rules that contradict existing knowledge. Her is an example.

Interaction 7.3.2.4

```
> i want to send an email to bob
I understand "i want to send an email to bob"
> wrong
Is "i want to send an email to bob" a way to express send mail, e.g.
"send"?
     1.  Yes
     2.  No
> 2
"i want to send an email to bob" is a way to express:
     1.  count mail, e.g.  "count"
     2.  list mail, e.g.  "list"
     3.  read mail, e.g.  "read"
     0.  None of the above
> 0
"i want to send an email to bob" is a way to express:
     1.  compose mail, e.g.  "compose"
     2.  forward mail, e.g.  "forward"
     3.  reply mail, e.g.  "reply"
     0.  None of the above
> 1
Sorry, I can't learn what you mean as it conflicts with my exisiting
knowledge.
```

It seems that in the user's mind, sending an e-mail to someone means composing it (and probably actually sending it afterwards). However, since according to the design of the application as reflected in the Ontology, sending an e-mail corresponds to a different concept from that of composing an e-mail. Therefore, when the two meanings are equated GSG detects a conflict and refuses to learn.

7.3.3 Unsuccessful

In a few occasions an interaction triggers the acquisition of a wrong rule, as in the following example.

Interaction 7.3.3

```
> how much is lullaby
I understand "lullaby"
Maybe you are trying to say:
     1.  add to cart, e.g.  "add"
     2.  rewind, e.g.  "rewind"
     3.  balance, e.g.  "balance"
     0.  None of the above
> 1
Learning from ISA...
Thanks for teaching me the meaning of "how much is"!
```

It seems that the user's intention changed from wanting to know how much the song *Lullaby* cost, to placing it into the shopping cart, or rather, the user may have thought that placing it into the shopping cart may be the only or easiest way to obtain the price for it. In any case, the end result is the acquisition of *how much is* under `<shoppingcart.addToCart>` (see LE m.2.2 in §E.2.2.2 for details).

7.4 Results

As mentioned before, the utterances from the user sessions are available in Appendix D, and the exhaustive listing of the learning epiosodes in Appendix E. What follows is a summary of these results, which includes statistics on the user sessions, analysis of the degree of exploration of the domain, results on the semantic accuracy of the resulting grammars on independent corpora, and analysis of the types of rules acquired.

7.4.1 E-Mail Task

Table 7.3 presents a tally of the user sessions for the E-Mail Task. The explanation of each row is as follows.

- **Duration**: duration of the session, in minutes.
- **Number of utterances**: number of utterances directed to the end-application, that is, not counting meta-utterances such as *wrong*, or answers to clarification questions.
- **Number of learning episodes**: number of attempts to learn the meaning of utterances. This number is of course less than or equal to the number of utterances.

- **Average number of choices**: average number of choices presented to the user per learning episode. For example, a multiple-choice question with three questions plus a generic "none of the above" counts as three choices[6] and a binary (yes/no) question counts as one choice. More exactly, in the multiple-choice case, the number of choices counted is the number of choices presented to the user until the correct choice is presented, i.e., the rank of the correct choice.

- **Number of rules learned**: number of distinct grammar subRHSs acquired in the session. The number of rules learned is not determined by the number of learning episodes because (*i*) a learning episode may not trigger the acquisition of any rule (for example, when it is realized that the input utterance is out of domain, or when a newly acquired rule is re-used), and (*ii*) a learning episode may trigger the acquisition of more than one subRHS (since more than one rule candidate extracted from the hypothesized parse tree may pass the subsumption, ambiguity, and parsebank disruption tests and added to the grammar).

- **Average GSG score**: average score of the behavior of the system per learning episode and acquired rule, according to the scoring table in Table 7.4. A score is given to each subRHS acquired (or lack thereof); therefore the denominator in the computation of the average is the sum of the number of learned subRHSs plus the number of learning episodes that did not trigger any rule acquistion. Also note that a minus sign after the score indicates that the resulting merged rule could be expressed by a more compact rule.

Looking at Table 7.3 one could argue that the user sessions, at an average duration of fifteen minutes, are not long enough to test the system. However, given the self-bounded nature of an individual's language[7] system performance will, if anything, increase over time, as the ratio of extragrammatical utterances decreases. In fact, the existence of "hits" by the newly acquired rules[8] even within a short session, provides additional, if anectodal, support for this principle.

[6]One could argue that it should count as four choices but the reasoning is that "none of the above" is more of a meta-choice, such as the ever-possible responses of "cancel" or "ignore."

[7]As demonstrated in [Lehman, 1989], e.g., "It is proven that user canonical forms grow in a *self-bounded* fashion, that is, an individual's language usage converges on a stable subset of the language, as opposed to unbounded growth" and "Self-bounded behavior was demostrated 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."

[8]I.e., the fact that a newly acquired rule is used (see, e.g., LE e.5.3 in §E.1.5.2).

	User 1	User 2	User 3	User 4	User 5
duration (minutes)	7	17	28	14	12
utterances	13	37	37	21	17
learning episodes	3	6	8	2	8
cancelations	0	0	2	1	0
avg. choices per LE	5.67	5.50	7.50	10.00	6.25
rules	7	5	4	4	6
GSG score	1.57	1.33	1.33	2.00	1.67

	User 6	User 7	User 8	User 9	User 10	Total	Average
duration (minutes)	8	20	12	18	9	**145**	**14.50**
utterances	12	27	6	19	14	**203**	**20.30**
learning episodes	5	7	4	9	7	**59**	**5.90**
cancelations	0	1	0	0	0	**4**	**0.40**
avg. choices per LE	5.00	3.00	8.25	6.89	3.57	—	**5.93**
rules	4	5	8	8	5	**56**	**5.60**
GSG score	1.00	1.63	1.13	0.92	1.50	—	**1.37**

Table 7.3: Summary of results for the E-Mail Task. (See §7.4.1 for details.)

As a way of summarizing the user utterances in §D.1, Table 7.5 analyzes the degree of exploration of the domain performed by the users. `<listMail>` and `<readMail>` are the most frequent actions.

As for the actual rules acquired, as mentioned in §7.1, the user sessions resulted in the acquisition of 56 rules (48 of them unique). One evaluation of the goodness of the acquired rules is provided by the average GSG score of 1.37 reported in Table 7.3. An even more compelling evaluation is provided on Tables 7.6 and 7.7.

Table 7.6 shows the semantic accuary of the kernel grammar and the union grammar (i.e., the grammar resulting from adding, to the kernel grammar, the 48 distinct new rules) on the user session corpus. The user session corpus is comprised of the 203 utterances collected from the user sessions.

A grade is assigned by a human expert to each utterance, according to the semantic accuracy of its analysis. A *correct* grade is given if and only if the parse tree is accurate and triggers the intended action. An *incorrect* grade is given if the utterance could be handled by the application and yet the semantic analysis would not trigger the action intended by the user. Finally, an *OOA* grade (for *out-of-application*) is assigned when the end-application would not be able to handle the utterance (e.g., in the E-Mail Task sessions, when the user requested for a spell-check, scroll-down, or undeletion of messages, none of which are supported by the e-mail client).

Score	Explanation
-2	**Terrible**: Acquisition of a wrong rule that prevents construction of correct and achievable parse (and all subsequent usages).
-1	**Bad**: Acquisition of a wrong rule without harmful side effects (and all subsequent usages).
1	**Good**: Acquisition of a good rule but with poor generalization (and all subsequent usages), acquisition of a semantic mapping without rules (and all subsequent usages), no acquisition if utterance is out of domain, or detection of conflict with existing knowledge.
2	**Excellent**: Acquisition of a good rule with good generalization (and all subsequent usages).

Table 7.4: Scores for the numeric evaluation of learning episodes.

As shown in Table 7.6, correct analyses increase from 56.16% (parsing with the kernel grammar) to 88.18% (parsing with the union grammar), achieving a 5-fold reduction in error rate.

To further evaluate the goodness of the acquired rules, in fact, to test their degree of generalization, the same two grammars were used to parse a completely independent test set in the same domain,[9] the results of which are presented in Table 7.7. Even in an independent test set it can be observed that correct analyses increase when using the union grammar, in this case from 44.33% to 74.23%, achieving a a 2-fold reduction in error rate.

After establishing that the acquired rules do indeed correctly extend the initial grammar, we turn our attention to the nature of the acquired rules. The detailed listing of learning episodes in §E.1 contains the acquired subRHS, the initial rule, and the rule after merging. Here, as a way of summary, an analysis is presented on the *lexical* vs. *structural* nature of the rules acquired.

A rule is considered *lexical* if its RHS is composed of terminals only, and *structural* otherwise (i.e., when its RHS contains at least one nonterminal). Table 7.8 shows the counts for lexical and structural acquired rules, broken down by user and grade, Table 7.9 provides

[9] The independent test set for E-Mail domain was obtained by asking the Language Technologies Institute community to donate sentences; the one for the Musicbox task was adapted from a similar task developed at Interactive Systems, Inc.

	User 1	User 2	User 3	User 4	User 5
<countMail>	0	0	0	0	0
<listMail>	5	9	14	2	3
<readMail>	2	15	7	3	3
<composeMail>	1	2	1	0	2
<sendMail>	0	1	1	1	2
<forwardMail>	0	0	0	0	0
<replyMail>	1	7	4	2	1
<deleteMail>	2	2	3	9	3
<sortMail>	2	1	4	3	2
OOA	0	0	3	1	1
Total	**13**	**37**	**37**	**21**	**17**

	User 6	User 7	User 8	User 9	User 10	Total	Percentage
<countMail>	0	0	0	0	0	0	**0.00%**
<listMail>	4	10	1	5	3	56	**27.59%**
<readMail>	2	6	0	1	2	41	**20.20%**
<composeMail>	1	2	2	1	1	13	**6.40%**
<sendMail>	2	3	0	2	0	12	**5.91%**
<forwardMail>	0	1	0	0	0	1	**0.49%**
<replyMail>	1	2	1	2	2	23	**11.33%**
<deleteMail>	1	1	1	3	3	28	**13.79%**
<sortMail>	1	2	1	4	3	23	**11.33%**
OOA	0	0	0	1	0	6	**2.96%**
Total	**12**	**27**	**6**	**19**	**14**	**203**	**100.00%**

Table 7.5: Degree of domain exploration in the E-Mail Task. For example, of the thirteen utterances from User 1 that were directed to the end-application, five where `<listMail>` commands.

the distribution of rule grades given rule types, and Table 7.10 the distribution of rule types given rule grades. It can be observed that lexical rules are more common than structural ones (about a 3:2 ratio), and that structural rules are slightly more difficult to learn correctly (cf. the 80.00% chance a lexical rule has of being graded *excellent* vs. the 60.87% chance for a structural one).

Grade	Kernel Grammar	Union Grammar
Correct	56.16%	88.18%
Incorrect	40.39%	8.37%
OOA	3.45%	3.45%

Table 7.6: Semantic accuracy on the E-Mail Task's user session corpus. Corpus size is 203 utterances.

Grade	Kernel Grammar	Union Grammar
Correct	44.33%	74.23%
Incorrect	55.67%	25.77%

Table 7.7: Semantic accuracy on the E-Mail Task's independent corpus. Corpus size is 97 utterances.

Grade and Type	User 1	User 2	User 3	User 4	User 5
+2 Lexical	3	4	3	1	4
+2 Structural	2	–	1	3	–
+1 Lexical	1	–	–	–	–
+1 Structural	1	–	–	1	2
-1 Lexical	–	1	–	–	–
-1 Structural	–	–	–	–	–
-2 Lexical	–	–	–	–	–
-2 Structural	–	–	–	–	–

Grade and Type	User 6	User 7	User 8	User 9	User 10	Total	Percentage
+2 Lexical	2	1	3	4	3	**28**	**48.28%**
+2 Structural	1	2	3	1	1	**14**	**24.14%**
+1 Lexical	–	1	1	–	1	**4**	**6.90%**
+1 Structural	–	1	1	1	–	**7**	**12.07%**
-1 Lexical	–	–	–	2	–	**3**	**5.17%**
-1 Structural	–	–	–	1	–	**1**	**1.72%**
-2 Lexical	–	–	–	–	–	**0**	**0.00%**
-2 Structural	1	–	–	–	–	**1**	**1.72%**

Table 7.8: Type of rule acquired in the E-Mail Task. The first column indicates the grade (from *excellent* (+2) to *terrible* (-2) (see Table 7.4)) and the type (*lexical* or *structural*) of the rules acquired.

	+2	**+1**	**-1**	**-2**	**Total**
Lexical	80.00%	11.43%	4.35%	0.00%	60.34%
Structural	60.87%	30.43%	4.35%	4.35%	39.66%

Table 7.9: Distribution, in the E-Mail Task, of the grade of acquired rules given their type. For example, given that an acquired rule is lexical, it has an 80.00% chance of being graded +2 (*excellent*). The column of totals indicates overall distribution of types.

	+2	**+1**	**-1**	**-2**
Lexical	66.67%	36.36%	75.00%	0.00%
Structural	33.33%	63.64%	25.00%	100.00%
Total	72.41%	18.97%	6.90%	1.72%

Table 7.10: Distribution, in the E-Mail Task, of the type of acquired rules given their grade. For example, given that the grade of an acquired rule is +2 (*excellent*), it has a 66.67% chance of being lexical. The row of totals indicates overall distribution of grades.

	User 1	**User 2**	**User 3**	**User 4**	**User 5**	**Total**	**Average**
duration (minutes)	11	18	12	15	14	**70**	**14.00**
utterances	18	24	13	22	20	**97**	**19.40**
learning episodes	9	7	8	5	4	**33**	**6.60**
cancelations	0	1	0	0	0	**1**	**0.20**
avg. choices per LE	3.56	3.29	4.25	4.00	7.00	—	**4.15**
rules	4	5	5	2	3	**19**	**3.80**
GSG score	1.88	1.14	1.50	1.40	1.75	—	**1.53**

Table 7.11: Summary of results for the Musicbox Task. (See §7.4.1 for details.)

7.4.2 Musicbox

This section presents an analysis of the results for the Musicbox Task that is parallel to the one just offered in §7.4.1 for the E-Mail Task. Table 7.11 presents a summary of the user sessions for the Musicbox Task. Contrasting Table 7.11 with Table 7.3 one can observe that, whereas session duration and average number of utterances per session are about the same, the average GSG score in the Musicbox is superior to the one in the E-Mail Task (1.53 vs. 1.37), as is the lower average of choices per learning episode (4.15 vs. 5.93).

As for the exploration of the domain, Table 7.12 presents the degree of exploration of the domain performed by the users. In this case, `<addToCart>` and `<listAllSongs>` are the most frequent actions.

As mentioned in §7.1, the user sessions resulted in the acquisition of 19 rules (all of them unique). To evaluate their correctness, the test shown in Table 7.13 was conducted. It shows the semantic accuracy of the kernel grammar and that of the union grammar, on the user session corpus (i.e., the 97 utterances collected from the user sessions, listed in §D.2). Note that correct analyses increase from 62.89% (parsing with the kernel grammar) to 94.85% (parsing with the union grammar), achieving a 16-fold reduction in error rate.

In addition, to evaluate the degree of generalization of the union grammar, the same two grammars were used to parse a completely independent test set in the same domain. Table 7.14 presents the results. Again, correct analyses increase from 21.28% to 41.49%, achieving a 1.35-fold reduction in error rate.

As for the type of rules acquired, Table 7.15 provides the absolute counts for lexical and structural rules, broken down by user and grade, Table 7.16 provides the distribution of rule grades given rule types, and Table 7.17 provides the distribution of rule types given rule grades. It is again found that lexical rules are more common (a 4:1 ratio in this case), and,

	User 1	**User 2**	**User 3**	**User 4**	**User 5**	**Total**	**Percentage**
`<listAllSongs>`	3	3	2	5	3	16	**16.49%**
`<getInformation>`	0	2	0	0	0	2	**2.06%**
`<play>`	3	4	2	2	2	13	**13.40%**
`<pause>`	0	0	0	0	0	0	**0.00%**
`<stop>`	2	2	1	1	1	7	**7.22%**
`<continue>`	0	0	0	0	0	0	**0.00%**
`<fastForward>`	3	0	1	1	1	6	**6.19%**
`<fastBackward>`	1	0	0	0	0	1	**1.03%**
`<rewind>`	0	0	0	0	4	4	**4.12%**
`<louder>`	0	1	1	0	2	4	**4.12%**
`<softer>`	0	1	0	1	2	4	**4.12%**
`<mute>`	0	0	1	0	0	1	**1.03%**
`<addToCart>`	3	3	3	6	3	18	**18.56%**
`<removeFromCart>`	1	1	0	3	0	5	**5.15%**
`<checkout>`	1	1	1	1	1	5	**5.15%**
`<balance>`	1	6	1	2	1	11	**11.34%**
Total	**18**	**24**	**13**	**22**	**20**	**97**	**100.00%**

Table 7.12: Degree of domain exploration in the Musicbox Task. For example, of the eighteen utterances from User 1 that were directed to the end-application, three where `<listAllSongs>` commands.

Grade	**Kernel Grammar**	**Union Grammar**
Correct	62.89%	94.85%
Incorrect	34.02%	2.06%
OOA	3.09%	3.09%

Table 7.13: Semantic accuracy on the Musicbox Task's user session corpus. Corpus size is 97 utterances.

albeit less clearly (possibly due to data sparseness), that structural rules are more difficult to learn correctly.

Grade	**Kernel Grammar**	**Union Grammar**
Correct	21.28%	41.49%
Incorrect	78.72%	58.51%

Table 7.14: Semantic accuracy on the Musicbox Task's independent test corpus. Corpus size is 94 utterances.

	User 1	**User 2**	**User 3**	**User 4**	**User 5**	**Total**	**Percentage**
+2 Lexical	5	1	3	1	1	**11**	**55.00%**
+2 Structural	–	–	1	–	2	**3**	**15.00%**
+1 Lexical	–	2	1	1	–	**4**	**20.00%**
+1 Structural	–	1	–	–	–	**1**	**5.00%**
-1 Lexical	–	1	–	–	–	**1**	**5.00%**
-1 Structural	–	–	–	–	–	**0**	**0.00%**
-2 Lexical	–	–	–	–	–	**0**	**0.00%**
-2 Structural	–	–	–	–	–	**0**	**0.00%**

Table 7.15: Type of rule acquired in the Musicbox Task. The first column indicates the grade (from *excellent* (+2) to *terrible* (-2) (see Table 7.4)) and the type (*lexical* or *structural*) of the rules acquired.

	+2	**+1**	**-1**	**-2**	**Total**
Lexical	68.75%	25.00%	6.25%	0.00%	80.00%
Structural	75.00%	25.00%	0.00%	0.00%	20.00%

Table 7.16: Distribution, in the Musicbox Task, of the grade of acquired rules given their type. For example, given that an acquired rule is lexical, it has a 68.75% chance of being graded + 2 (*excellent*). The column of totals indicates overall distribution of types.

	+2	**+1**	**-1**	**-2**
Lexical	78.57%	80.00%	100.00%	n/a
Structural	21.43%	20.00%	0.00%	n/a
Total	70.00%	25.00%	5.00%	0.00

Table 7.17: Distribution, in the Musicbox Task, of the type of acquired rules given their grade. For example, given that the grade of an acquired rule is + 2 (*excellent*), it has a 78.47% chance of being lexical. The row of totals indicates overall distribution of grades.

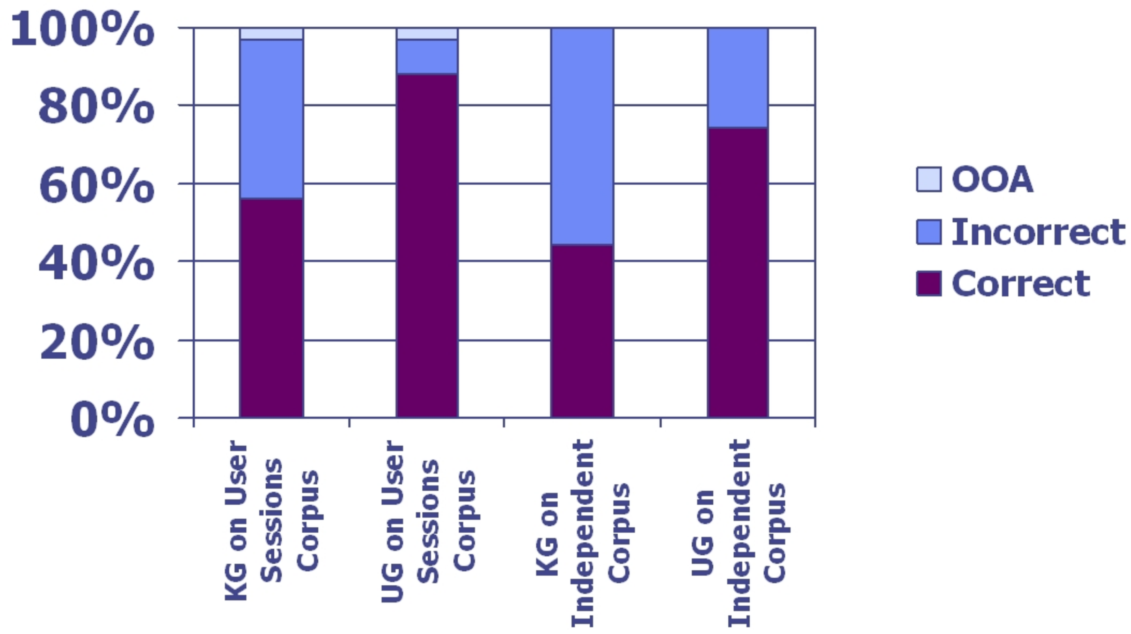


Figure 7.8: Semantic accuracy results for the E-Mail Task. The first column depicts the percentage of correct, incorrect, and out-of-application (OOA) utterances for the kernel grammar (KG) on the corpus comprised of the utterances taken from the user sessions (Session Corpus). The second column shows the performance of the union grammar (UG) on the same Session Corpus. The union grammar is created by adding, to the kernel grammar, all the rules acquired during the user sessions. The third and fourth columns repeat the experiment, this time over an independent corpus of sentences.

7.4.3 Summary

As a final summary of results, Figures 7.8 and 7.9 depict the data contained in Tables 7.6, 7.7, 7.13 and 7.14 in graphical form. They show a substantial improvement in semantic accuracy of the union grammar over the kernel grammar for all tasks and corpora. Finally, 7.18 gives another view on the same data, this time in terms of correctness increment and error decrement factors. Clearly, these results are very satisfactory and validate the thesis of this dissertation.

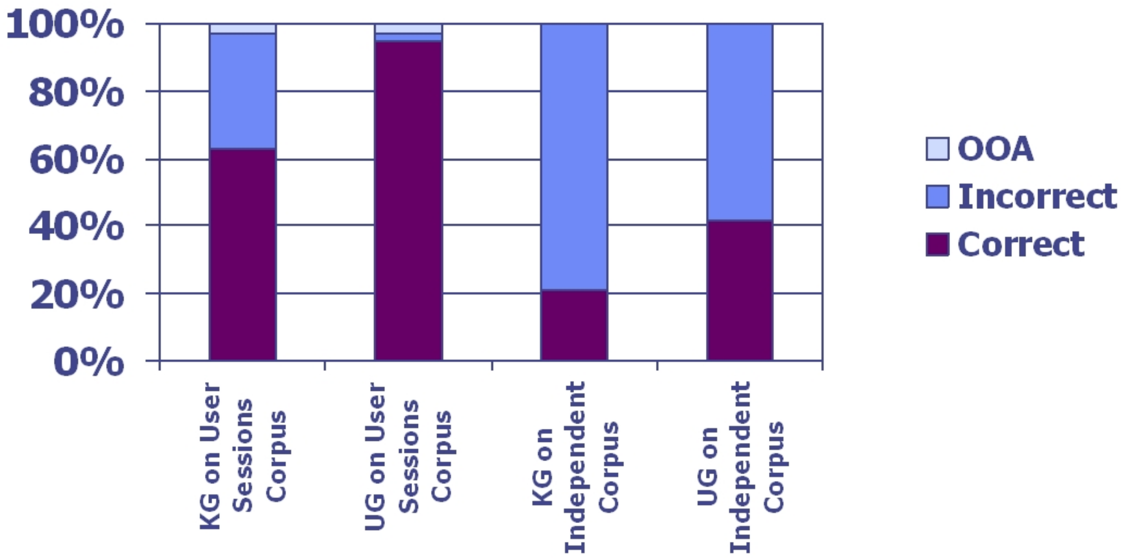


Figure 7.9: Semantic accuracy results for the Musicbox Task. (See Figure 7.8 for an explanation.)

	Corpus Size	**Correctness Increment in Absolute Percentage Points**	**Correctness Increment Factor**	**Error Decrement Factor**
E-Mail Task, user sessions corpus	203	32.09	1.57	4.83
E-Mail Task, independent corpus	97	29.90	1.67	2.16
Musicbox Task, user sessions corpus	97	31.96	1.51	16.51
Musicbox Task, independent corpus	94	20.21	1.95	1.35

Table 7.18: Comparison of semantic accuracy between the union grammar and the kernel grammar for both the E-Mail Task and the Musicbox Task, from the data reported in Tables 7.6, 7.7, 7.13 and 7.14. The first column lists the corpus sizes; the second column lists the difference in absolute percentage points between the semantic accuracy of the union grammar and the semantic accuracy of the kernel grammar; the third column lists the factor that the increment in semantic accuracy reported in the second column represents, in terms of correctness increment; and the fourth column lists the factor that the increment in semantic accuracy reported in the second column represents, this time in terms of error reduction.

7.5 User Comments

A rather unexpected (and very rewarding) aspect of conducting the studies just described was the enjoyment that the users seemed to experience as they intereracted with GSG. A few of their comments:

- "Fascinating to do — would like to see how more sophisticated concepts might be taught"
- "Quite an interesting way of interacting with computer"
- "Really enjoyed being able to 'teach' the computer to pick up synonyms for commands and the concept of the computer 'understanding' "
- "Learned quickly"
- "Fun to interact with — appears 'intelligent' — it's like talking to a kid"

Being compared to a child, with its small "knowledge base" but immense potential and eagerness to learn, is probably the best compliment GSG can receive.

Chapter 8

Conclusion

This chapter concludes the main body of the document with a review of the principal contributions of this dissertation and a sketch of future directions for research.

8.1 Discussion

The results presented in Chapter 7 demonstrate that it is feasible to program a computer to carry on meaningful dialogues with end-users through which linguistic knowledge is acquired, without requiring exorbitant amounts of domain knowledge coded into it. GSG, the system described at length in the preceding chapters, incorporates a mixture of quantitative and qualitative reasoning that allows it to exhibit both *robustness* in the analysis of natural language and *flexibility* in the acquisition of linguistic knowledge. Thus, one can argue that, compared to other systems in the literature, GSG is in the best position to (*i*) give a reasonable answer to the end-user in the face of extragrammaticality, and (*ii*) extend the underlying grammar in a general way.

Several systems or, rather, behaviors, can be taken as a baseline comparison to GSG. The first baseline would be provided by a parser that is not robust, that, for example, does not allow for multiple-tree interpretations or any type of skipping. In that case the answer to any extragrammatical sentence would most likely be an uncoöperative *No parse found — please rephrase.* GSG is obviously superior to that. Another baseline could be provided by a robust parser, such as Lehman's minimal-distance parser CHAMP [Lehman, 1989], Seneff's NT-bigram-based TINA [Seneff, 1992], Lavie's GLR* skipping parser [Lavie, 1996], or Rosé's fragment-combining ROSE [Rosé, 1997]. The behavior of all these systems, with the exception of Lehman's, are robust parsers but not learners, i.e., they are not able to incorporate new knowledge and in fact, given the same extragrammatical utterance, would have to go through the same repair steps. The differences with Lehman's system are discussed in §3.4, but GSG's main improvement is the usage of a variety of learning strategies rather than relying on minimal-distance parsing alone. Finally, when compared to systems whose main advantage is that they are completely trained from data, such as the topic detector presented in [Chu-Carroll and Carpenter, 1999], or the NLU systems based on HMMs or other information-theoretic networks discussed in §3.2 and §3.6, it is clear that GSG is in a different class, as these systems are mainly flat classifiers and cannot produce the nested structures necessary to analyze natural language in non-trivial domains.

8.2 Major Contributions

The main contributions of GSG can be summarized as follows.

- **Rapid deployment of NLU front-ends**. GSG decreases development time: The grammar for a new application can be developed and launched within days (as opposed to months), as only a Kernel Grammar is required.

- **Low initial knowledge, high yield approach to NLU interfaces**. GSG extracts, from a relatively simple context-free grammar, enough information (Ontology, Parsebank, Hypotactic and Paratactic Models) to operate sucessfully.

- **Combination of knowledge sources**. GSG incorporates multiple knowledge sources (semantic grammar, part-of-speech tags and syntactic grammar, end-application constraints) into a unified framework of quantitative (e.g., Prediction Models) and qualitative (e.g., Ontology) reasoning.

- **Combination of learning strategies**. GSG utilizes the knowledge sources in a variety of learning strategies (All-top parsing, Anchor Mother Prediction, Required/Is-a/... Daughter Search, Verbal Head Search, Parser Predictions, Vertical Generalization, Horizontal Generalization, Rule Merging).

- **Natural-language dialogue**. GSG is able to combine these different knowledge sources and learning strategies into a coherent, mixed-initiative conversation with the end-user.

- **Sophisticated rule management**. GSG vertically and horizontally generalizes acquired rules and tests them for subsumption and ambiguity introduction. Moreover, the Rule Merging algorithm constructs rules that are more general than a simple addition as an alternative.

- **Usage of a standard grammar formalism**. GSG uses grammars written in the standard JSGF formalism for both its input (Kernel Grammar) and its output (extended grammar).

- **Acquired rules improve coverage on independent corpus**. Empirical results show that the acquired rules not only resolve the particular extragrammatical sentence that gave rise to them, but also that they are general enough to apply to other sentences.

- **Results hold across domains and developers**. These positive results hold across different application domains and developers.

- **User satisfaction increases**. Application end-users enjoy the ability to teach the system their language patterns.

8.3 Future Directions

When building a system of GSG's magnitude there are many places where one has to choose between what is possible (and even desirable) and what is feasible to accomplish within a reasonable amount of time. What follows are some of the directions in which GSG could improve.

8.3.1 Integration with a Speech Recognizer

An objective of GSG has always been to provide a *speech* interface to the end application. The design of GSG (from using a fast and robust parser to limiting multiple-choice questions to three items at a time) has been done in view of an eventual integration with a speech recognizer and a speech synthesizer. When GSG becomes fully integrated with a speech recognizer, it remains to be seen how an optimal point in the tradeoff between the wide coverage but relatively low word recognition accuracy obtained with a loose dictation grammar, and the narrow coverage but high word accuracy achieved with a tight task grammar, can be found, and how the degradation of the input is going to affect GSG's behavior.

8.3.2 End-application as Knowledge Source

It is clear that the more linguistic and domain knowledge GSG has, the better its hypotheses will be. The fact that GSG is able to work with a grammar as its sole source of domain knowledge is definitely an advantage, but, at the same time, it also makes sense to incorporate other knowledge sources, as demonstrated in the Musicbox Task where the end-application provided further filtering of anchor mothers depending on its dialogue state (see §7.2.2).

This filtering can be seen as a first step toward a more sophisticated collaboration between GSG and the end application: A stochastic model could be built so that NT probabilities would depend on the state of the end application and possibly other parameters. Then, for example, the likelihood of `<shoppingcart.checkout>` in the Musicbox Task could be made to automatically increase over time (within a session) as well as, say, with the number of songs in the cart (and maybe the purchasing history of the user could even be taken into account).

From GSG's perspective, all the mechanisms necessary to support such a model are in place: Even if direct participation in the prediction of the anchor mother is not feasible, reranking of the predicted anchor mothers is definitely allowed.

8.3.3 Context-dependent Learning

A related question is allowing for context-dependent learning. For example, in one occasion in the E-Mail Task a user wanted to teach GSG that *ok, do it* means `<sendMail>`, but only after a `<forwardMail>` command. The present system, however, is unable to model such inter-utterance contextual constraints. An idea would be, therefore, to extend GSG beyond a pure context-free grammar so that the particular subRHS `ok do it` under `<sendMail>` would only be active if the previous action had been a `<forwardMail>`. At the same time, this limitation on the applicability of acquired rules would, of course, negatively impact generalization performance.

8.3.4 Acquisition of New Concepts

A very interesting area of research would be the acquisition of new concepts within GSG. In fact, from the implementation's point of view, all the mechanisms are in place to allow for the creation of new NTs. The problem is rather the difficulty of handling new concepts once they are created. For example, it would be possible, at the end of Interaction 7.3.2.2 (*undelete message from lucy*, see §7.3.2.2), to create a new NT `<undelete>` and rule `<undelete> = undelete <_MAIL_ARGUMENT>+`. But the end application would not be able to process this new command. In fact, if the concept is not in the grammar, it most likely means that it is not supported by the end application. Thus, not until a new class of flexible end applications exists, does it make sense to attempt the acquisition of concepts. Otherwise, the system would only mislead the user by purporting to understand a concept that cannot be executed.

8.3.5 Usage of Semantic Distance

GSG is flexible in its naming of NTs (see §6.3.6.4) but such flexibility could be taken a step further by using existing semantic knowledge sources such as WordNet [Fellbaum (ed.), 1998] to compute a semantic distance between words and NTs. For example, when a user in the E-Mail Task said *read email about travel*, she was referring to a message with the subject *you're going on a cruise.* In a semantic net, *cruise* would have a strong association with *travel*, and so it could be selected as the most likely message the user has in mind.

8.3.6 Anaphora Resolution

An important extension of GSG would be an Anaphora Resolution module, i.e., an algorithm that given, say, *show last message from cynthia* followed by *reply to her* would identify *her* as referring to *cynthia.* The lack of such a module is perhaps the largest impediment to mantaining a truly natural conversation in the current system.

8.3.7 Rule Compaction after Merging

More for aesthetic reasons that for any other consideration, GSG could benefit from an algorithm that would compact the RHS resulting from a rule merger (see §6.3.5.4, and all learning episodes in Appendix E whose score ends with a minus sign) to its most succinct representation. For example, `a b* b*` would become `a b*`. At the same time, such an algorithm would be expensive (NP-hard, see [Kam et al., 1994]), and having a compacted RHS would make it less straightforward to undo a merger.

8.3.8 Library of Grammars

With the establishment of a language technologies industry, it has become an economic necessity to reuse grammar components (as exemplified by Nuance's SpeechObjects [Nuance, 2000] or Tellme's Intrinsic Grammars [Tellme, 2000]). GSG supports the full JSAPI import mechanisms, which allow for modularized grammars and dynamic changes. Yet, it would be useful to, on the one hand, develop a set of foundational grammar modules (for, say, dates and times), and, on the other, design a meta-JSGF language to parameterize such modules, so that, say `<PizzaPrice[Price:USD:0-50]>` would mean that NT `<PizzaPrice>` is a `<Price>` with values ranging from free to US$50.

8.4 Conclusion

As one user remarked ("At times I felt the computer was teaching *me* to use its vocabulary, altough eventually I taught it"), there will always be users that adapt their language to what they know the system understands, rather than the other way around. In a sense, this is a normal human response: as we engage in a dialogue, we tend to adopt our interlocutor's words and mannerisms. At the same time, given the open-endedness of natural language, it is clear that adaptive systems are extremely useful: A crucial difference between a speech interface and, say, a graphical user interface, is that in the latter case the user can easily learn the functionality of the application by clicking on menues and submenues. In the case of speech, however, the functionality will become apparent through the dialogue, so it is of paramount importance, if the interaction is to be a real conversation, that the NLU system be able to learn.

GSG, by virtue of its built-in robustness, minimal initial knowledge requirements, and learning abilities, begins to embody the kind of qualities that NLU systems should possess, if they are to provide, without exorbitant development effort, a conversational interface to a multiplicity of applications that feels truly natural to humans.

Appendices

Appendix A

Small Grammar to Illustrate SOUP's Heuristics

A.1 G3

```
#JSGF V1.0 ISO8859-1 en;

/**
 * ============================================================================
 * @author           Marsal Gavalda
 * ============================================================================
 */

grammar g3;

// -------------------------------------
// 3^0 = 1
// -------------------------------------
public <t> = ( <1>   | <2>   | <3>   )+ ;

// -------------------------------------
// 3^1 = 3
// -------------------------------------
<1>        = ( <11>  | <12>  | <13>  )+ ;
<2>        = ( <21>  | <22>  | <23>  )+ ;
<3>        = ( <31>  | <32>  | <33>  )+ ;

// -------------------------------------
// 3^2 = 9
// -------------------------------------
<11>       = ( <111> | <112> | <113> )+ ;
<12>       = ( <121> | <122> | <123> )+ ;
<13>       = ( <131> | <132> | <133> )+ ;
// -------------------------------------
<21>       = ( <211> | <212> | <213> )+ ;
<22>       = ( <221> | <222> | <223> )+ ;
<23>       = ( <231> | <232> | <233> )+ ;
// -------------------------------------
<31>       = ( <311> | <312> | <313> )+ ;
<32>       = ( <321> | <322> | <323> )+ ;
<33>       = ( <331> | <332> | <333> )+ ;

// -------------------------------------
// 3^3 = 27
// -------------------------------------
<111>      =  a  |  111 ;
<112>      =  b  |  112 ;
<113>      =  c  |  113 ;
// -------------------------------------
<121>      =  d  |  121 ;
<122>      =  e  |  122 ;
<123>      =  f  |  123 ;
// -------------------------------------
<131>      =  g  |  131 ;
<132>      =  h  |  132 ;
<133>      =  i  |  133 ;
```

```
// -------------------------------------
// -------------------------------------
<211>      =  j  |  211 ;
<212>      =  k  |  212 ;
<213>      =  l  |  213 ;
// -------------------------------------
<221>      =  m  |  221 ;
<222>      =  n  |  222 ;
<223>      =  o  |  223 ;
// -------------------------------------
<231>      =  p  |  231 ;
<232>      =  q  |  232 ;
<233>      =  r  |  233 ;
// -------------------------------------
// -------------------------------------
<311>      =  s  |  311 ;
<312>      =  t  |  312 ;
<313>      =  u  |  313 ;
// -------------------------------------
<321>      =  v  |  321 ;
<322>      =  w  |  322 ;
<323>      =  x  |  323 ;
// -------------------------------------
<331>      =  y  |  331 ;
<332>      =  z  |  332 ;
<333>      =  -  |  333 ;
// =EOF==================================================================
```

Appendix B

Syntactic Grammar

B.1 Part Of Speech Tag Set

`ANA`	Anaphoric element, e.g. *I can understand* <u>*that*/`ANA`</u>.
`PRPA`	Personal pronoun in accusative case.
`NEG`	Negation particle.
`AFF`	Affirmative particle.
`CCC`	Constituent conjunction (cf. `CC`).
`AUX-N`	Negated `AUX`, e.g. *isn't*.
`CV`	Conversational words, e.g. *hi*, *bye-bye*.
`AUX`	Auxiliary verb, e.g. *might*, *can*.
`CC`	Clause conjunction (cf. `CCC`).
`CD`	Cardinal number.
`DT`	Determiner.
`PDT`	Plural determiner, e.g. *all*.
`DT-AUX`	Determiner-auxiliary, e.g. *that's*
`EX`	Expletive, e.g. *there*, *here*.
`EX-AUX`	Expletive-auxiliary, e.g. *there's*, *here's*.
`FW`	Foreign word.
`JJ`	Adjective.
`JJR`	Comparative adjective.
`JJS`	Superlative adjective.
`NN`	Noun.
`NNS`	Plural noun.
`NNP`	Proper noun.
`NNPS`	Plural proper noun.
`PREP`	Preposition.
`PRP`	Personal pronoun.
`PRP$`	Possessive personal pronoun.
`RB`	Adverb (modifier), e.g. *always*, *just*, *kind_of*.
`RBR`	Adverb (complementizer).
`RBS`	Adverb (superlative).
`RP`	Verbal particle, e.g. *set ...* <u>*up*/`RP`</u>, *walked ...* <u>*out*/`RP`</u>.
`TO`	*to* + infinitive.
`UH`	Filled pause, e.g. *uh-huh*.
`VB`	Verb (infinitive).
`VBD`	Verb (past).
`VBG`	Verb (gerund).
`VBN`	Verb (past participle).
`VBP`	Verb (present).
`VBZ`	Verb (inflected 3rd-person, singular, present).
`WDT`	Relative pronoun.
`WP`	Adjectival *wh*-particle, e.g. *what company is ...*
`WRB`	Oblique argument wh-particle, e.g. *where*, *how*, *when*.

B.2 Top-level Categories of Syntactic Grammar

NP	Noun phrase.
ADV	Adverbial phrase.
PP	Prepositional phrase.
VB	Verb.
VBNEG	Negated verb.
VPART	Verbal particle.
BABB	Babble.
EXPL	Expletive.
CONJ	Conjunction.
WH	*wh*-word.
WHNP	*wh*-word + noun phrase.
PRDADJ	Predicate adjective.
TOINF	*to* + infinitive.
NEG	Negative particle.
AFF	Affirmative particle.
AUX	Auxiliary.
AUXNEG	Negated auxiliary.

B.3 Syntactic Grammar

```
#JSGF V1.0 ISO8859-1 en;

/**
 * ===========================================================================
 * @taskName        "Syntactic chunk grammar for English"
 * ---------------------------------------------------------------------------
 * @taskDescription "syntactic shallow parsing"
 * ===========================================================================
 * @stringMap       "'"            --> "+"
 * @stringMap       "+d like "     --> " would like "
 * @stringMap       "+s had "      --> " has had "
 * @stringMap       "+s been "     --> " has been "
 * @stringMap       " you+re "     --> " you are "
 * @stringMap       " what+s "     --> " what is "
 * @stringMap       " there+s "    --> " there is "
 * @stringMap       " that+s "     --> " that is "
 * @stringMap       " it+s "       --> " it is "
 * @stringMap       " i+m "        --> " i am "
 * @stringMap       "+s "          --> " +s "
 * @stringMap       " can+t "      --> " can not "
 * @stringMap       "n+t "         --> " not "
 * @stringMap       " per cent "   --> " percent "
 * ===========================================================================
 * @author           Klaus Zechner
 * @author           Marsal Gavalda
 * ===========================================================================
 */

grammar SynEng;

// ---------------------------------------------------------------------------
// top principal NTs
// ---------------------------------------------------------------------------
public <np> =
  ( [ pdt] [ dt | "prp$"] [ <JJ_MOD>] <n_head>+)
| ( <pn_head>)
| ( [ pdt] ana)
;

public <pp> =
  ( <p_head> [ pdt] [ dt | "prp$"] [ <JJ_MOD>] <n_head>+)
| ( <p_head> ( <pn_head> | ( [pdt] ana)))
```

```
;

public <vb> =
  ( [ aux] [ <adv>] <VB>* <vb_head>)
        ;

public <vbneg> =
  ( ( ( [ aux | <VB>] neg) | aux-n) [ <adv>] <VB>* <vb_head>)
;

public <babb> =
  ( xx)
| ( uh)
;

public <adv> =
  ( <RB>+)
| ( prep rb)
;

public <expl> =
  (ex [ <VB> | aux])
| (ex-aux)
;

public <conj> =
  ( ccc)
| ( cc)
| ( wdt)
;

public <wh> =
  ( wp)
| ( wrb)
;

public <prdadj> =
  ( <JJ_MOD>)
;

public <toinf> =
          ( to <vb_head>)
;

public <neg> =
  ( neg)
;

public <aff> =
          ( aff)
;
public <vpart> =
          ( rp)
;

public <whnp> =
  (wp [ <JJ_MOD>] <np>)
;

public <aux> =
( aux+)
;

public <auxneg> =
( aux-n)
;

// ---------------------------------------------------------------------------
// non-top principal NTs
// ---------------------------------------------------------------------------
<n_head> =
( <NN>)
```

```
;

<vb_head> =
( <VB>)
;

<p_head> =
( prep)
;

<pn_head> =
  ( prp)
| ( prpa)
;

// ---------------------------------------------------------------------------
// auxiliary NTs
// ---------------------------------------------------------------------------
<NN> =
  ( nn)
| ( nns)
| ( nnp)
| ( nnps)
;
<VB> =
  ( vbz)
| ( vbp)
| ( vbd)
| ( vbg)
| ( vbn)
| ( vb)
;
<RB> =
  ( rb)
| ( rbr)
| ( rbs)
;
<JJ_MOD> =
  ( [ cd] <adv>* <JJ_ONLY>+)
;
<JJ> =
  ( <JJ_ONLY>)
| ( "prp$")
| ( rb)
| ( rbr)
| ( rbs)
;

<JJ_ONLY> =
  ( jj)
| ( jjs)
| ( jjr)
| ( cd)
;
// =EOF======================================================================
```

Appendix C

Semantic Grammars

C.1 GSG's Metagrammar

```
#JSGF V1.0 ISO8859-1 en;

/**
 * ==========================================================================
 * @taskName        "Growing Semantic Grammars's Metagrammar"
 * --------------------------------------------------------------------------
 * @taskDescription "where you can teach the computer your language!"
 * --------------------------------------------------------------------------
 * @exampleSentence "help"
 * ==========================================================================
 * @stringMap       "'"             --> "+"
 * @stringMap       "+d like "      --> " would like "
 * @stringMap       "+s had "       --> " has had "
 * @stringMap       "+s been "      --> " has been "
 * @stringMap       " you+re "      --> " you are "
 * @stringMap       " what+s "      --> " what is "
 * @stringMap       " there+s "     --> " there is "
 * @stringMap       " that+s "      --> " that is "
 * @stringMap       " it+s "        --> " it is "
 * @stringMap       " i+m "         --> " i am "
 * @stringMap       "+s "           --> " +s "
 * @stringMap       " can+t "       --> " can not "
 * @stringMap       "n+t "          --> " not "
 * @stringMap       " per cent "    --> " percent "
 * ==========================================================================
 * @author           Marsal Gavalda
 * ==========================================================================
 */
grammar gsg;

// ==========================================================================
// Universal Module
// ==========================================================================

public <top> =
  ( <means>)
| ( <isa>)
| ( <ignore>)

| ( <help>)

| ( <yes>)
| ( <no>)
| ( <cancel>)

| ( <greeting>)
| ( <farewell>)

| ( <thank>)
| ( <thankBack>)

| ( <summarize>)
;

// --------------------------------------------------------------------------
// Special GSG operators
```

```
// ------------------------------------------------------------------------------
<means> =
  ( means)
| ( means the same as)
| ( means like)
| ( is the same as)
| ( is like)
| ( is shorthand for)
| ( is short for)
| ( stands for)
| ( has the same meaning as)
| ( has the same meaning like)
| ( refers to)
| ( i mean)
| ( i am referring to)
;
<isa> =
  ( is a)
| ( is an)
| ( isa)
| ( is-a)
| ( is an example of)
| ( is a kind of)
| ( is a form of)
| ( is a way of)
| ( is a way to express)
| ( is a way of expressing)
;
<ignore> =
  ( "<s>" (ignore | skip | forget [ about] | never mind) [ it | this | that] "</s>")
| ( "<s>" (it | this | that) is not ( relevant | important) "</s>")
;
// ------------------------------------------------------------------------------
// Top-level
// ------------------------------------------------------------------------------
<help> =
  ( "<s>" help "</s>")
| ( "<s>" help me [ out] [ here] "</s>")
| ( i <_NEED> [ some] help  [ here])
| ( i am not <_SURE> i [ really] <_UNDERSTAND>)
| ( what do you ( understand | know [ about]))
| ( what can i ( say | do))
;
<yes> =
  ( "<s>" yes "</s>")
| ( "<s>" affirmative "</s>")
| ( "<s>" positive "</s>")
| ( "<s>" ok "</s>")
| ( "<s>" o k "</s>")
| ( "<s>" sounds <_GOOD> "</s>")
| ( "<s>" [ that is] <_RIGHT> "</s>")
;
<no> =
  ( "<s>" no "</s>")
| ( "<s>" negative "</s>")
| ( "<s>" none of the above)
| ( "<s>" all [ <_CHOICES>] <_BAD>)
| ( "<s>" [ ( this | that) is] <_BAD>)
| ( "<s>" i do not <_THINK> so)
;
<cancel> =
  ( "<s>" ( cancel | abort) "</s>")
| ( (that | this) is not what i
            ( mean | meant | ( ( wanted | intended) [ to say]))
          )
;
<greeting> =
  ( hello)
| ( how are you [ today])
```

```
| ( how is it going)
| ( how are you doing)
| ( hi)
| ( hullo)
| ( what is up)
| ( whassup)
;
<farewell> =
  ( [ good] bye+ [ now])
| ( [ may] god be with ( ye | you))
| ( farewell)
| ( fare [ you] well)
| ( so long)
| ( see (you | ya) [ later])
| ( [ i will be] seeing you)
| ( adieu)
| ( "<s>" quit)
| ( "<s>" exit)
| ( "<s>" die)
;
<thank> =
  ( thank you)
| ( thanks)
| ( thank you [ so] very* much)
| ( many thanks)
;
<thankBack> =
  ( you are [ always] welcome)
| ( you are more than welcome)
| ( my pleasure)
| ( it was a pleasure)
| ( no problem)
;
<summarize> =
  ( "<s>" summarize)
| ( "<s>" summary)
| ( show summary)
| ( make summary)
| ( create summary)
| ( display summary)
| ( "<s>" [ so] [ in a] nutshell)
| ( "<s>" sum up)
;
// ------------------------------------------------------------------------------
// Auxiliary
// ------------------------------------------------------------------------------
<_RIGHT> =
  ( [ all] right)
| ( alright)
| ( correct)
;
<_GOOD> =
  ( good)
| ( great)
| ( perfect)
| ( wonderful)
;
<_BAD> =
  ( bad)
| ( wrong)
| ( incorrect)
| ( not <_RIGHT>)
| ( [ way] off)
| ( terrible)
| ( awful)
| ( nonsense)
| ( nonsensical)
| ( rubbish)
```

```
| ( bullshit)
;
<_THINK> =
  ( think)
| ( believe)
;
<_NEED> =
( [ can | could | would] [ <_DEFINETELY>] ( need | use))
;
<_DEFINETELY> =
  ( definetely)
| ( surely)
| ( positively)
;
<_SURE> =
( [ <_ENTIRELY>] ( sure | positive))
;
<_ENTIRELY> =
  ( entirely)
| ( hundred percent)
| ( 100 percent)
| ( at all)
| ( really)
;
<_UNDERSTAND> =
  ( understand)
| ( follow)
| ( get it)
;
<_CHOICES> =
  ( choices)
| ( suggestions)
| ( guesses)
| ( estimates)
| ( guestimates)
;
// =EOF=====================================================================
```

C.2 E-Mail Task Grammar

```
#JSGF V1.0 ISO8859-1 en;

/**
 * ==========================================================================
 * @taskName        "E-Mail Manager"
 * --------------------------------------------------------------------------
 * @taskDescription "where you can manage your e-mail through spoken commands!"
 * --------------------------------------------------------------------------
 * @exampleSentence "do i have any mail"
 * @exampleSentence "please show me all messages about the meeting"
 * @exampleSentence "reply to the last message from peter"
 * @exampleSentence "sort emails by sender"
 * @exampleSentence "retrieve all messages from bob that were sent after christmas"
 * @exampleSentence "by christmas i'm referring to december twenty fifth"
 * @exampleSentence "please retrieve emails to mary before christmas"
 * ==========================================================================
 * @stringMap       "'"           --> "+"
 * @stringMap       "+d like "    --> " would like "
 * @stringMap       "+s had "     --> " has had "
 * @stringMap       "+s been "    --> " has been "
 * @stringMap       " you+re "    --> " you are "
 * @stringMap       " what+s "    --> " what is "
 * @stringMap       " there+s "   --> " there is "
 * @stringMap       " that+s "    --> " that is "
 * @stringMap       " it+s "      --> " it is "
 * @stringMap       " i+m "       --> " i am "
 * @stringMap       "+s "         --> " +s "
 * @stringMap       " can+t "     --> " can not "
 * @stringMap       "n+t "        --> " not "
 * ==========================================================================
 * @author           Marsal Gavalda
 * ==========================================================================
 */

grammar voiceEMail;

// ==========================================================================
// The Wilcard rule
// ==========================================================================
<_WILDCARD> =
  ( "_$any$_")
;

// ==========================================================================
// Top-level
// ==========================================================================

public <countMail> =
  ( [ <_VERB_DESIRE>] <_COUNT> [ <_TO_FOR_ME>] <_MAIL_ARGUMENT>*)
| ( [ <_VERB_DESIRE>] [ <_KNOW>] <_HOW_MANY> <_MAIL_ARGUMENT>*)
;

public <listMail> =
  ( [ <_VERB_DESIRE>] (<_LIST>  [ <_TO_FOR_ME>] | <_ASK_TO_CHECK>) <_MAIL_ARGUMENT>*)
;

public <readMail> =
  ( [ <_VERB_DESIRE>] <_READ> <_MAIL_ARGUMENT>*)
;

public <composeMail> =
  ( [ <_VERB_DESIRE>] <_COMPOSE> <_MAIL_ARGUMENT>*)
;

public <sendMail> =
  ( [ <_VERB_DESIRE>] <_SEND> <_MAIL_ARGUMENT>*)
;

public <forwardMail> =
  ( [ <_VERB_DESIRE>] <_FORWARD> <_MAIL_ARGUMENT>*)
;

public <replyMail> =
  (  [ <_VERB_DESIRE>] <_REPLY> [ <_TO_FOR_ME>] <_MAIL_ARGUMENT>*)
;

public <deleteMail> =
  ( [ <_VERB_DESIRE>] <_DELETE> <_MAIL_ARGUMENT>*)
;

public <sortMail> =
```

```
  ( [ <_VERB_DESIRE>] <_SORT> <_MAIL_ARGUMENT>*
    ( [ <_SORT_MODE>] [ <_SORT_BY>] | ( <_SORT_BY> <_SORT_MODE>))
  )
;

// ---------------------------------------------------------------------------
// Auxiliary
// ---------------------------------------------------------------------------
<_VERB_DESIRE> =
  ( <_I> <_WANT> to)
| ( please)
;

<_MAIL_ARGUMENT> =
  ( [ <_ARTICLE>] <_MAIL> [ <_I_HAVE> | <_TO_FOR_ME>])
| ( <_RECIPIENT>)
| ( <mailUnread>)
| ( <_SENDER>)
| ( <_SUBJECT>)
| ( <_DATE>)
| ( <_MESSAGE_IDX>)
| ( <_RELATIVE> [ <_MAIL_VERB_PAST>])
| ( <_MAIL_VERB_PAST>)
;

<_ARTICLE> =
  ( a)
| ( an)
| ( [ yet] another)
| ( the)
;

<_MAIL_VERB_PAST> =
  ( sent)
| ( received)
;

<_HOW_MANY> =
( how many)
;

<_KNOW> =
  ( know)
| ( find out)
;

<_COUNT> =
  ( count)
| ( add up)
;

<_ASK_TO_CHECK> =
  ( do i have)
| ( is there [ any | anything])
| ( there is [ any | anything])
;

<_LIST> =
  ( list)
| ( get)
| ( search [ for])
;

<_TO_FOR_ME> =
( [ for | to] me)
;

<_I_HAVE> =
  (i ( have | got))
| (i [ have] received)
;

<_READ> =
  ( read)
| ( print)
| ( tell [me about])
;

<_DELETE> =
  ( delete)
;

<_SORT> =
```

```
  ( sort)
;
<_SORT_BY> =
  ( <_BY> [ the] <_SORT_BY_> [ of [ the] <_MAIL>]) // _BY is required
;
<_SORT_BY_> =
( <sortBy__date>
| <sortBy__subject>
| <sortBy__sender>
| <sortBy__recipient>
| <sortBy__size>
)
;
<_BY> =
  ( by)
| ( according [ to])
| ( following)
;
<sortBy__date> =
  ( date)
| ( time)
| ( the order in which <_MAIL> was recieved)
;
<sortBy__subject> =
  ( subject)
| ( thread)
;
<sortBy__sender> =
  ( sender)
| ( origin)
| ( source)
;
<sortBy__recipient> =
  ( recipient)
| ( recipients)
| ( addressee)
| ( addressees)
;
<sortBy__size> =
  ( size)
;
<_SORT_MODE> =
  ( [ in] ( <sortMode__ascending> | <sortMode__descending>) [ mode])
;
<sortMode__ascending> =
  ( ascending)
| ( increasing)
;
<sortMode__descending> =
  ( descending)
| ( decreasing)
;
<mailUnread> =
  ( new)
| ( unread)
;
<_SUBJECT> =
  ( <_SUBJECT_PRE> [ <_ARTICLE>] <subject__STRING>)
;
<_SUBJECT_PRE> =
  ( on)
| ( about)
| ( regarding)
| ( dealing with)
| ( <_THAT_WHICH> ( deals | deal) with)
| ( with regards to)
| ( on the <_SUBJECT_WORD> of)
```

```
| ( [ with] subject [keyword | keywords])
;
<subject__STRING> =
  ( "<subject>")
| ( <_WILDCARD>+)
;
<_SUBJECT_WORD> =
  ( subject)
| ( matter)
| ( issue)
| ( problem)
;
<_DATE> =
  ( <dateRange>)
| ( [ <_DATE_POINT_PRE>] <datePoint>)
;

<dateRange> =
  ( <_DATE_START_PRE> <dateStart>)
| ( <dateStart> <_DATE_START_POST>)

| ( <_DATE_END_PRE> <dateEnd>)
| ( <dateEnd> <_DATE_END_POST>)

| ( <_DATE_RANGE_PRE> <dateStart> <_DATE_RANGE_IN> <dateEnd>)
;
<dateStart> =
  ( <datePoint>)
;
<dateEnd> =
  ( <datePoint>)
;
<datePoint> =
  ( <_DATE_POINT_ARGUMENT>+)
;
<_DATE_RANGE_PRE> =
  ( between)
| ( within)
| ( from)
| ( <_DATE_START_PRE>)
;
<_DATE_RANGE_IN> =
  ( and)
| ( to)
| ( <_DATE_END_PRE>)
;
<_DATE_START_PRE> =
  ( after)
| ( past)
| ( since)
| ( (starting | beginning) [ at | from])
| ( no earlier than)
| ( not ( before | prior to | preceding))
;
<_DATE_START_POST> =
  ( or ( later | after that | thereafter))
| ( old)
| ( ago)
;
<_DATE_END_PRE> =
  ( before)
| ( until)
| ( till)
| ( prior to)
| ( older than)
| ( no later than)
| ( preceding)
;
<_DATE_END_POST> =
  ( or ( earlier | before that))
;
```

```
<_DATE_POINT_PRE> =
  ( on)
| ( at)
// e.g. "list messages from march" means march only,
// cf.  "list messages from march to april" --> <dateRange>
| ( from)
;

<_DATE_POINT_ARGUMENT> =
  ( <_DATE_RELATIVE>)
| ( <_DATE_FIXED>)
| ( <_DAY_OF_WEEK>)
| ( <_MONTH> <_DAY_OF_MONTH>)                    // july first
| ( [ <_DAY_OF_MONTH>] <_MONTH>)                 // first of july
| ( [ on] [ the] <_DAY_OF_MONTH>)                // on the first
| ( <_HOUR>)
| ( <_TIME_OF_DAY>)
| ( <_YEAR>)
| ( <_DATE_POINT_MODIFIER>)
;
<_DATE_FIXED> =
  ( <dateFixed__jan01>)
| ( <dateFixed__jul04>)
;
<dateFixed__jan01> =
  ( new year)
| ( new years day)
| ( new year "+s" day)
;
<dateFixed__jul04> =
  ( independence day)
| ( fourth of july)
;
<_DATE_RELATIVE> =
  ( <dateRelative__today>)
| ( <dateRelative__tomorrow>)
| ( <dateRelative__yesterday>)
| ( <dateRelative__week>)
| ( <dateRelative__month>)
| ( <dateRelative__year>)
| ( <dateRelative__century>)
| ( <dateRelative__millennium>)
;
<dateRelative__today> =
( today)
;
<dateRelative__tomorrow> =
( tomorrow)
;
<dateRelative__yesterday> =
( yesterday)
;
<dateRelative__week> =
  ( week)
| ( weeks)
;
<dateRelative__month> =
  ( month)
| ( months)
;
<dateRelative__year> =
  ( year)
| ( years)
;
<dateRelative__century> =
  ( century)
| ( centuries)
;
```

```
<dateRelative__millennium> =
  ( millennium)
| ( millennia)
;
<_DATE_POINT_MODIFIER> =
  ( <datePointModifier__this>)
| ( <datePointModifier__next>)
| ( <datePointModifier__previous>)
;
<datePointModifier__this> =
( this)
;
<datePointModifier__next> =
( next)
;
<datePointModifier__previous> =
  ( previous)
| ( last)
;
<_DAY_OF_WEEK> =
  ( <dayOfWeek__0>)
| ( <dayOfWeek__1>)
| ( <dayOfWeek__2>)
| ( <dayOfWeek__3>)
| ( <dayOfWeek__4>)
| ( <dayOfWeek__5>)
| ( <dayOfWeek__6>)
;
<_MONTH> =
  ( [ <_MONTH_PRE>] <_MONTH_VAL>)
;
<_MONTH_PRE> =
( on)
;
<_MONTH_VAL> =
  ( <month__1>)
| ( <month__2>)
| ( <month__3>)
| ( <month__4>)
| ( <month__5>)
| ( <month__6>)
| ( <month__7>)
| ( <month__8>)
| ( <month__9>)
| ( <month__10>)
| ( <month__11>)
| ( <month__12>)
;
<_DAY_OF_MONTH> =
  ( [ [ on] the] <dayOfMonth__INTEGER> [ of])
;
<dayOfMonth__INTEGER> =
  ( <_CARDINAL_NUMBER_1_31>)
| ( <_ORDINAL_NUMBER_0_99>)
;
<_HOUR> =
( [ at] <hour__INTEGER> <_HOUR_POST>*)
;
<hour__INTEGER> =
  ( <_CARDINAL_NUMBER_0_24>)
;
<_HOUR_POST> =
  ( hour)
| ( hours)
| ( [ and] <_MINUTE>)
;
<_MINUTE> =
  ( <minute__INTEGER> [ minute | minutes])
;
```

```
<minute__INTEGER> =
  ( <_CARDINAL_NUMBER_0_60>)
;
<_YEAR> =
  ( [ on] <year__INTEGER> [ year | anno domini])
;
<year__INTEGER> =
  ( <_CARDINAL_NUMBER_0_99999>)
;
<_TIME_OF_DAY> =
  ( [ at | in the] <_TIME_OF_DAY_VAL>)
;
<_TIME_OF_DAY_VAL> =
  ( <timeOfDay__am>)
| ( <timeOfDay__pm>)
| ( <timeOfDay__STRING>)
;
<timeOfDay__am> =
  ( a m)
| ( ante meridiem)
| ( morning)
;
<timeOfDay__pm> =
  ( p m)
| ( post meridiem)
| ( afternoon)
;
<timeOfDay__STRING> =
  ( noon)
| ( midnight)
| ( evening)
| ( night)
;
<_MESSAGE_IDX> =
  ( <messageIndex__all>)
| ( <messageIndex__1>)
| ( [ number] <messageIndex__INTEGER>)
| ( [ the] <messageIndex__INTEGER_ORDINAL> <_MAIL>)
// e.g. "the last two"
| ( [ the] <messageIndex__last> [ <messageIndex__INTEGER>])
;
<messageIndex__all> =
  ( all)
| ( any)
| ( anything)
| ( everything)
;
<messageIndex__1> =
  ( top)
| ( initial)
| ( first)
;
<messageIndex__last> =
  ( last)
| ( latest)
| ( most recent)
| ( newest)
| ( bottom)
| ( final)
;
<messageIndex__INTEGER> =
  ( <_CARDINAL_NUMBER_0_99999>)
| ( <_CARDINAL_NUMBER_0_9>+)
;
<messageIndex__INTEGER_ORDINAL> =
  ( <_ORDINAL_NUMBER_0_99999>)
;
```

```
<_I> =
( i)
;
<_WANT> =
  ( want)
| ( need)
| ( desire)
| ( would like)
;
<_COMPOSE> =
  ( compose)
| ( write)
;
<_SEND> =
  ( ( send | ship) [ it])
| ( "<s>" ( mail | email | e-mail | e mail) it)
;
<_FORWARD> =
  ( forward)
| ( pass [ on | along])
;
<_REPLY> =
  ( reply [ to])
| ( answer)
;
<_MAIL> =
  ( message)
| ( messages)
| ( [ electronic] ( mail | mails))
| ( email)
| ( emails)
| ( e-mail)
| ( e-mails)
| ( ( piece | pieces) of ( mail | email | e-mail))
| ( letter)
| ( letters)
| ( memo)
| ( memos)
| ( note)
| ( notes)
;
<_SENDER> =
  ( ( from | sent by) <sender>)
| ( <sender> "+s" <_MAIL>)
;
<_RECIPIENT> =
  ( ( [ addressed | sent] to) <recipient>)
;
<_RELATIVE> =
  ( <_THAT_WHICH> ( was | were | ( ( has | have) been)))
| ( "+s")   // as in "today+s message"
;
<_THAT_WHICH> =
  ( that)
| ( which)
;
<sender> =
  ( <name__STRING>)
| ( <emailAddress__STRING>)
;
<recipient> =
  ( <name__STRING>)
| ( <emailAddress__STRING>)
;
<name__STRING> =
  ( <_PERSON_OR_INSTITUTION_NAME>)
| ( <_MAILING_LIST_NAME>)
;
<_PERSON_OR_INSTITUTION_NAME> =
  ( "<personName>")
```

```
| ( lucy | donald | joseph | spamela | cynthia)
| ( <_WILDCARD>)
;
<_MAILING_LIST_NAME> =
  ( the <_WILDCARD>+ [ mailing] list)
;
<emailAddress__STRING> =
  ( <_EMAIL_ADDRESS_PART>+ <STRING__d064> <_EMAIL_ADDRESS_PART>+)
;
<_EMAIL_ADDRESS_PART> =
  ( <emailAddressPart__STRING>)
;
<emailAddressPart__STRING> =
  ( <_CHARACTER>)
| ( <_WILDCARD>)
;
// ==============================================================================
// Time expressions
// ==============================================================================
// ------------------------------------------------------------------------------
// Days of week
// ------------------------------------------------------------------------------
<dayOfWeek__0> =
  ( sunday)
;
<dayOfWeek__1> =
  ( monday)
;
<dayOfWeek__2> =
  ( tuesday)
;
<dayOfWeek__3> =
  ( wednesday)
;
<dayOfWeek__4> =
  ( thursday)
;
<dayOfWeek__5> =
  ( friday)
;
<dayOfWeek__6> =
  ( saturday)
;


// ------------------------------------------------------------------------------
// Months
// ------------------------------------------------------------------------------
<month__1> =
  ( january)
;
<month__2> =
  ( february)
;
<month__3> =
  ( march)
;
<month__4> =
  ( april)
;
<month__5> =
  ( may)
;
<month__6> =
  ( june)
;
<month__7> =
```

```
  ( july)
;
<month__8> =
  ( august)
;
<month__9> =
  ( september)
;
<month__10> =
  ( october)
;
<month__11> =
  ( november)
;
<month__12> =
  ( december)
;
// ----------------------------------------------------------------------------
// Cardinal Numbers
// ----------------------------------------------------------------------------
<_CARDINAL_NUMBER_0_99999> =
  ( <_CARDINAL_NUMBER_THOUSANDS> [ <_CARDINAL_NUMBER_HUNDREDS>]
    [ <_CARDINAL_NUMBER_TENS>] [ <_CARDINAL_NUMBER_UNITS>]
  )
| ( <_CARDINAL_NUMBER_0_999>)
;
<_CARDINAL_NUMBER_0_999> =
  ( <_CARDINAL_NUMBER_HUNDREDS> [ <_CARDINAL_NUMBER_TENS>]
    [ <_CARDINAL_NUMBER_UNITS>]
  )
| ( <_CARDINAL_NUMBER_0_99>)
;
<_CARDINAL_NUMBER_0_99> =
  ( <_CARDINAL_NUMBER_TENS> [ <_CARDINAL_NUMBER_UNITS>])
| ( <_CARDINAL_NUMBER_11_19>)
| ( <_CARDINAL_NUMBER_0_9>)
;
<_CARDINAL_NUMBER_0_9> =
  ( <_CARDINAL_NUMBER_UNITS>)
;
<_CARDINAL_NUMBER_11_19> =
  ( <INTEGER_CARDINAL__11>)
| ( <INTEGER_CARDINAL__12>)
| ( <INTEGER_CARDINAL__13>)
| ( <INTEGER_CARDINAL__14>)
| ( <INTEGER_CARDINAL__15>)
| ( <INTEGER_CARDINAL__16>)
| ( <INTEGER_CARDINAL__17>)
| ( <INTEGER_CARDINAL__18>)
| ( <INTEGER_CARDINAL__19>)
;
// For day of month
<_CARDINAL_NUMBER_1_31> =
  ( <INTEGER_CARDINAL__1>)
| ( <INTEGER_CARDINAL__2>)
| ( <INTEGER_CARDINAL__3>)
| ( <INTEGER_CARDINAL__4>)
| ( <INTEGER_CARDINAL__5>)
| ( <INTEGER_CARDINAL__6>)
| ( <INTEGER_CARDINAL__7>)
| ( <INTEGER_CARDINAL__8>)
| ( <INTEGER_CARDINAL__9>)
| ( <_CARDINAL_NUMBER_11_19>)
| ( <INTEGER_CARDINAL__20> [ <_CARDINAL_NUMBER_UNITS>])
| ( <INTEGER_CARDINAL__30> [ <INTEGER_CARDINAL__1>])
;
// For hour
<_CARDINAL_NUMBER_0_24> =
  ( <_CARDINAL_NUMBER_0_9>)
| ( <_CARDINAL_NUMBER_11_19>)
| ( <INTEGER_CARDINAL__20> [ <INTEGER_CARDINAL__1>])
```

```
| ( <INTEGER_CARDINAL__20> <INTEGER_CARDINAL__2>)
| ( <INTEGER_CARDINAL__20> <INTEGER_CARDINAL__3>)
| ( <INTEGER_CARDINAL__20> <INTEGER_CARDINAL__4>)
;

// For minute
<_CARDINAL_NUMBER_0_60> =
  ( <_CARDINAL_NUMBER_0_9>)
| ( <INTEGER_CARDINAL__10>)
| ( <_CARDINAL_NUMBER_11_19>)
| ( <INTEGER_CARDINAL__20> [ <_CARDINAL_NUMBER_UNITS>])
| ( <INTEGER_CARDINAL__30> [ <_CARDINAL_NUMBER_UNITS>])
| ( <INTEGER_CARDINAL__40> [ <_CARDINAL_NUMBER_UNITS>])
| ( <INTEGER_CARDINAL__50> [ <_CARDINAL_NUMBER_UNITS>])
| ( <INTEGER_CARDINAL__60>)
;

<_CARDINAL_NUMBER_UNITS> =
  ( <INTEGER_CARDINAL__0>)
| ( <INTEGER_CARDINAL__1>)
| ( <INTEGER_CARDINAL__2>)
| ( <INTEGER_CARDINAL__3>)
| ( <INTEGER_CARDINAL__4>)
| ( <INTEGER_CARDINAL__5>)
| ( <INTEGER_CARDINAL__6>)
| ( <INTEGER_CARDINAL__7>)
| ( <INTEGER_CARDINAL__8>)
| ( <INTEGER_CARDINAL__9>)
;

<_CARDINAL_NUMBER_TENS> =
  ( <INTEGER_CARDINAL__10>)
| ( <INTEGER_CARDINAL__20>)
| ( <INTEGER_CARDINAL__30>)
| ( <INTEGER_CARDINAL__40>)
| ( <INTEGER_CARDINAL__50>)
| ( <INTEGER_CARDINAL__60>)
| ( <INTEGER_CARDINAL__70>)
| ( <INTEGER_CARDINAL__80>)
| ( <INTEGER_CARDINAL__90>)
;

<_CARDINAL_NUMBER_HUNDREDS> =
  ( <INTEGER_CARDINAL__100>)
| ( <_CARDINAL_NUMBER_0_99> <INTEGER_CARDINAL__100>)
;

<_CARDINAL_NUMBER_THOUSANDS> =
  ( <INTEGER_CARDINAL__1000>)
| ( <_CARDINAL_NUMBER_0_999> <INTEGER_CARDINAL__1000>)
;

<INTEGER_CARDINAL__0> =
  ( 0)
| ( zero)
;

<INTEGER_CARDINAL__1> =
  ( 1)
| ( one)
;

<INTEGER_CARDINAL__2> =
  ( 2)
| ( two)
;

<INTEGER_CARDINAL__3> =
  ( 3)
| ( three)
;

<INTEGER_CARDINAL__4> =
  ( 4)
| ( four)
;

<INTEGER_CARDINAL__5> =
  ( 5)
| ( five)
;

<INTEGER_CARDINAL__6> =
  ( 6)
```

```
| ( six)
;
<INTEGER_CARDINAL__7> =
  ( 7)
| ( seven)
;
<INTEGER_CARDINAL__8> =
  ( 8)
| ( eight)
;
<INTEGER_CARDINAL__9> =
  ( 9)
| ( nine)
;
<INTEGER_CARDINAL__10> =
  ( 10)
| ( ten)
;
<INTEGER_CARDINAL__11> =
  ( 11)
| ( eleven)
;
<INTEGER_CARDINAL__12> =
  ( 12)
| ( twelve)
;
<INTEGER_CARDINAL__13> =
  ( 13)
| ( thirteen)
;
<INTEGER_CARDINAL__14> =
  ( 14)
| ( fourteen)
;
<INTEGER_CARDINAL__15> =
  ( 15)
| ( fifteen)
;
<INTEGER_CARDINAL__16> =
  ( 16)
| ( sixteen)
;
<INTEGER_CARDINAL__17> =
  ( 17)
| ( seventeen)
;
<INTEGER_CARDINAL__18> =
  ( 18)
| ( eighteen)
;
<INTEGER_CARDINAL__19> =
  ( 19)
| ( nineteen)
;
<INTEGER_CARDINAL__20> =
  ( 20)
| ( twenty)
;
<INTEGER_CARDINAL__30> =
  ( 30)
| ( thirty)
;
<INTEGER_CARDINAL__40> =
  ( 40)
| ( forty)
;
<INTEGER_CARDINAL__50> =
  ( 50)
| ( fifty)
;
```

```
<INTEGER_CARDINAL__60> =
  ( 60)
| ( sixty)
;
<INTEGER_CARDINAL__70> =
  ( 70)
| ( seventy)
;
<INTEGER_CARDINAL__80> =
  ( 80)
| ( eighty)
;
<INTEGER_CARDINAL__90> =
  ( 90)
| ( ninety)
;
<INTEGER_CARDINAL__100> =
  ( 100)
| ( hundred)
;
<INTEGER_CARDINAL__1000> =
  ( 1000)
| ( thousand)
;
// ---------------------------------------------------------------------------
// Ordinal Numbers
// ---------------------------------------------------------------------------
<_ORDINAL_NUMBER_0_99999> =
  ( <_ORDINAL_NUMBER_THOUSANDS>)
| ( <_CARDINAL_NUMBER_THOUSANDS> <_CARDINAL_NUMBER_HUNDREDS>
    <_ORDINAL_NUMBER_TENS> [ <_ORDINAL_NUMBER_UNITS>]
  )
| ( <_CARDINAL_NUMBER_THOUSANDS> <_CARDINAL_NUMBER_HUNDREDS>
    [ <_ORDINAL_NUMBER_TENS>] <_ORDINAL_NUMBER_UNITS>
  )
| ( <_ORDINAL_NUMBER_0_999>)
;
<_ORDINAL_NUMBER_0_999> =
  ( <_ORDINAL_NUMBER_HUNDREDS>)
| ( <_CARDINAL_NUMBER_HUNDREDS> <_CARDINAL_NUMBER_TENS>
    <_ORDINAL_NUMBER_UNITS>
  )
| ( <_ORDINAL_NUMBER_0_99>)
;
<_ORDINAL_NUMBER_0_99> =
  ( <_ORDINAL_NUMBER_TENS>)
| ( <_CARDINAL_NUMBER_TENS> <_ORDINAL_NUMBER_UNITS>)
| ( <_ORDINAL_NUMBER_11_19>)
| ( <_ORDINAL_NUMBER_UNITS>)
;
<_ORDINAL_NUMBER_11_19> =
  ( <INTEGER_ORDINAL__11>)
| ( <INTEGER_ORDINAL__12>)
| ( <INTEGER_ORDINAL__13>)
| ( <INTEGER_ORDINAL__14>)
| ( <INTEGER_ORDINAL__15>)
| ( <INTEGER_ORDINAL__16>)
| ( <INTEGER_ORDINAL__17>)
| ( <INTEGER_ORDINAL__18>)
| ( <INTEGER_ORDINAL__19>)
;
//<_ORDINAL_NUMBER_1_31> =
//   ( <_ORDINAL_NUMBER_UNITS>)
// | ( <INTEGER_ORDINAL__10>)
// | ( <_ORDINAL_NUMBER_11_19>)
// | ( <INTEGER_ORDINAL__20>)
// | ( <INTEGER_CARDINAL__20> <_ORDINAL_NUMBER_UNITS>)
// | ( <INTEGER_ORDINAL__30>)
// | ( <INTEGER_CARDINAL__30> <INTEGER_ORDINAL__1>)
//       ;
<_ORDINAL_NUMBER_UNITS> =
```

```
  ( <INTEGER_ORDINAL__1>)
| ( <INTEGER_ORDINAL__2>)
| ( <INTEGER_ORDINAL__3>)
| ( <INTEGER_ORDINAL__4>)
| ( <INTEGER_ORDINAL__5>)
| ( <INTEGER_ORDINAL__6>)
| ( <INTEGER_ORDINAL__7>)
| ( <INTEGER_ORDINAL__8>)
| ( <INTEGER_ORDINAL__9>)
;
<_ORDINAL_NUMBER_TENS> =
  ( <INTEGER_ORDINAL__10>)
| ( <INTEGER_ORDINAL__20>)
| ( <INTEGER_ORDINAL__30>)
| ( <INTEGER_ORDINAL__40>)
| ( <INTEGER_ORDINAL__50>)
| ( <INTEGER_ORDINAL__60>)
| ( <INTEGER_ORDINAL__70>)
| ( <INTEGER_ORDINAL__80>)
| ( <INTEGER_ORDINAL__90>)
;
<_ORDINAL_NUMBER_HUNDREDS> =
  ( <INTEGER_ORDINAL__100>)
| ( <_CARDINAL_NUMBER_0_99> <INTEGER_ORDINAL__100>)
;
<_ORDINAL_NUMBER_THOUSANDS> =
  ( <INTEGER_ORDINAL__1000>)
| ( <_CARDINAL_NUMBER_0_999> <INTEGER_ORDINAL__1000>)
;
<INTEGER_ORDINAL__1> =
  ( 1st)
| ( first)
;
<INTEGER_ORDINAL__2> =
  ( 2nd)
| ( second)
;
<INTEGER_ORDINAL__3> =
  ( 3rd)
| ( third)
;
<INTEGER_ORDINAL__4> =
  ( 4th)
| ( fourth)
;
<INTEGER_ORDINAL__5> =
  ( 5th)
| ( fifth)
;
<INTEGER_ORDINAL__6> =
  ( 6th)
| ( sixth)
;
<INTEGER_ORDINAL__7> =
  ( 7th)
| ( seventh)
;
<INTEGER_ORDINAL__8> =
  ( 8th)
| ( eighth)
;
<INTEGER_ORDINAL__9> =
  ( 9th)
| ( ninth)
;
<INTEGER_ORDINAL__10> =
  ( 10th)
| ( tenth)
;
<INTEGER_ORDINAL__11> =
  ( 11th)
| ( eleventh)
```

```
;
<INTEGER_ORDINAL__12> =
  ( 12th)
| ( twelfth)
;
<INTEGER_ORDINAL__13> =
  ( 13th)
| ( thirteenth)
;
<INTEGER_ORDINAL__14> =
  ( 14th)
| ( fourteenth)
;
<INTEGER_ORDINAL__15> =
  ( 15th)
| ( fifteenth)
;
<INTEGER_ORDINAL__16> =
  ( 16th)
| ( sixteenth)
;
<INTEGER_ORDINAL__17> =
  ( 17th)
| ( seventeenth)
;
<INTEGER_ORDINAL__18> =
  ( 18th)
| ( eighteenth)
;
<INTEGER_ORDINAL__19> =
  ( 19th)
| ( nineteenth)
;
<INTEGER_ORDINAL__20> =
  ( 20th)
| ( twentieth)
;
<INTEGER_ORDINAL__30> =
  ( 30th)
| ( thirtieth)
;
<INTEGER_ORDINAL__40> =
  ( 40th)
| ( fortieth)
;
<INTEGER_ORDINAL__50> =
  ( 50th)
| ( fiftieth)
;
<INTEGER_ORDINAL__60> =
  ( 60th)
| ( sixtieth)
;
<INTEGER_ORDINAL__70> =
  ( 70th)
| ( seventieth)
;
<INTEGER_ORDINAL__80> =
  ( 80th)
| ( eightieth)
;
<INTEGER_ORDINAL__90> =
  ( 90th)
| ( ninetieth)
;
<INTEGER_ORDINAL__100> =
  ( 100th)
| ( hundredth)
;
<INTEGER_ORDINAL__1000> =
  ( 1000th)
```

```
| ( thousandth)
;
// ---------------------------------------------------------------------------
// Characters
// ---------------------------------------------------------------------------
<_CHARACTER> =
  ( <STRING__d097>)
| ( <STRING__d098>)
| ( <STRING__d099>)
| ( <STRING__d100>)
| ( <STRING__d101>)
| ( <STRING__d102>)
| ( <STRING__d103>)
| ( <STRING__d104>)
| ( <STRING__d105>)
| ( <STRING__d106>)
| ( <STRING__d107>)
| ( <STRING__d108>)
| ( <STRING__d109>)
| ( <STRING__d110>)
| ( <STRING__d111>)
| ( <STRING__d112>)
| ( <STRING__d113>)
| ( <STRING__d114>)
| ( <STRING__d115>)
| ( <STRING__d116>)
| ( <STRING__d117>)
| ( <STRING__d118>)
| ( <STRING__d119>)
| ( <STRING__d120>)
| ( <STRING__d121>)
| ( <STRING__d122>)
| ( <STRING__d064>)
| ( <STRING__d046>)
| ( <STRING__d045>)
| ( <STRING__d095>)
| ( <STRING__d048>)
| ( <STRING__d049>)
| ( <STRING__d050>)
| ( <STRING__d051>)
| ( <STRING__d052>)
| ( <STRING__d053>)
| ( <STRING__d054>)
| ( <STRING__d055>)
| ( <STRING__d056>)
| ( <STRING__d057>)
;
<_DIGIT> =
  ( <STRING__d048>)
| ( <STRING__d049>)
| ( <STRING__d050>)
| ( <STRING__d051>)
| ( <STRING__d052>)
| ( <STRING__d053>)
| ( <STRING__d054>)
| ( <STRING__d055>)
| ( <STRING__d056>)
| ( <STRING__d057>)
;
<STRING__d097> =
  ( a)
;
<STRING__d098> =
  ( b)
;
<STRING__d099> =
  ( c)
;
<STRING__d100> =
  ( d)
;
```

```
<STRING__d101> =
  ( e)
;
<STRING__d102> =
  ( f)
;
<STRING__d103> =
  ( g)
;
<STRING__d104> =
  ( h)
;
<STRING__d105> =
  ( i)
;
<STRING__d106> =
  ( j)
;
<STRING__d107> =
  ( k)
;
<STRING__d108> =
  ( l)
;
<STRING__d109> =
  ( m)
;
<STRING__d110> =
  ( n)
;
<STRING__d111> =
  ( o)
;
<STRING__d112> =
  ( p)
;
<STRING__d113> =
  ( q)
;
<STRING__d114> =
  ( r)
;
<STRING__d115> =
  ( s)
;
<STRING__d116> =
  ( t)
;
<STRING__d117> =
  ( u)
;
<STRING__d118> =
  ( v)
;
<STRING__d119> =
  ( w)
;
<STRING__d120> =
  ( x)
;
<STRING__d121> =
  ( y)
;
<STRING__d122> =
  ( z)
;
<STRING__d064> =
  ( "@")
| ( at)
;
```

```
<STRING__d046> =
  ( ".")
| ( dot)
| ( period)
;
<STRING__d045> =
  ( "-")
| ( dash)
| ( hyphen)
| ( minus)
;
<STRING__d095> =
  ( "_")
| ( under score)
| ( underscore)
;
<STRING__d048> =
  ( 0)
| ( zero)
;
<STRING__d049> =
  ( 1)
| ( one)
;
<STRING__d050> =
  ( 2)
| ( two)
;
<STRING__d051> =
  ( 3)
| ( three)
;
<STRING__d052> =
  ( 4)
| ( four)
;
<STRING__d053> =
  ( 5)
| ( five)
;
<STRING__d054> =
  ( 6)
| ( six)
;
<STRING__d055> =
  ( 7)
| ( seven)
;
<STRING__d056> =
  ( 8)
| ( eight)
;
<STRING__d057> =
  ( 9)
| ( nine)
;
// =EOF=========================================================================
```

C.3 Musicbox Grammars

C.3.1 Generic Grammar

```
#JSGF V1.0 ISO8859-1 en;

/**
 * ============================================================================
 * @taskName        "Generic"
 * ----------------------------------------------------------------------------
 * @taskDescription ""
 * ----------------------------------------------------------------------------
 * @exampleSentence "reset"
 * ============================================================================
 * ============================================================================
 * @author           Matthias Denecke
 * ============================================================================
 */

grammar generic;

public <reset> =
( [please] ( reset | abort) [please] );

// =EOF========================================================================
```

C.3.2 Musicbox Grammar

```
#JSGF V1.0 ISO8859-1 en;

/**
 * ==========================================================================
 * @taskName        "Musicbox"
 * --------------------------------------------------------------------------
 * @taskDescription "where you can play and buy mp3s through spoken commands!"
 * --------------------------------------------------------------------------
 * @exampleSentence "what do you have"
 * @exampleSentence "play vaga luna"
 * @exampleSentence "add point blank to shopping cart"
 * @exampleSentence "wait means pause"
 * ==========================================================================
 * @stringMap       "'"             --> "+"
 * @stringMap       "+d like "      --> " would like "
 * @stringMap       "+s had "       --> " has had "
 * @stringMap       "+s been "      --> " has been "
 * @stringMap       " you+re "      --> " you are "
 * @stringMap       " what+s "      --> " what is "
 * @stringMap       " there+s "     --> " there is "
 * @stringMap       " that+s "      --> " that is "
 * @stringMap       " it+s "        --> " it is "
 * @stringMap       " i+m "         --> " i am "
 * @stringMap       "+s "           --> " +s "
 * @stringMap       " can+t "       --> " can not "
 * @stringMap       "n+t "          --> " not "
 * @stringMap       " per cent "    --> " percent "
 * ==========================================================================
 * @author           Matthias Denecke
 * ==========================================================================
 */

grammar musicbox;

<_WILDCARD> = "_$any$_";

public <listAllSongs> =
  ( (what | which | whose) [ <_SONG>] <_ARE_AVAILABLE> "</s>")
| ( [ please] <_SHOW> [ me] all [ <_AVAILABLE>] <_SONG> "</s>")
        ;

public <getInformation> =
  ( (list | get | give) [me] [some] ( information | info) [ on | about])
| ( [<_I>] <_VERB_DESIRE> know [ on | about])
| ( (what | which | whose) [ <_SONG>] <_ARE_AVAILABLE> ( in <language> | <_ARTIST>))
| ( (do you have | is there) (anything | any thing | ([a | any] <_SONG>)))
| ( [ please] <_SHOW> [ me] [ <_AVAILABLE>] <_SONG>)
        ;

public <play> =
( <_I> <_VERB_DESIRE> <_HEAR> | [ <_POLITE_WORD>] <_PLAY>);

public <pause> =
( [<_POLITE_WORD>] pause [<_POLITE_WORD>] );

public <stop> =
( [<_POLITE_WORD>] (stop | halt) [<_POLITE_WORD>]);

public <continue> =
( [<_POLITE_WORD>] (continue | go on) [<_POLITE_WORD>]);

public <fastForward> =
( [<_POLITE_WORD>] [fast] forward [<_POLITE_WORD>] );

public <fastBackward> =
( [<_POLITE_WORD>] [fast] backward [<_POLITE_WORD>] );

public <rewind > =
( [<_POLITE_WORD>] rewind [<_POLITE_WORD>] );

public <louder> =
( [<_POLITE_WORD>] [ <_PLAY>] louder [<_POLITE_WORD>] );
```

```
public <softer> =
( [<_POLITE_WORD>] [ <_PLAY>] softer [<_POLITE_WORD>] );

public <mute> =
( [<_POLITE_WORD>] mute [<_POLITE_WORD>] );

public <object> =
( <objectDescription> |
  <objectName> );

<objectName> =
( [<_quantity>] <recordingTitle> |
  [<_quantity>] <trackTitle> );
<objectDescription> =
  (  [<_quantity>] [<ordinal>] <_objectDescriptionArgument>+)
;
<_objectDescriptionArgument> =
  ( <_ADJECTIVE>)
| ( <objectSong>)
| ( <_ARTIST>)
| ( <recordingArtistName>)
| ( <_IN_LANGUAGE>)
;

<_IN_LANGUAGE> = in <language>;
<objectSong> =
( <_SONG>);
<_SONG> =
( item | items | song | songs | piece | pieces | hit | hits);
<_SHOW> =
( show | list | display)
;
<_ARE_AVAILABLE> =
  ( do you have [ <_AVAILABLE>])
| ( ( is | are) ( there | <_AVAILABLE>))
| ( (i can | can i) (hear | listen [ to]))
;
<_AVAILABLE> =
  ( available)
;

<_VERB_DESIRE> =
( want [to] |
will |
need [to] |
+d like |
like );
<_I> =
( i | we );
<_HEAR> =
( hear | listen to );

<_POLITE_WORD> =
( please );

<_quantity> =
( a | an | some | all | the );

<_ARTIST> =
( [by | from] <recordingArtistName>);


<_PLAY> =
( play );
<_ADJECTIVE> =
( great |
```

```
  greatest         |
  cool |
  coolest |
  groovy |
  hip );

<ordinal> =
( first | second | third | fourth | fifth | sixth | seventh | eighth
 | ninth | tenth | eleventh | twelfth ) ;


//------------------------------------------ to be read in from the database
<trackTitle> =
<_WILDCARD>;

//------------------------------------------ to be read in from the database
<recordingArtistName> =
<_WILDCARD>;

//------------------------------------------ to be read in from the database
<recordingTitle> =
<_WILDCARD>;

//------------------------------------------ to be read in from the database
<language> =
<_WILDCARD>;

// =EOF=====================================================================
```

C.3.3 Shopping Cart Grammar

```
#JSGF V1.0 ISO8859-1 en;

/*
 * ============================================================================
 * @taskName         "Shopping Cart"
 * ----------------------------------------------------------------------------
 * @taskDescription  "where you can shop through spoken commands!"
 * ----------------------------------------------------------------------------
 * @exampleSentence "buy"
 * ============================================================================
 * ============================================================================
 * @author          Matthias Denecke
 * ============================================================================
 */

grammar shoppingcart;

public <addToCart> =
( add | buy );

public <removeFromCart> =
( remove );

public <checkout> =
( [please] checkout [please] |
  [okay] i am done );

public <balance> =
( [what] [is] [the | my] balance |
  how much is that );

public <toCart> =
( [in | to | into] [my | the] [shopping] cart );

public <fromCart> =
( from [my | the] [shopping] cart );

// =EOF=======================================================================
```

Appendix D

Utterances from User Sessions

D.1 E-Mail Task

D.1.1 User 1

do i have any mail

check for new mail

sort mail by date in descending order

delete mails dated 12th april

reply to message from cynthia

compose to bob

main

check mail

delete mail with size 600

show topics on nasdaq

read mail nasdaq soars

check mail

sort by size descending

D.1.2 User 2

did i get any email from peter

did i get any email today

read the email from spamela

see the list of current emails again

see email from goku

delete email from goku

see cynthia's email

reply to cynthia

list

read email from cynthia

reply to this message

reply to cynthia's email

read email from cynthia again

reply

reply to cynthia

list

read donald

list

read joseph

read donald

list

read donald's email

see donald's email

reply to joseph

list

show message from donald

reply to donald

remove email from donald

i want to send peter a message

compose email to peter

send

list

arrange email by date

read lucy's email

see lucy

open lucy's email

list

D.1.3 User 3

do i have mail

sort messages by date

sort messages by time

delete newswire messages

delete messages from newswire

reply to spameta

check messages

do i have any mail

create a new message to goku

send message to goku

display new messages

list messages

show messages from spameta

show messages from spamela

sort messages by size

delete message from lucy

undelete message from lucy

retrieve message from lucy

undo last deleted message

undo delete

show all messages about travel

display all messages

list all messages

display message from searchbot

list messages

reply to cynthia

list messages

open message from donald

display message from donald

show message from lucy

display message from lucy

reply to lucy

reply to message from lucy

list messages from newswire

sort newswire messages by date

list messages from donald

display messages from donald

D.1.4 User 4

please check my mail

scroll down

delete spamela's emails

delete spamela's mails

delete get rich

delete spam mail

sort by date

please arrange messages by recency

read goku

delete spamela

delete 3rd

read mail

list mail

sort by date

delete 5th message

delete 3rd message

delete 6th message

read 4th message

reply

reply to 4th message

send message to bob

D.1.5 User 5

can you check if i got a new mail from cynthia

reply to cynthia

ok send it

draft a new mail

draft a new mail to bob

spell check the message

send message

show me all the mails i got today

show me mails from cynthia

open cynthia's mail

find messages with subject potluck

now find messages with subject nasdaq

delete messages from today

delete all messages from spamela

delete everything

sort by date

sort by size

D.1.6 User 6

do i have any new messages

open message from cynthia

reply to cynthia

send

go back to messages

delete messages from spamela

check messages

sort messages by date

message to bob

send

open messages

list messages

D.1.7 User 7

do i have any mail

sort my mail by date received

did i receive mail yesterday

read mail from cynthia

reply to cynthia

compose a message

compose a message to bob

send message to bob

check for new mail

sort mail by date

delete mail from goku

read mail from searchbot

close mail from searchbot

read mail from cynthia

display inbox

read mail from don

list mail

reply to lucy's mail

send lucy mail

list mail

forward mail from lucy to bob

send mail

check for new mail

list mail

read mail from josehp

read mail from joseph

close mail from joseph

D.1.8 User 8

can i see my mails

show me my messages by date

will you get rid of mails from newswire

i would like to reply to cynthia

can i draft a new message to bob

can i create a new message to bob

D.1.9 User 9

check my email

kill spam

check mail

newest first

show all messages

sort by date

read cynthia's message

reply to her

reply to cynthia

send it

write to bob

check spelling

send it

show messages

put lists first

show all messages

put important messages first

delete mail from spamela

block mail from lucy

D.1.10 User 10

do i have mail

show me all mail from donald

sort the mail by date

sort the mail by size

sort the mail by sender

delete all mail from newswire

read cythia's message

read cynthia's message

reply to cynthia's message

show me my mail

i want to send a message to bob

get rid of all messages from spamela

ignore message from lucy

respond to goku's message

D.2 Musicbox Task

D.2.1 User 1

what songs do you have

play thunder road

ff

ff

fb

ff

toss thunder road into cart

stop

drop thunder road from cart

list all songs

play another song by springsteen

play fade

shut up

list all songs

buy jamming

buy exodus

show balance

go to checkout

D.2.2 User 2

what do you have

play jamming

how much is it

add bob marley to my shopping cart

add jamming to my shopping cart

what do you have

make it softer

play overture

what's my current balance

what's my current balance

buy overture

what's my total

play lullaby

a little louder

how much is lullaby

what's my total

what's my total

remove overture from my shopping cart

how much do i owe

what other music do you have

stop playing lullaby

play fade

i'm ready to pay

stop

D.2.3 User 3

what do you have

play three little birds

put song in my shopping basket

put three little birds in my basket

stop

what do you have

play no woman no cry

make it louder

fast forward

put no woman no cry into basket

quiet please

what's my current total

time to pay up

D.2.4 User 4

search for songs by title

find songs played by they might be giants

what songs do i have now

get new songs

buy a song

list songs

play a song

what do you have

remove songs by bob marley

add jamming to my shopping cart

add fade to my shopping cart

add border to my shopping cart

remove all bruce springsteen songs from my shopping cart

remove fade from my shopping cart

how much have i spents so far

add i shot the sheriff

add all tom waits songs

play jamming

make is softer

fast forward a bit

stop playing

check out

D.2.5 User 5

show me all the songs

play caro mio ben

repeat caro mio ben

lower the volume

show me all the songs

what songs do you have

play caro mio ben

repeat caro mio ben

louder

louder

lower the volume

repeat caro mio ben

rewind

fast forward

stop

i want to buy caro mio ben

buy o cessate

buy vaga luna

what's my balance

ok proceed to checkout

Appendix E

Learning Episodes

E.1 E-Mail Task's Detailed Results

E.1.1 User 1

E.1.1.1 Summary

Duration:	7 minutes
Number of utterances:	13
Number of learning episodes:	3
Number of cancelations:	0
Average number of choices:	5.67
Number of rules learned:	7
Average GSG score:	1.57

E.1.1.2 Details

LE e.1.1

Trigger utterance: *check for new email*

Total number of choices: 2

SubRHS learned under <voiceEMail.listMail>:

<voiceEMail._ASK_TO_CHECK> *for* <voiceEMail._MAIL_ARGUMENT>*

Original rule:

```
public <listMail> = [ <_VERB_DESIRE>] (<_LIST> [<_TO_FOR_ME>] |
<_ASK_TO_CHECK>) <_MAIL_ARGUMENT>*;
```

Resulting rule:

```
public <listMail> = [ <_VERB_DESIRE>] (<_LIST> [<_TO_FOR_ME>] |
<_ASK_TO_CHECK> | <_ASK_TO_CHECK> for <_MAIL_ARGUMENT>*) <_MAIL_ARGUMENT>*);
```

Score: 2-. An expert grammar writer would have created a more compact rule, e.g., public <listMail> = [<_VERB_DESIRE>] (<_LIST> [<_TO_FOR_ME>] | <_ASK_TO_CHECK> for) <_MAIL_ARGUMENT>*; but the coverage would be the same.

SubRHS learned under <voiceEMail._ASK_TO_CHECK>:

check

Original rule:

```
<_ASK_TO_CHECK> = do i have | is there [ any | anything] | there is [ any |
anything];
```

Resulting rule:

```
<_ASK_TO_CHECK> = do i have | is there [ any | anything] | there is [ any |
anything] | check;
```

Score: 2

LE e.1.2

Trigger utterance: *main*

Total number of choices: 8

SubRHS learned under `<voiceEMail.countMail>`:

`<voiceEMail._KNOW>`

Original rule:

```
public <countMail> = [<_VERB_DESIRE>] <_COUNT> [<_TO_FOR_ME>]
<_MAIL_ARGUMENT>* | [<_VERB_DESIRE>] [<_KNOW>] <_HOW_MANY> <_MAIL_ARGUMENT>*;
```

Resulting rule:

```
public <countMail> = [<_VERB_DESIRE>] <_COUNT> [<_TO_FOR_ME>]
<_MAIL_ARGUMENT>* | [<_VERB_DESIRE>] [<_KNOW>] <_HOW_MANY> <_MAIL_ARGUMENT>*
| <_KNOW>;
```

Score: 1. Slightly surprising rule but user did select `<countMail>` as the meaning of *main*.

SubRHS learned under `voiceEMail._KNOW`:

main

Original rule:

```
<KNOW> = know | find out;
```

Resulting rule:

```
<KNOW> = know | find out | main;
```

Score: 2. Again, user explicitly said that *main* is an example of *know*.

LE e.1.3

Trigger utterance: *show topics on nasdaq*

Total number of choices: 7

SubRHS learned under <voiceEMail.listMail>:

```
<voiceEMail._ASK_TO_CHECK> <voiceEMail._LIST> <voiceEMail._MAIL_ARGUMENT>*
```

Original rule:

```
public <listMail> = [<_VERB_DESIRE>] (<_LIST> [<_TO_FOR_ME>] |
<_ASK_TO_CHECK> | <_ASK_TO_CHECK> for <_MAIL_ARGUMENT>*) <_MAIL_ARGUMENT>*;
```

Resulting rule:

```
public <listMail> = [<_VERB_DESIRE>] (<_LIST> [<_TO_FOR_ME>] |
<_ASK_TO_CHECK> | <_ASK_TO_CHECK> for <_MAIL_ARGUMENT>* | <_ASK_TO_CHECK>
<_LIST> <_MAIL_ARGUMENT>*) <_MAIL_ARGUMENT>*;
```

Score: 2-. Same comment as in e.1.1.

SubRHS learned under <voiceEMail._ASK_TO_CHECK>:

show

Original rule:

```
<_ASK_TO_CHECK> = do i have | is there [ any | anything] | there is [ any |
anything] | check;
```

Resulting rule:

```
<_ASK_TO_CHECK> = do i have | is there [ any | anything] | there is [ any |
anything] | check | show;
```

Score: 2

SubRHS learned under <voiceEMail._LIST>:

topics

Original rule:

```
<voiceEMail._LIST> = list | get | search [for];
```

Resulting rule:

```
<voiceEMail._LIST> = list | get | search [for] | topics;
```

Score: 1. An expert grammmar writer would probably placed *topics* under `SUBJECT_PRE` but user answered yes to whether *topics* is a way to express *list.*

E.1.2 User 2

E.1.2.1 Summary

Duration:	17 minutes
Number of utterances:	37
Number of learning episodes:	6
Number of cancelations:	0
Average number of choices:	6.17
Number of rules learned:	5
Average GSG score:	1.33

E.1.2.2 Details

LE e.2.1

Trigger utterance: *see email from goku*

Total number of choices: 5. User initiative at choice 3.

SubRHS learned under `<voiceEMail._READ>`:

see

Original rule:

```
<_READ> = read | print | tell [me about];
```

Resulting rule:

```
<_READ> = read | print | tell [me about] | see;
```

Score: 2

LE e.2.2

Trigger utterance: *read donald*

Total number of choices: 10

SubRHS learned: None.

Score: 1. User didn't like any of the choices presented.

LE e.2.3

Trigger utterance: *show message from donald*

Total number of choices: 5

SubRHS learned under `<voiceEMail._READ>`:

show

Original rule:

```
<_READ> = read | print | tell [me about] | see;
```

Resulting rule:

```
<_READ> = read | print | tell [me about] | see | show;
```

Score: 2

LE e.2.4

Trigger utterance: *remove email from donald*

Total number of choices: 9

SubRHS learned under `<voiceEMail._DELETE>`:

remove

Original rule:

```
<_DELETE> = delete;
```

Resulting rule:

```
<_DELETE> = delete | remove;
```

Score: 2

LE e.2.5

Trigger utterance: *arrange email by date*

Total number of choices: 2

SubRHS learned under `<voiceEMail._SORT>`:

resort

Original rule:

```
<_SORT> = sort;
```

Resulting rule:

```
<_SORT> = sort | arrange;
```

Score: 2

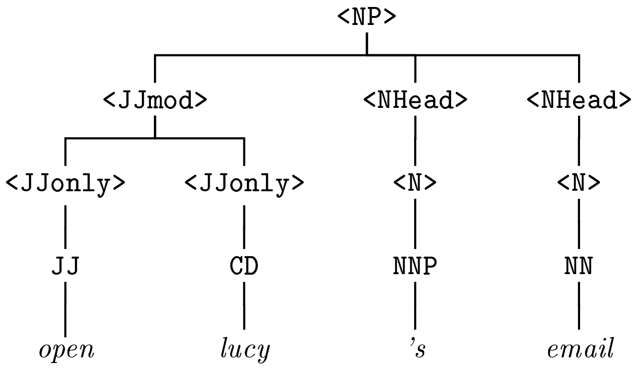


Figure E.1: Syntactic parse of *open lucy's email*. Note that *open* is tagged as adjective rather than verb, and *lucy* as a cardinal number rather than proper noun. The wrong tagging misleads the learning methods, which construct the rule in LE e.2.6.

LE e.2.6

Trigger utterance: *open lucy's email*

Total number of choices: 6

SubRHS learned under <voiceEMail._READ>:

open lucy

Original rule:

```
<_READ> = read | print | tell [me about] | see | show;
```

Resulting rule:

```
<_READ> = read | print | tell [me about] | see | show | open lucy;
```

Score: -1. Error arises from wrong segmentation, which is triggered by the syntactic parse in Figure E.1. Both *open* and *lucy* are OOV words and since they are syntactically parsed under a noun phrase they are thought to belong together.

E.1.3 User 3

E.1.3.1 Summary

Duration:	28 minutes
Number of utterances:	37
Number of learning episodes:	8
Number of cancelations:	2
Average number of choices:	7.50
Number of rules learned:	4
Average Gsg score:	1.33

E.1.3.2 Details

LE e.3.1

Trigger utterance: *reply to spameta*

Total number of choices: 9

SubRHS learned: None.

Score: 1. The original utterance was correctly parsed but due to a bug in the application, no reply message was generated, at which point the user said *wrong* and triggered a fruitless learning episode.

Cancelation e.3.1

```
> check messages
I don't understand right away what you mean but let me guess...
"check messages" is a way to express:
     1.  list mail, e.g.  "list"
     2.  count mail, e.g.  "count"
     0.  None of the above
> 1
"check " is a way to express:
     1.  ask to check, e.g.  "do i have"
     2.  list, e.g.  "list"
     0.  None of the above
> cancel
Ok, back to square one
```

Comment: User did not like choices — there could be some learning curve to the interface, as later in the session she did map her intentions to the presented choices.

Cancelation e.3.2

```
> undo last deleted message
I don't understand right away what you mean but let me guess...
"undo last deleted message" is a way to express:
     1.  list mail, e.g.  "list"
     2.  count mail, e.g.  "count"
     3.  send mail, e.g.  "send"
     0.  None of the above
> 0
Is "undo last deleted message" a way to express reply mail, e.g.  "reply"?
     1.  Yes
     2.  No
> 0
By "0" are you referring to:
     1.  integer cardinal 0, e.g.  "0"
     2.  ordinal number 0 99, e.g.  "10th"
     3.  ordinal number 0 999, e.g.  "100th"
     0.  None of the above
> cancel
```

Comment: GSG's interface, always eager for a flexible, mixed-initiative dialogue, assumes that *0*, since it is not one of the options given to the second question above, refers to the anchor mother; whereas the user obviously was using *0* to mean option "`2. No`," accustomed to the constant "`0. None of the above`" option in multiple-choice questions. The solution would be of course to change "`1. Yes`, `2. No`," into a more consistent "`1. Yes`, `0. No`."

LE e.3.2

Trigger utterance: *create a new message to goku*

Total number of choices: 5

SubRHS learned under <voiceEMail.composeMail>:

```
<voiceEMail._COMPOSE> <voiceEMail._ARTICLE> <voiceEMail._MAIL_ARGUMENT>*
```

Original rule:

```
public <voiceEMail.composeMail> = [<_VERB_DESIRE>] <_COMPOSE>
<_MAIL_ARGUMENT>*;
```

Resulting rule:

```
public <voiceEMail.composeMail> = [<_VERB_DESIRE>] ( <_COMPOSE> |
<_COMPOSE> <_ARTICLE> <_MAIL_ARGUMENT>* ) <_MAIL_ARGUMENT>*;
```

Score: 2-.

SubRHS learned under <voiceEMail._COMPOSE>:

create

Original rule:

```
public <_COMPOSE> = compose | write;
```

Resulting rule:

```
public <_COMPOSE> = compose | write | create;
```

Score: 2.

LE e.3.3

Trigger utterance: *display new messages*

Total number of choices: 5

SubRHS learned under <voiceEMail._READ>:

display

Original rule:

```
<_READ> = read | print | tell [me about];
```

Resulting rule:

```
<_READ> = read | print | tell [me about] | display;
```

Score: 2

LE e.3.4

Trigger utterance: *show messages from spameta*

Total number of choices: 5

SubRHS learned under `<voiceEMail._LIST>`:

show

Original rule:

```
<_LIST> = list | get | search [for];
```

Resulting rule:

```
<_LIST> = list | get | search [for] | show;
```

Score: 2

LE e.3.5

Trigger utterance: *undelete message from lucy*

Total number of choices: 9

SubRHS learned: None.

Score: 1. Out of domain.

LE e.3.6

Trigger utterance: *retrieve message from lucy*

Total number of choices: 9

SubRHS learned: None.

Score: 1. Out of domain.

LE e.3.7

Trigger utterance: *undo last deleted message*

Total number of choices: 9

SubRHS learned: None.

Score: 1. Out of domain.

LE e.3.8

Trigger utterance: *open message from donald*

Total number of choices: 9

SubRHS learned: None.

Score: 1. User did not accept any choice, including *list mail* and *read mail.*

E.1.4 User 4

E.1.4.1 Summary

Duration:	14 minutes
Number of utterances:	21
Number of learning episodes:	2
Number of cancelations:	1
Average number of choices:	10.00
Number of rules learned:	4
Average GSG score:	2.00

E.1.4.2 Details

LE e.4.1

Trigger utterance: *please check my mail*

Total number of choices: 4

SubRHS learned under `<voiceEMail.readMail>`:

`[<voiceEMail._VERB_DESIRE>] <voiceEMail._READ>` *my* `<voiceEMail._MAIL_ARGUMENT>*`

Original rule:

```
public <readMail> = [<_VERB_DESIRE>] <_READ> <_MAIL_ARGUMENT>*;
```

Resulting rule:

```
public <readMail> = [<_VERB_DESIRE>] ( <_READ> | [<_VERB_DESIRE>] <_READ> my
<_MAIL_ARGUMENT>* ) <_MAIL_ARGUMENT>*;
```

Score: 2-

SubRHS learned under `<voiceEMail._READ>`:

check

Original rule:

```
<_READ> = read | print | tell [me about];
```

Resulting rule:

```
<_READ> = read | print | tell [me about] | check;
```

Score: 2

Cancelation e.4.1

```
> scroll down
I don't understand right away what you mean but let me guess...
"scroll down" is a way to express:
     1.  count mail, e.g.  "count"
     2.  list mail, e.g.  "list"
     3.  read mail, e.g.  "read"
     0.  None of the above
> cancel
Ok, back to square one
```

Comment: User probably realized that his intention was not supported by the application.

LE e.4.2

Trigger utterance: *please arrange messages by recency*

Total number of choices: 16

SubRHS learned under `<voiceEMail._SORT>`:

arrange

Original rule:

`<_SORT> = sort;`

Resulting rule:

`<_SORT> = sort | arrange;`

Score: 2

SubRHS learned under `<voiceEMail.sortBy_date>`:

recency

Original rule:

`<sortBy_date> = date | time | the order in which <_MAIL> was recieved;`

Resulting rule:

`<sortBy_date> = date | time | the order in which <_MAIL> was recieved | recency;`

Score: 2

E.1.5 User 5

E.1.5.1 Summary

Duration:	12 minutes
Number of utterances:	17
Number of learning episodes:	8
Number of cancelations:	0
Average number of choices:	6.25
Number of rules learned:	6
Average GSG score:	1.67

E.1.5.2 Details

LE e.5.1

Trigger utterance: *can you check if i got a message from cynthia*

Total number of choices: 21

SubRHS learned under <voiceEMail.listMail>:

can you check if <voiceEMail._I_HAVE> <voiceEMail._MAIL_ARGUMENT>*

Original rule:

```
public <listMail> = [ <_VERB_DESIRE>] (<_LIST> [<_TO_FOR_ME>] |
<_ASK_TO_CHECK>) <_MAIL_ARGUMENT>*;
```

Resulting rule:

```
public <listMail> = [ <_VERB_DESIRE>] (<_LIST> [<_TO_FOR_ME>] |
<_ASK_TO_CHECK> | can you check if <_I_HAVE> <_MAIL_ARGUMENT>* )
<_MAIL_ARGUMENT>*;
```

Score: 1-. An expert grammar writer would have probably place *can you check if* under <_ASK_TO_CHECK>, but segmentation algorithm treated each word as a separate segment.

LE e.5.2

Trigger utterance: *draft a new mail*

Total number of choices: 3

SubRHS learned under `<voiceEMail.composeMail>`:

```
<voiceEMail._COMPOSE> <voiceEMail._ARTICLE> <voiceEMail._MAIL_ARGUMENT>*
```

Original rule:

```
public <voiceEMail.composeMail> = [<_VERB_DESIRE>] <_COMPOSE>
<_MAIL_ARGUMENT>*;
```

Resulting rule:

```
public <voiceEMail.composeMail> = [<_VERB_DESIRE>] ( <_COMPOSE> |
<_COMPOSE> <_ARTICLE> <_MAIL_ARGUMENT>* ) <_MAIL_ARGUMENT>*;
```

Score: 1. A human writer would have simply inserted `[ <_ARTICLE>]` between `<_COMPOSE>` and `<_MAIL_ARGUMENT>*`, a much more compact rule than the produced by GSG, but the coverage would be the same.

SubRHS learned under `voiceEMail._COMPOSE`:

draft

Original rule:

```
<voiceEMail._COMPOSE> = compose | write;
```

Resulting rule:

```
<voiceEMail._COMPOSE> = compose | write | draft;
```

Score: 2

LE e.5.3

Trigger utterance: *draft a new mail to bob*

Total number of choices: 0

SubRHS learned: None.

Score: 2. Re-use of rule acquired in e.5.2.

LE e.5.4

Trigger utterance: *spell check the message*

Total number of choices: 11

SubRHS learned: None.

Score: 1. Out of domain.

LE e.5.5

Trigger utterance: *show me all the mails i got today*

Total number of choices: 4

SubRHS learned under `<voiceEMail._LIST>`:

show

Original rule:

```
<_LIST> = list | get | search [for];
```

Resulting rule:

```
<_LIST> = list | get | search [for] | show;
```

Score: 2

LE e.5.6

Trigger utterance: *open cynthia's mail*

Total number of choices: 5

SubRHS learned under `<voiceEMail._READ>`:

see

Original rule:

```
<_READ> = read | print | tell [me about];
```

Resulting rule:

```
<_READ> = read | print | tell [me about] | open;
```

Score: 2

LE e.5.7

Trigger utterance: *find messages with subject potluck*

Total number of choices: 6

SubRHS learned under `<voiceEMail._LIST>`:

find

Original rule:

```
<_LIST> = list | get | search [for] | show;
```

Resulting rule:

```
<_LIST> = list | get | search [for] | show | find;
```

Score: 2

LE e.5.8

Trigger utterance: *now find messages with subject nasdaq*

Total number of choices: 0

SubRHS learned: None.

Score: 2. Re-use of rule acquired in e.5.7. (Harmless skipping of *now*.)

E.1.6 User 6

E.1.6.1 Summary

Duration:	8 minutes
Number of utterances:	12
Number of learning episodes:	5
Number of cancelations:	0
Average number of choices:	5.00
Number of rules learned:	4
Average GSG score:	1.00

E.1.6.2 Details

LE e.6.1

Trigger utterance: *open message from cynthia*

Total number of choices: 5

SubRHS learned under <voiceEMail._READ>:

open

Original rule:

```
<_READ> = read | print | tell [me about];
```

Resulting rule:

```
<_READ> = read | print | tell [me about] | open;
```

Score: 2

LE e.6.2

Trigger utterance: *go back to messages*

Total number of choices: 10

SubRHS learned under <voiceEMail.readMail>:

go back <voiceEMail._DATE_RANGE_IN> <voiceEMail._MAIL_ARGUMENT>*

Original rule:

```
public <readMail> = [<_VERB_DESIRE>] <_READ> <_MAIL_ARGUMENT>*;
```

Resulting rule:

```
public <readMail> = [<_VERB_DESIRE>] ( <_READ> | go back <_DATE_RANGE_IN>
<_MAIL_ARGUMENT>* ) <_MAIL_ARGUMENT>*;
```

Score: 2-. Harmless generalization of *to* to <_DATE_RANGE_IN>.

LE e.6.3

Trigger utterance: *check messages*

Total number of choices: 3

SubRHS learned under `<voiceEMail._ASK_TO_CHECK>`:

check

Original rule:

`<_ASK_TO_CHECK> = do i have | is there [ any | anything] | there is [ any | anything];`

Resulting rule:

`<_ASK_TO_CHECK> = do i have | is there [ any | anything] | there is [ any | anything] | check;`

Score: 2

LE e.6.4

Trigger utterance: *message to bob*

Total number of choices: 4

SubRHS learned under `<voiceEMail.composeMail>`:

`<voiceEMail._MAIL_ARGUMENT>+`

Original rule:

```
public <composeMail> = [<_VERB_DESIRE>] <_COMPOSE> <_MAIL_ARGUMENT>*;
```

Resulting rule:

```
public <composeMail> = [<_VERB_DESIRE>] ( <_COMPOSE> | <_MAIL_ARGUMENT>+ )
<_MAIL_ARGUMENT>*;
```

Score: -2. This rule has the potential of disrupting the grammar, because, henceforth, a single `<_MAIL_ARGUMENT>` can be considered a `<composeMail>`. It was not flagged as changing the parsebank because in fact the parsebank does not change, since all the current parses remain achievable. Also, the rule by itself does not introduce ambiguity nor is subsumed by any existing rule. In fact, it is a specialization of a rule such as the very same RHS of `<composeMail>` before the addition. A solution to prevent this overtly horizontally generalized rule would be to have a "negative parsebank" of phrases that should remain unparsable. Another solution (and the one used to create the Union Grammar) is to add beginning and end of utterance markers to the acquired RHS so that, e.g., *message to bob* be parsed as `<_MAIL_ARGUMENT>` only if that's the entire sentence. In general, however, we would lose the generalization achieved at the merging of the new RHS with the original rule, e.g., insertion after `<_VERB_DESIRE>`. In any event, a "reset" command can always be given to revert to the original grammar.

Interaction e.6.5

```
> open messages
I understand "open messages"
> wrong
Is "open messages" a way to express read mail, e.g.  "read"?
     1.  Yes
     2.  No
> 2
"open messages" is a way to express:
     1.  count mail, e.g.  "count"
     2.  list mail, e.g.  "list"
     3.  compose mail, e.g.  "compose"
     0.  None of the above
> 2
Sorry, I can't learn what you mean as it conflicts with my existing
knowledge.
```

LE e.6.5

Trigger utterance: *open messages*

Total number of choices: 3

SubRHS learned: None.

Score: 1. See Interaction e.6.5. Detection of conflict since user had said (see LE e.6.1) that *open* means `readMail`. Of course the user may argue that *open* means `readMail` if he wants to open a single piece of mail, whereas it means `listMail` if more than one piece of mail match his criteria. That, however, is beyond the abilities of GSG given the context-freedom of the grammar.

E.1.7 User 7

E.1.7.1 Summary

Duration:	20 minutes
Number of utterances:	27
Number of learning episodes:	7
Number of cancelations:	1
Average number of choices:	3.00
Number of rules learned:	5
Average GSG score:	1.63

E.1.7.2 Details

LE e.7.1

Trigger utterance: *sort means rank*

Total number of choices: 1

SubRHS learned: None.

Score: 1. As it turns out, *sort* was already in the grammar (but not *rank*). Cf. *rank means sort*.

LE e.7.2

Trigger utterance: *did i receive any mail yesterday*

Total number of choices: 9

SubRHS learned under <voiceEMail.listMail>:

did <voiceEMail._I> *receive* <voiceEMail._MAIL_ARGUMENT>*

Original rule:

```
public <listMail> = [<_VERB_DESIRE>] ( <_LIST> [<_TO_FOR_ME>] |
<_ASK_TO_CHECK> ) <_MAIL_ARGUMENT>*;
```

Resulting rule:

```
public <listMail> = [<_VERB_DESIRE>] ( <_LIST> [<_TO_FOR_ME>] |
<_ASK_TO_CHECK> | did <_I> receive <_MAIL_ARGUMENT>* ) <_MAIL_ARGUMENT>*;
```

Score: 1-. An expert grammar writer would have probably placed *did i receive* under <_ASK_TO_CHECK>, but segmentation algorithm treated each word as a separate segment.

LE e.7.3

Trigger utterance: *check for new mail*

Total number of choices: 3

SubRHS learned under `<voiceEMail.countMail>`:

`[<voiceEMail._KNOW>]` *for* `<voiceEMail._MAIL_ARGUMENT>*`

Original rule:

```
public <countMail> = [<_VERB_DESIRE>] <_COUNT> [<_TO_FOR_ME>]
<_MAIL_ARGUMENT>* | [<_VERB_DESIRE>] [<_KNOW>] <_HOW_MANY> <_MAIL_ARGUMENT>*;
```

Resulting rule:

```
public <countMail> = [<_VERB_DESIRE>] <_COUNT> [<_TO_FOR_ME>]
<_MAIL_ARGUMENT>* | [<_VERB_DESIRE>] [<_KNOW>] <_HOW_MANY> <_MAIL_ARGUMENT>*
| [<_KNOW>] for <_MAIL_ARGUMENT>*;
```

Score: 2-

SubRHS learned under `voiceEMail._KNOW`:

check

Original rule:

```
<KNOW> = know | find out;
```

Resulting rule:

```
<KNOW> = know | find out | check;
```

Score: 2. User explicitly said that *check* is an example of *know*.

Cancelation e.7.1

```
> close mail from searchbot
I don't understand right away what you mean but let me guess...
"close mail from searchbot" is a way to express:
     1.  count mail, e.g.  "count"
     2.  reply mail, e.g.  "reply"
     3.  list mail, e.g.  "list"
     0.  None of the above
> cancel
Ok, back to square one
```

Comment: User may have realized that no concept matches his intention. But see LE e.7.4.

LE e.7.4

Trigger utterance: *close mail means list mail*

Total number of choices: 5

SubRHS learned under `<voiceEMail.listMail>`:

close `<_MAIL_ARGUMENT>*`

Original rule:

```
public <listMail> = [<_VERB_DESIRE>] ( <_LIST> [<_TO_FOR_ME>] |
<_ASK_TO_CHECK> | did <_I> receive <_MAIL_ARGUMENT>* ) <_MAIL_ARGUMENT>*;
```

Resulting rule:

```
public <listMail> = [ <_VERB_DESIRE>] (<_LIST> [<_TO_FOR_ME>] |
<_ASK_TO_CHECK> | did <_I> receive <_MAIL_ARGUMENT>* | close
<_MAIL_ARGUMENT>*) <_MAIL_ARGUMENT>*;
```

Score: 2-. This meaning equivalence may be a bit surprising. Most likely, the user wanted to close the current message and then list all the messages (cf. LE e.6.2 *go back to messages*) and, since there is not explicit "`<closeMail>`" concept in the ontology, the most effective way is to map *close* into `<listMail>`.

LE e.7.5

Trigger utterance: *display inbox*

Total number of choices: 3

SubRHS learned under `<voiceEMail._LIST>`:

display inbox

Original rule:

```
<voiceEMail._LIST> = list | get | search [for];
```

Resulting rule:

```
<voiceEMail._LIST> = list | get | search [for] | display inbox;
```

Score: 1. Similar case as in LE e.2.6. To begin with, the first part-of-speech for *display*, was, according to the POS lexicon used, `NN`, followed by `VB`. So the syntactic parse in Figure E.2 is obtained, which groups the two unknown words.

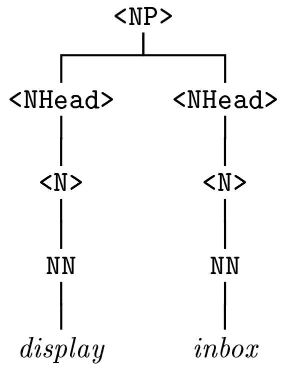


Figure E.2: Syntactic parse of *display inbox*. Note that *display* is tagged as a noun rather than verb. The wrong tagging misleads the learning methods, which construct the rule in LE e.7.5.

LE e.7.6

Trigger utterance: *check for new mail*

Total number of choices: 0

SubRHS learned: None.

Score: 2. Re-use of rule acquired in LE e.7.3.

LE e.7.7

Trigger utterance: *close mail from joseph*

Total number of choices: 0

SubRHS learned: None.

Score: 2. Re-use of rule acquired in LE e.7.4.

E.1.8 User 8

E.1.8.1 Summary

Duration:	12 minutes
Number of utterances:	6
Number of learning episodes:	4
Number of cancelations:	0
Average number of choices:	8.25
Number of rules learned:	8
Average GSG score:	1.13

E.1.8.2 Details

LE e.8.1

Trigger utterance: *can i see my emails*

Total number of choices: 9

SubRHS learned under `<voiceEMail.listMail>`:

`<voiceEMail._ASK_TO_CHECK> <voiceEMail._I> <voiceEMail._LIST>` *my* `<voiceEMail._MAIL_ARGUMENT>*`

Original rule:

```
public <listMail> = [ <_VERB_DESIRE>] (<_LIST> [<_TO_FOR_ME>] |
<_ASK_TO_CHECK>) <_MAIL_ARGUMENT>*;
```

Resulting rule:

```
public <listMail> = [ <_VERB_DESIRE>] (<_LIST> [<_TO_FOR_ME>] |
<_ASK_TO_CHECK> | <_ASK_TO_CHECK> <_I> <_LIST> my <_MAIL_ARGUMENT>*)
<_MAIL_ARGUMENT>*);
```

Score: 2-. An expert grammar writer would have probably placed *can i* under `<_VERB_DESIRE>`, but, being a semantic grammar, the placement under `<_ASK_TO_CHECK>` does make sense too. In fact, it is reused in subsequent utterances (see LE e.8.3 and LE e.8.4) without harmful effects.

SubRHS learned under `<voiceEMail._ASK_TO_CHECK>`:

can

Original rule:

```
<_ASK_TO_CHECK> = do i have | is there [ any | anything] | there is [ any |
anything];
```

Resulting rule:

```
<_ASK_TO_CHECK> = do i have | is there [ any | anything] | there is [ any |
anything] | can;
```

Score: -1. See comment above.

SubRHS learned under `<voiceEMail._LIST>`:

see

Original rule:

```
<voiceEMail._LIST> = list | get | search [for];
```

Resulting rule:

```
<voiceEMail._LIST> = list | get | search [for] | see;
```

Score: 2

LE e.8.2

Trigger utterance: *show me my messages by date*

Total number of choices: 6

SubRHS learned under `<voiceEMail.sortMail>`:

show `<voiceEMail._TO_FOR_ME>` *my* `<voiceEMail._MAIL_ARGUMENT>*` `<voiceEMail._SORT_BY>`

Original rule:

```
public <sortMail> = [<_VERB_DESIRE>] <_SORT> <_MAIL_ARGUMENT>* (
[<_SORT_MODE>] [<_SORT_BY>] | <_SORT_BY> <_SORT_MODE> );
```

Resulting rule:

```
public <sortMail> = [<_VERB_DESIRE>] ( <_SORT> <_MAIL_ARGUMENT>* (
[<_SORT_MODE>] [<_SORT_BY>] | <_SORT_BY> <_SORT_MODE> ) | show <_TO_FOR_ME>
my <_MAIL_ARGUMENT>* <_SORT_BY> );
```

Score: 1. Note that *show* has been added directly under `<sortMail>`; this is because the user rejected the system's initial suggestion that *show* is a way to express `<_SORT>` and later asked for the entire question about *show* to be ignored. Also, *my* has been placed directly under `<sortMail>` too by virtue of being a possessive pronoun. Note that `<_TO_FOR_ME>` could have been horizontally generalized to `[<_TO_FOR_ME>]` if global optionality of `<_TO_FOR_ME>` would have been taken into account.

LE e.8.3

Trigger utterance: *can i draft a new message to bob*

Total number of choices: 8

SubRHS learned under <voiceEMail.replyMail>:

```
<voiceEMail._ASK_TO_CHECK> <voiceEMail._I> <voiceEMail._REPLY>
<voiceEMail._ARTICLE> <voiceEMail._MAIL_ARGUMENT>*
```

Original rule:

```
public <replyMail> = [<_VERB_DESIRE>] <_REPLY> [<_TO_FOR_ME>]
<_MAIL_ARGUMENT>*;
```

Resulting rule:

```
public <replyMail> = [<_VERB_DESIRE>] ( <_REPLY> | <_ASK_TO_CHECK> <_I>
<_REPLY> <_ARTICLE> <_MAIL_ARGUMENT>* ) [<_TO_FOR_ME>] <_MAIL_ARGUMENT>*;
```

Score: 2- - 1 = 1. Re-use of *can* under **<_ASK_TO_CHECK>** discounts 1 point.

SubRHS learned under <voiceEMail._REPLY>:

draft

Original rule:

```
<_REPLY> = reply [to] | answer;
```

Resulting rule:

```
<_REPLY> = reply [to] | answer | draft;
```

Score: 2

LE e.8.4

Trigger utterance: *can i create a new message to bob*

Total number of choices: 10

SubRHS learned under <voiceEMail.composeMail>:

```
<voiceEMail._ASK_TO_CHECK> <voiceEMail._I> <voiceEMail._COMPOSE>
<voiceEMail._ARTICLE> <voiceEMail._MAIL_ARGUMENT>*
```

Original rule:

```
public <voiceEMail.composeMail> = [<_VERB_DESIRE>] <_COMPOSE>
<_MAIL_ARGUMENT>*;
```

Resulting rule:

```
public <voiceEMail.composeMail> = [<_VERB_DESIRE>] ( <_COMPOSE> |
<_ASK_TO_CHECK> <_I> <_COMPOSE> <_ARTICLE> <_MAIL_ARGUMENT>* )
<_MAIL_ARGUMENT>*;
```

Score: 2- - 1 = 1. Re-use of *can* under <_ASK_TO_CHECK> discounts 1 point.

SubRHS learned under voiceEMail._COMPOSE:

create

Original rule:

```
public <_COMPOSE> = compose | write;
```

Resulting rule:

```
public <_COMPOSE> = compose | write | create;
```

Score: 2

E.1.9 User 9

E.1.9.1 Summary

Duration:	18 minutes
Number of utterances:	19
Number of learning episodes:	9
Number of cancelations:	0
Average number of choices:	6.89
Number of rules learned:	8
Average GSG score:	0.92

E.1.9.2 Details

LE e.9.1

Trigger utterance: *check my email*

Total number of choices: 3

SubRHS learned under <voiceEMail.listMail>:

<voiceEMail._ASK_TO_CHECK> *my* <voiceEMail._MAIL_ARGUMENT>*

Original rule:

```
public <listMail> = [ <_VERB_DESIRE>] (<_LIST> [<_TO_FOR_ME>] |
<_ASK_TO_CHECK>) <_MAIL_ARGUMENT>*;
```

Resulting rule:

```
public <listMail> = [ <_VERB_DESIRE>] (<_LIST> [<_TO_FOR_ME>] |
<_ASK_TO_CHECK> | <_ASK_TO_CHECK> my <_MAIL_ARGUMENT>*) <_MAIL_ARGUMENT>*);
```

Score: 2-

SubRHS learned under <voiceEMail._ASK_TO_CHECK>:

check

Original rule:

```
<_ASK_TO_CHECK> = do i have | is there [ any | anything] | there is [ any |
anything];
```

Resulting rule:

```
<_ASK_TO_CHECK> = do i have | is there [ any | anything] | there is [ any |
anything] | check;
```

Score: 2

LE e.9.2

Trigger utterance: *kill spam*

Total number of choices: 10

SubRHS learned under `<voiceEMail._DELETE>`:

kill

Original rule:

`<_DELETE> = delete;`

Resulting rule:

`<_DELETE> = delete | kill;`

Score: 2

SubRHS learned under `<voiceEMail._SUBJECT>`:

`<voiceEMail.subject_STRING>`

Original rule:

`<_SUBJECT> = <_SUBJECT_PRE> [ <_ARTICLE>] <subject_STRING>;`

Resulting rule:

```
<_SUBJECT> = <_SUBJECT_PRE> [ <_ARTICLE>] <subject_STRING> |
<subject_STRING>;
```

Score: -1. Structurally similar situation as in LE e.6.3. In this case `<_SUBJECT>` loses the requirement that it be preceded by `<_SUBJECT_PRE>`, with the consequence that henceforth any word parsed as wildcard can be a `<_SUBJECT>`. However, this proves not to be so disruptive because the wildcard-alone RHS does not percolate up to a top-level NT (i.e., `<_SUBJECT>` can not be parsed alone by any top-level NT). Also, even though the user said that *lists* is a way to express "subject string" (i.e., `<subject_STRING>`) it is not the right place; in fact, this utterance is out of domain because it requires that some messages be flagged as "spam" (unsolicited e-mail). However, see next LE e.9.3.

SubRHS learned under `<voiceEMail.subject_STRING>`:

spam

Original rule:

`<subject_STRING> = "<subject>" | <_WILDCARD>+;`

Resulting rule:

`<subject_STRING> = "<subject>" | <_WILDCARD>+ | spam;`

Score: -1. See comment above.

LE e.9.3

Trigger utterance: *kill spam means delete mail from spamela*

Total number of choices: 0

SubRHS learned: None.

Score: 1. Learned meaning equivalence represented in Figure E.3.

LE e.9.4

Trigger utterance: *check mail*

Total number of choices: 0

SubRHS learned: None.

Score: 2. Re-use of rule acquired in LE e.9.1

LE e.9.5

Trigger utterance: *newest first*

Total number of choices: 10

SubRHS learned: None.

Score: 1. The learning episode constructs the parse tree in Figure E.4 which gives rise to the sub RHS `<_MAIL_ARGUMENT>* <_ORDINAL_NUMBER_0_99>` under `<sortMail>`, but it is found that it introduces ambiguity (since, for instance, `<_ORDINAL_NUMBER_0_99>` is parsed under `<dayOfMonth__INTEGER`), so the learning epiosode is canceled. Note that this is safer than LE e.6.3. In any case, the next utterance from the user was *newest first means sort by date* (see LE e.9.6).

LE e.9.6

Trigger utterance: *newest first means sort by date*

Total number of choices: 0

SubRHS learned: None.

Score: 1. Learned meaning equivalence represented in Figure E.5.

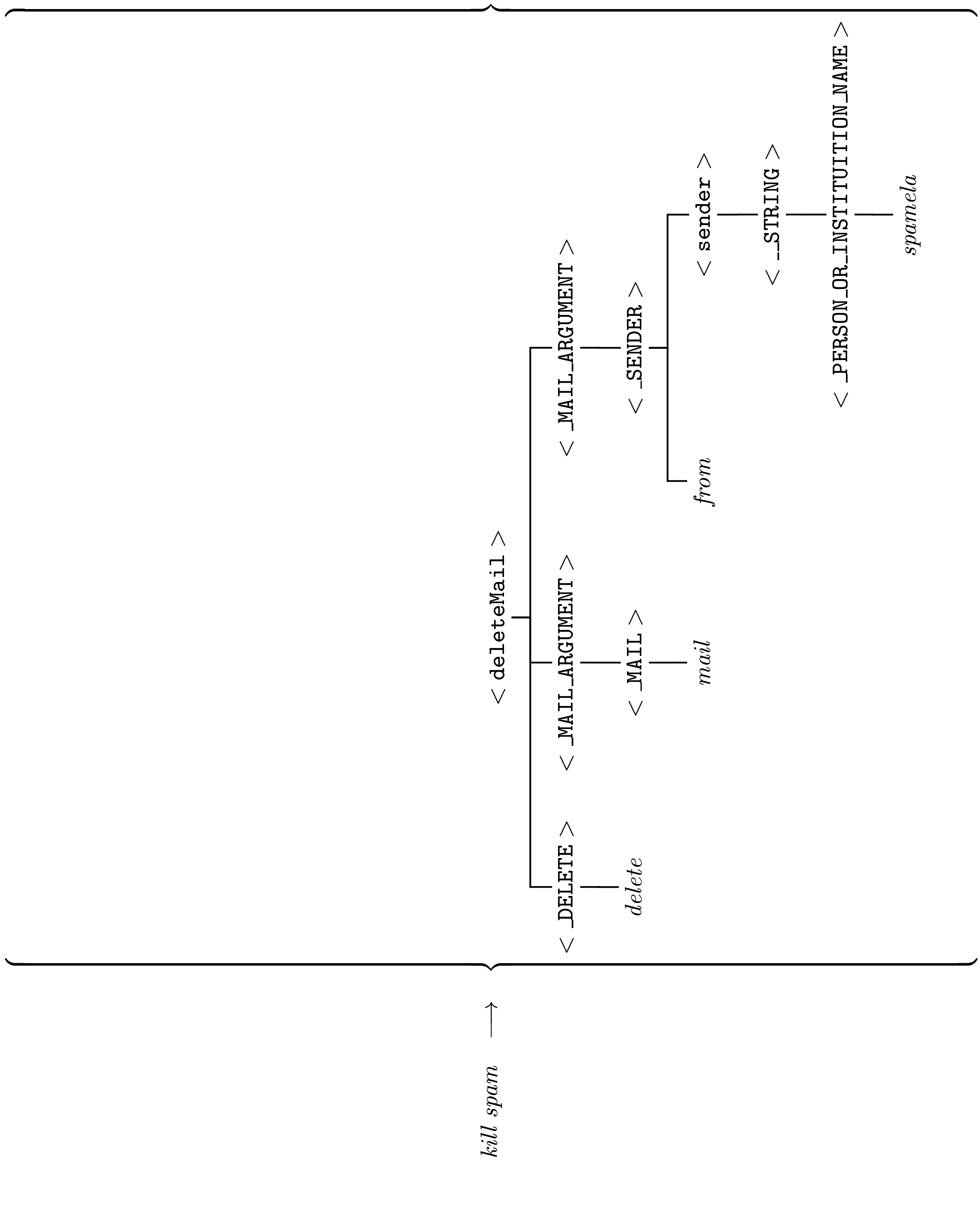


Figure E.3: Mapping learned for *kill spam* after LE e.9.3.

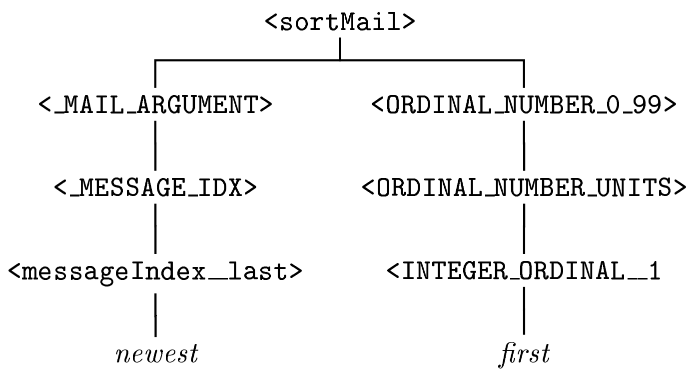


Figure E.4: Parse tree for *newest first* as constructed by the learning methods in LE e.9.5. No rules are learned from it as they would introduce ambiguity.

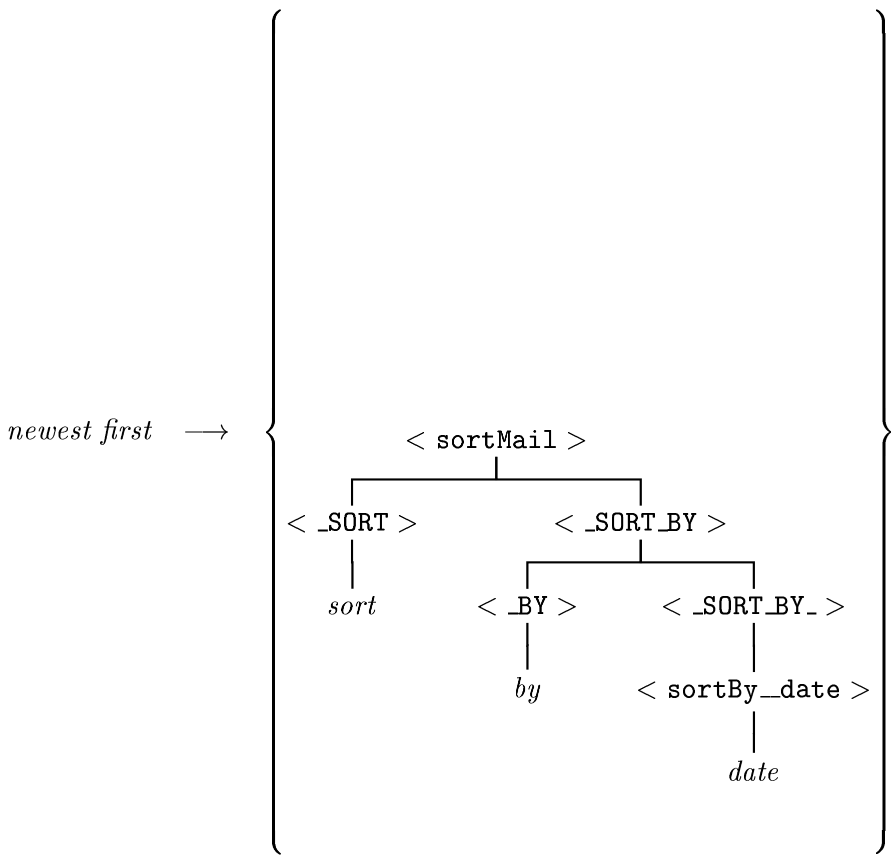


Figure E.5: Mapping learned for *newest first* after LE e.9.6.

LE e.9.7

Trigger utterance: *check spelling*

Total number of choices: 10

SubRHS learned: None.

Score: 1. Out of domain.

LE e.9.8

Trigger utterance: *put lists first*

Total number of choices: 20

SubRHS learned under `<voiceEMail.sortMail>`:

`<voiceEMail._SORT> <voiceEMail.sortBy_sender> <voiceEMail._MAIL_ARGUMENT>+`

Original rule:

```
public <sortMail> = [<_VERB_DESIRE>] <_SORT> <_MAIL_ARGUMENT>* (
[<_SORT_MODE>] [<_SORT_BY>] | <_SORT_BY> <_SORT_MODE> );
```

Resulting rule:

```
public <sortMail> = [<_VERB_DESIRE>] (<_SORT> <_MAIL_ARGUMENT>* (
[<_SORT_MODE>] [<_SORT_BY>] | <_SORT_BY> <_SORT_MODE> ) | <_SORT>
<sortBy_sender> <_MAIL_ARGUMENT>+);
```

Score: 1-. The user's intention is in fact out of domain because it requires that some messages be flagged as coming from a mailing list server rather than a single person.

SubRHS learned under `<voiceEMail._SORT>`:

put

Original rule:

```
<_SORT> = sort;
```

Resulting rule:

```
<_SORT> = sort | put;
```

Score: 2

SubRHS learned under `<voiceEMail.sortBy_sender>`:

lists

Original rule:

```
<sortBy_sender> = sender | origin | source;
```

Resulting rule:

```
<sortBy_sender> = sender | origin | source | lists;
```

Score: -1. Concept out of domain — see note above.

LE e.9.9

Trigger utterance: *block mail from lucy*

Total number of choices: 9

SubRHS learned under `<voiceEMail._DELETE>`:

block

Original rule:

```
<_DELETE> = delete | kill;
```

Resulting rule:

```
<_DELETE> = delete | kill | block;
```

Score: 2

E.1.10 User 10

E.1.10.1 Summary

Duration:	9 minutes
Number of utterances:	14
Number of learning episodes:	7
Number of cancelations:	0
Average number of choices:	3.57
Number of rules learned:	5
Average GSG score:	1.50

E.1.10.2 Details

LE e.10.1

Trigger utterance: *show me all mail from donald*

Total number of choices: 3

SubRHS learned under `<voiceEMail._LIST>`:

show

Original rule:

```
<_LIST> = list | get | search [for];
```

Resulting rule:

```
<_LIST> = list | get | search [for] | show;
```

Score: 2

LE e.10.2

Trigger utterance: *show me my mail*

Total number of choices: 0

SubRHS learned: None.

Score: 2. Re-use of rule acquired in LE e.10.1.

LE e.10.3

Trigger utterance: *i want to send an email to bob*

Total number of choices: 5

SubRHS learned: None.

Score: 1. Conflict detected. See interaction e.5.

LE e.10.4

Trigger utterance: *get rid of all messages from spamela*

Total number of choices: 5

SubRHS learned: None.

Score: 0. User volunteered information: see below.

LE e.10.5

Trigger utterance: *get rid of means delete*

Total number of choices: 1

SubRHS learned under `<voiceEMail._DELETE>`:

get rid of

Original rule:

```
<_DELETE> = delete;
```

Resulting rule:

```
<_DELETE> = delete | get rid of;
```

Score: 2. See interaction e.2.

LE e.10.6

Trigger utterance: *ignore message from lucy*

Total number of choices: 9

SubRHS learned under <voiceEMail._DELETE>:

ignore

Original rule:

<_DELETE> = delete | get rid of;

Resulting rule:

<_DELETE> = delete | get rid of | ignore;

Score: 2.

LE e.10.7

Trigger utterance: *respond to goku's email*

Total number of choices: 2

SubRHS learned under <voiceEMail.replyMail>:

<voiceEMail._REPLY> <voiceEMail._DATE_RANGE_IN> <voiceEMail._MAIL_ARGUMENT>*

Original rule:

public <replyMail> = [<_VERB_DESIRE>] <_REPLY> [<_TO_FOR_ME>] <_MAIL_ARGUMENT>*;

Resulting rule:

public <replyMail> = [<_VERB_DESIRE>] (<_REPLY> | <_REPLY> <_DATE_RANGE_IN> <_MAIL_ARGUMENT>*) [<_TO_FOR_ME>] <_MAIL_ARGUMENT>*;

Score: 2-. Harmless generalization of *to* to <_DATE_RANGE_IN> (same as in LE e.6.5), but it would have been better to place the *to* under <_REPLY> (see below).

SubRHS learned under voiceEMail._REPLY:

respond

Original rule:

<_REPLY> = reply [to] | answer;

Resulting rule:

<_REPLY> = reply [to] | answer | respond;

Score: 1. An expert grammar writer would have generalized to <_REPLY> = (reply | respond) [to] | answer;.

E.2 Musicbox Task's Detailed Results

E.2.1 User 1

E.2.1.1 Summary

Duration:	11 minutes
Number of utterances:	18
Number of learning episodes:	8
Number of cancelations:	0
Average number of choices:	3.13
Number of rules learned:	4
Average GSG score:	1.88

E.2.1.2 Details

LE m.1.1

Trigger utterance: *ff*

Total number of choices: 6

SubRHS learned under `<musicbox.fastForward>`:

ff

Original rule:

```
public <fastForward> = [<_POLITE_WORD>] [fast] forward [<_POLITE_WORD>];
```

Resulting rule:

```
public <fastForward> = [<_POLITE_WORD>] [fast] ( forward | ff )
[<_POLITE_WORD>];
```

Score: 2+

LE m.1.2

Trigger utterance: *fb means fast backward*

Total number of choices: 1

SubRHS learned under `<musicbox.fastBackward>`:

fb

Original rule:

```
public <fastBackward> = [<_POLITE_WORD>] [fast] backward [<_POLITE_WORD>];
```

Resulting rule:

```
public <fastBackward> = [<_POLITE_WORD>] [fast] ( backward | fb )
[<_POLITE_WORD>];
```

Score: 2+

LE m.1.3

Trigger utterance: *ff*

Total number of choices: 0

SubRHS learned: None.

Score: 2. Re-use of rule acquired in m.1.1.

LE m.1.4

Trigger utterance: *fb*

Total number of choices: 0

SubRHS learned: None.

Score: 2. Re-use of rule acquired in m.1.2.

LE m.1.5

Trigger utterance: *ff*

Total number of choices: 0

SubRHS learned: None.

Score: 2. Re-use of rule acquired in m.1.1.

LE m.1.6

Trigger utterance: *toss thunder road into cart*

Total number of choices: 3

SubRHS learned under `<shoppingcart.addToCart>`:

toss

Original rule:

```
public <addToCart> = add | buy;
```

Resulting rule:

```
public <addToCart> = add | buy | toss;
```

Score: 2

LE m.1.7

Trigger utterance: *drop thunder road from cart*

Total number of choices: 14

SubRHS learned under `<shoppingcart.removeFromCart>`:

drop

Original rule:

```
public <removeFromCart> = remove;
```

Resulting rule:

```
public <removeFromCart> = remove | drop;
```

Score: 2

Interaction m.1.8

```
> play another song by springsteen
I understand "play song"
> wrong
"play another song by springsteen" is a way to express:
1.  play, e.g.  "i want hear"
2.  object, e.g.  "great"
3.  recording artist name, e.g.  "tom waits"
0.  None of the above
> 1
Sorry, I can't learn what you mean as it is not supported by the end
application.
```

LE m.1.8

Trigger utterance: *play another song by springsteen*

Total number of choices: 1

SubRHS learned: None.

Score: 1. Beyond the capabilities of the application.

LE m.1.9

Trigger utterance: *shut up*

Total number of choices: 7

SubRHS learned under `<musicbox.stop`:

shut up

Original rule:

```
public <stop> = [<_POLITE_WORD>] ( stop | halt ) [<_POLITE_WORD>];
```

Resulting rule:

```
public <stop> = [<_POLITE_WORD>] ( stop | halt | shut up) [<_POLITE_WORD>];
```

Score: 2+. Note that henceforth the system understands *please shut up please* (should anyone say that!)

E.2.2 User 2

E.2.2.1 Summary

Duration:	18 minutes
Number of utterances:	24
Number of learning episodes:	7
Number of cancelations:	1
Average number of choices:	3.29
Number of rules learned:	5
Average GSG score:	1.00

E.2.2.2 Details

Cancelation m.2.1

```
> how much is it
I don't understand right away what you mean but let me guess...
"how much is it" is a way to express:
1. balance, e.g. "balance"
2. track title, e.g. "my all"
3. recording notes, e.g. "rolling stone on what is something of a
transit"
0. None of the above
> 0
Is "how much is it" a way to express get information, e.g. "list
information"?
1. Yes
2. No
> cancel
Ok, back to square one
```

Comment: User probably realized there was no direct way of obtaining price, but see LE m.2.2.

LE m.2.1

Trigger utterance: *what's my total*

Total number of choices: 1

SubRHS learned under `<shoppingcart.balance>`:

what is my total

Original rule:

```
public <balance> = [what] [is] [the | my] balance | how much is that;
```

Resulting rule:

```
public <balance> = [what] [is] [the | my] balance | how much is that |
what is my total;
```

Score: 1. An expert grammar writer would have generalized to `public <balance> = [what] [is] [the | my] [balance | total] | how much is that;`.

LE m.2.2

Trigger utterance: *how much is lullaby*

Total number of choices: 1

SubRHS learned under `<shoppingcart.addToCart>`:

how much is

Original rule:

```
public <addToCart> = add | buy;
```

Resulting rule:

```
public <addToCart> = add | buy | how much is;
```

Score: -1. See interaction m.2

LE m.2.3

Trigger utterance: *what's my total*

Total number of choices: 0

SubRHS learned: None.

Score: 2. Re-use of rule acquired in m.2.1

LE m.2.4

Trigger utterance: *what's my total*

Total number of choices: 0

SubRHS learned: None.

Score: 2. Re-use of rule acquired in m.2.1

LE m.2.5

Trigger utterance: *how much do i owe*

Total number of choices: 1

SubRHS learned under `<shoppingcart.balance>`:

how much do i owe

Original rule:

```
public <balance> = [what] [is] [the | my] balance | how much is that;
```

Resulting rule:

```
public <balance> = [what] [is] [the | my] balance | how much is that | how
much do i owe;
```

Score: 2

LE m.2.6

Trigger utterance: *what other music do you have*

Total number of choices: 6

SubRHS learned under `<musicbox.listAllSongs>`:

what other music `<musicbox._ARE_AVAILABLE>`

Original rule:

```
public <listAllSongs> = ( what | which | whose ) [<_SONG>] <_ARE_AVAILABLE>
"</s>" | [please] <_SHOW> [me] all [<_AVAILABLE>] <_SONG> "</s>";
```

Resulting rule:

```
public <listAllSongs> = ( what | which | whose ) [<_SONG>] <_ARE_AVAILABLE>
"</s>" | [please] <_SHOW> [me] all [<_AVAILABLE>] <_SONG> "</s>" | what
other music <_ARE_AVAILABLE>;
```

Score: 1. An expert grammar writer would have probably inserted `[ other ]` between `( what | which | whose)` and `[ <_SONG>]` and placed *music* under `<_SONG>`.

LE m.2.7

Trigger utterance: *i'm ready to pay*

Total number of choices: 14

SubRHS learned under `<shoppingcart.checkout>`:

i am ready to pay

Original rule:

```
public <checkout> = [please] checkout [please] | [okay] i am done;
```

Resulting rule:

```
public <checkout> = [please] checkout [please] | [okay] i am done | i am
ready to pay;
```

Score: 1. An expert grammar writer would have probably generalized to `public <checkout> = [please] checkout [please] | [okay] i am [done | ready to pay];`. Note that *i* has not been generalized to `<musicbox._I>` because `<musicbox._I>` belongs to a different grammar that is never referenced (imported) from `shoppingcart`.

E.2.3 User 3

E.2.3.1 Summary

Duration:	12 minutes
Number of utterances:	13
Number of learning episodes:	8
Number of cancelations:	0
Average number of choices:	4.25
Number of rules learned:	5
Average GSG score:	1.50

E.2.3.2 Details

LE m.3.1

Trigger utterance: *put song in my shopping basket*

Total number of choices: 1

SubRHS learned: None.

Score: -1. System responded with a "not supported by end-application" message because *put* and *in my shopping basket* were two non-contiguous unparsed segments that surround an ontologically top-level segment. In this case the answer was not appropriate because the intention is supported by the end-application. But see next learning episode (LE m.3.2).

LE m.3.2

Trigger utterance: *shopping basket means shopping cart*

Total number of choices: 1

SubRHS learned under `<shoppingcart.toCart>`:

[*shopping*] *basket*

Original rule:

```
public <toCart> = [in | to | into] [my | the] [shopping] cart;
```

Resulting rule:

```
public <toCart> = [in | to | into] [my | the] [shopping] ( cart |
[shopping] basket );
```

Score: 2-. An expert grammar writer would have written a more compact `public <toCart> = [in | to | into] [my | the] [shopping] ( cart | basket )` but GSG does correctly make the *shopping* in *shopping basket* optional.

LE m.3.3

Trigger utterance: *put song in my shopping basket*

Total number of choices: 3

SubRHS learned under `<shoppingcart.addToCart>`:

put

Original rule:

```
public <addToCart> = add | buy;
```

Resulting rule:

```
public <addToCart> = add | buy | put;
```

Score: 2

LE m.3.4

Trigger utterance: *put three little birds in my basket*

Total number of choices: 0

SubRHS learned: None.

Score: 2. Re-use of rules acquired in m.3.2 and m.3.3.

LE m.3.5

Trigger utterance: *put no woman no cry into basket*

Total number of choices: 0

SubRHS learned: None.

Score: 2. Re-use of rules acquired in m.3.2 and m.3.3.

LE m.3.6

Trigger utterance: *quiet please*

Total number of choices: 11

SubRHS learned under `<musicbox.mute>`:

quiet `[<_POLITE_WORD>]`

Original rule:

```
public <mute> = [<_POLITE_WORD>] mute [<_POLITE_WORD>];
```

Resulting rule:

```
public <mute> = [<_POLITE_WORD>] ( mute | quiet [<_POLITE_WORD>] )
[<_POLITE_WORD>];
```

Score: 2-.

LE m.3.7

Trigger utterance: *what's my current total*

Total number of choices: 1

SubRHS learned under `<shoppingcart.balance>`:

what is my current total

Original rule:

```
public <balance> = [what] [is] [the | my] balance | how much is that;
```

Resulting rule:

```
public <balance> = [what] [is] [the | my] balance | how much is that |
what is my current total;
```

Score: 1. An expert grammar writter would have written `public <balance> = [what] [is] [the | my] [current] (balance | toal) | how much is that;` but that requires external knowledge that *current* is in this case optional and that *total* is in this case synonym of *balance*.

LE m.3.8

Trigger utterance: *time to pay up*

Total number of choices: 17

SubRHS learned under `<shoppingcart.checkout>`:

time to pay up

Original rule:

```
public <checkout> = [please] checkout [please] | [okay] i am done;
```

Resulting rule:

```
public <checkout> = [please] checkout [please] | [okay] i am done | time
to pay up;
```

Score: 2. Correct rule but user had to go through 17 choices! See Future Directions ***: more dialogue-state modeling, e.g., likelihood of `checkout` should increase with time and with number of songs in cart.

E.2.4 User 4

E.2.4.1 Summary

Duration:	15 minutes
Number of utterances:	22
Number of learning episodes:	4
Number of cancelations:	0
Average number of choices:	4.00
Number of rules learned:	2
Average GSG score:	1.40

E.2.4.2 Details

LE m.4.1

Trigger utterance: *search for songs by title*

Total number of choices: 3

SubRHS learned: None.

Score: 1. GSG detects impossibility to learn as the unparsed material surrounds a segment parsed by an ontologically top-level root (*songs*). The response here is appropriate as the end-application would not support the resulting feature structure with an embedded `<musicbox.object>`.

LE m.4.2

Trigger utterance: *find songs played by they might be giants*

Total number of choices: 4

SubRHS learned: None.

Score: 1. Same case as m.4.1.

LE m.4.3

Trigger utterance: *how much have i spents so far* (sic)

Total number of choices: 1

SubRHS learned under `<shoppingcart.balance>`:

how much have i spents so far

Original rule:

```
public <balance> = [what] [is] [the | my] balance | how much is that;
```

Resulting rule:

```
public <balance> = [what] [is] [the | my] balance | how much is that | how
much have i spents so far;
```

Score: 2-

LE m.4.4

Trigger utterance: *check out*

Total number of choices: 12

SubRHS learned under `<shoppingCart.checkout>`:

check out

Original rule:

```
public <checkout> = [please] checkout [please] | [okay] i am done;
```

Resulting rule:

```
public <checkout> = [please] checkout [please] | [okay] i am done | check
out;
```

Score: 1. Note that an expert grammar writer would have generalized to `[please] (checkout | check out) [please]`. At the same time, GSG would have generalized to exactly that if the initial `[please]` in the original rule had scoped over all the alternatives.

LE m.4.5

Trigger utterance: *check out means checkout*

Total number of choices: 0

SubRHS learned: None.

Score: 2. For some reason, user explicitly restated meaning equivalence even after successful LE m.4.4.

E.2.5 User 5

E.2.5.1 Summary

Duration:	14 minutes
Number of utterances:	20
Number of learning episodes:	4
Number of cancelations:	0
Average number of choices:	7.00
Number of rules learned:	3
Average GSG score:	1.75

E.2.5.2 Details

LE m.5.1

Trigger utterance: *show me all the songs*

Total number of choices: 1

SubRHS learned under `<musicbox.getInformation>`:

`<musicbox._SHOW>` *me* `<musicbox._quantity>`

Original rule:

```
public <getInformation> = ( list | get | give ) [me] [some] ( information
| info ) [on | about] | [<_I>] <_VERB_DESIRE> know [on | about] | ( what |
which | whose ) [<_SONG>] <_ARE_AVAILABLE> ( in <language> | <_ARTIST> ) |
( do you have | is there ) ( anything | any thing | [a | any] <_SONG> ) |
[please] <_SHOW> [me] [<_AVAILABLE>] <_SONG>;
```

Resulting rule:

```
public <getInformation> = ( list | get | give ) [me] [some] ( information
| info ) [on | about] | [<_I>] <_VERB_DESIRE> know [on | about] | ( what |
which | whose ) [<_SONG>] <_ARE_AVAILABLE> ( in <language> | <_ARTIST> ) |
( do you have | is there ) ( anything | any thing | [a | any] <_SONG> ) |
[please] <_SHOW> [me] [<_AVAILABLE>] <_SONG> | <_SHOW> me <_quantity>;
```

Score: 2-

LE m.5.2

Trigger utterance: *repeat caro mio ben*

Total number of choices: 9

SubRHS learned: None.

Score: 1. Choice “rewind” was presented but was not liked by user. But see LE m.5.4.

LE m.5.3

Trigger utterance: *lower the volume*

Total number of choices: 12 (9) + 1 (Choice 9th was correct but was overseen by the user, who, at choice 12th decided to rephrase: *lower the volume means softer* after which only a confirmation question had to be answered.)

SubRHS learned under `musicbox.softer`:

lower `<musicbox._quantity>` *volume*

Original rule:

```
public <musicbox.softer> = [<_POLITE_WORD>] [play] softer [<_POLITE_WORD>];
```

Resulting rule:

```
public <musicbox.softer> = [<_POLITE_WORD>] [play] ( softer | lower
<_quantity> volume) [<_POLITE_WORD>];
```

Score: 2. Expert grammar writer could have hardly done better — overgeneralization of *the* as `musicbox._quantity` is harmless.

LE m.5.4

Trigger utterance: *repeat caro mio ben*

Total number of choices: 5

SubRHS learned under `musicbox.rewind`:

repeat

Original rule:

```
public <musicbox.rewind> = [<_POLITE_WORD>] rewind [<_POLITE_WORD>];
```

Resulting rule:

```
public <musicbox.rewind> = [<_POLITE_WORD>] ( rewind | repeat)
[<_POLITE_WORD>];
```

Score: 2+. User eventually mapped her idea of repeating to the application's `<rewind>`.

References

Allen, James (1995). *Natural Language Understanding*, 2nd ed., Benjamin/Cummings.

Allen, James, et al. (1996). Robust Understanding in a Dialogue System. In *Proceedings of ACL-1996.*

Baker, C.L. and J.J. McCarthy (eds.) (1981). *The logical problem of language acquisition.* The MIT Press.

Bloom, P. (ed.) (1994). *Language acquisition — Core readings.* The MIT Press.

Brent, M.R. (1997). *Computational approaches to language acquisition.* The MIT Press.

Brill, Eric (1994). Some Advances in Part of Speech Tagging. In *Proceedings of AAAI-1994.*

Carbonell, Jaime G. (1979). Towards a self-extending parser. In *Proceedings of ACL-79.*

Carpenter, Bob (1998). *Type-Logical Semantics.* The MIT Press.

Chomsky, Noam (1956). Three models for the description of language. In *IRE Trans. Info. Theory*, 2(3), 113–124.

Chomsky, Noam (1959). On certain formal properties of grammars. In *Information and Control*, 2 (3), 137–167.

Chu-Carroll, Jennifer and Bob Carpenter (1999). Vector-Based Natural Language Call Routing. In *Computational Linguistics*, 25(3), pages 361–388.

Culy, Christopher (1985). The Complexity of the vocabulary of Bambara. In *Linguistics and Philosophy*, 8, pages 345–351.

Dalrymple, Mary (1999). Lexical-Functional Grammar. In *MIT Encyclopedia of the Cognitive Sciences*, edited by Rob Wilson and Frank Keil. The MIT Press.

Denecke, Matthias (1997). An Information-based Approach for Guiding Multi-modal Human-Computer Interaction. In *Proceedings of IJCAI-1997.*

Dennett, Daniel C. (1984). Can machines think? In *How We Know*, edited by M. Shafto. Harper & Row.

Fellbaum, Christiane (ed.) (1998). *WordNet: An Electronic Lexical Database.* The MIT Press.

Glass, James (1999). Challenges for Spoken Dialogue Systems. In *Proceedings of the 1999 IEEE ASRU Workshop.*

Gertner, A.N. and A.L. Gorin (1993). Adaptive language acquisition for an airline information subsystem. In *Artificial Neural Networks for Speech and Vision*, edited by R. Mammone.

Goodman, R.M., C.M. Higgins, J.W. Miller, and P. Smyth (1992). Rule-based neural network for classification and probability estimation. In *Neural Comput.*

Granger, R.H. (1977). FOUL-UP: A program that figures out meanings of words from context. In *Proceedings of the Fifth International Joint Conference on Artificial Intelligence (IJCAI-77).*

Gorin, Allen L. (1995). On automated language acquisition. In *Journal of the Acoustical Society of America*, June 1995.

Grune, Dick, and Ceriel J.H. Jacobs (1990). *Parsing Techniques: A Practical Guide.* Ellis Horwood.

Hall, P. and G. Dowling (1980). Approximate string matching. In *ACM Computing Surveys*, vol. 12, pp. 381–402.

Jelinek, Frederick (1997). *Statistical Methods for Speech Recognition.* The MIT Press.

Java^{TM} Speech API, version 1.0 (1998). `http://java.sun.com/products/java-media/speech`.

Java^{TM} Speech Grammar Format, version 1.0 (1998). `http://java.sun.com/products/java-media/speech/forDevelopers/JSGF`.

Jurafsky, Dan, and James H. Martin (2000). *Speech and Language Processing.* Prentice Hall.

T. Kam, T. Villa, R. Brayton, and A. Sangiovanni-Vincentelli (1994). A Fully Implicit Algorithm of Exact-state Minimization. In *Proc. IEEE Design Automation Conf.*, pp. 684–690.

Kiefer, Berrnd and Hans-Ulrich Krieger (1998). A Bag of Useful Techniques for Efficient and Robust Parsing. DFKI Research Report 98-04.

Kiyono, M. and J. Tsujii (1993). Linguistic knowledge acquisition from parsing failures. In *Proceedings of the Sixth Conference of the European Chapter of the Association for Computational Linguistics.*

Lavie, Alon (1996). *GLR*: A Robust Grammar-Focused Parser for Spontaneously Spoken Language.* Ph.D. dissertation. School of Computer Science, Carnegie Mellon University.

Lehman, Jill, and Jaime Carbonell (1989). Learning the user's language: A step towards automated creation of user models. In Wahlster, W. and A. Kobska (eds), *User Modelling in Dialog Systems.*

Lehman, Jill (1989). *Adaptive Parsing: Self-extending Natural Language Interfaces.* Ph.D. dissertation, School of Computer Science, Carnegie Mellon University.

Manning, Cristopher D. and Hinrich Schütze (1999). *Foundations of Statistical Natural Language Processing.* The MIT Press.

Mayfield, Laura, Marsal Gavaldà, Wayne Ward and Alex Waibel (1995). Concept-Based Speech Translation. In *Proceedings of ICASSP-1995.*

Mayfield, L., M. Gavaldà, Y.-H. Seo, B.Suhm, W. Ward, and A. Waibel (1995). Parsing real input in JANUS: A concept-based approach. In *Proceedings of TMI-95.*

Miller, S., R. Bobrow, R. Ingria, and R. Schwartz (1994). Hidden understanding models of natural language. In *Proceedings of ACL-94.*

Miller, S., M. Bates, R. Bobrow, R. Ingria, J. Makhoul, and R. Schwartz (1995). Recent progress in hidden understanding models. In *Proceedings of the spoken language systems technology workshop*, ARPA, Morgan Kaufmann.

Mitchell, Tom. (1997). *Machine Learning.* McGraw Hill.

Miller, Wolfgang (1997). Ph.D. dissertation. Universität Karlsruhe.

Morrill, Glyn (1994). *Type Logical Grammar: Categorial Logic of Signs.* Kluwer Academic Publishers.

Napoli, Donna Jo (1993). *Syntax: Theory and Problems.* Oxford University Press.

Neidle, Carol (1994). Lexical-Functional Grammar. In *Encyclopedia of Language and Linguistics.* Pergamon Press.

Newell, A. and H. A. Simon (1972). *Human Problem Solving.* Prenctice-Hall.

Nigam, Kamal, Andrew McCallum, Sebastian Thrun and Tom Mitchell (2000). Text Classification from Labeled and Unlabeled Documents using EM. To appear in *Machine Learning.*

Nuance^{TM} (2000). `http://www.nuance.com/index.htma?SCREEN=speechobjects`.

Parekh, R. and V. Honavar (2000). Automata Induction, Grammar Inference, and Language Acquisition. In *Handbook of Natural Language Processing.* Dale, Moisl and Somers (eds). Marcel Dekker.

Perlmutter, David M. (1983). *Studies in Relational Grammar I.* University of Chicago Press.

Perlmutter, David M. and C. Rosen (eds.) (1984). *Studies in Relational Grammar II.* University of Chicago Press.

Pieraccini, R. and E. Levin (1993). A Learning Approach to Natural Language Understanding. In *New advances and trends in speececognition and coding*, NATO ASI Series, Springer-Verlag.

Pollard, Carl, and Ivan A. Sag (1994) *Head-Driven Phrase Structure Grammar.* The University of Chicago Press.

Rabiner, L. (1989). A tutorial on hidden Markov models and selected applications in speech recognition. In *Proceedings of the IEEE*, 77(2), pp. 257–286.

Manny Rayner and David Carter (1996). Fast Parsing using Pruning and Grammar Specialization. In *Proceedings of ACL-1996.*

Rosé, Carolyn Penstein (1997). *Robust Interactive Dialogue Interpretation.* Ph.D. dissertation. Language Technologies Institute, School of Computer Science, Carnegie Mellon University.

Rosé, Carolyn P. and Alon Lavie (1999). LCFLEX*: An Efficient Robust Left-Corner Parser.* University of Pittsburgh, Learning Research and Development Center, LRDC Technical Report LRDC-NLU-1.

Rudnicky, Alex, et al. (1999). Creating Natural Dialogs in the Carnegie Mellon COMMUNICATOR System. In *Proceedings of Eurospeech-1999.*

Sankar, A. and A.L. Gorin (1993). Adaptive language acquisition in a multisensory device. In *Artificial neural networks for speech and vision*, edited by R. Mammone.

Seneff, Stephanie (1992). TINA: a natural language system for spoken language applications. In *Computational Linguistics*, 18(1), pp. 61–83.

Shieber, Stuart (1985). Evidence Against the Context-freeness of Natural Language. In *Linguistics and Philosophy*, 8, pp. 333–343.

Sleator, Daniel and Davy Temperley (1993). Parsing English with a Link Grammar. In *Proceedings of the Third International Workshop on Parsing Technologies.*

Jonathan Slocum, Jonathan (1981). A Practical Comparison of Parsing Strategies. In *Proceedings of ACL-1981.*

Tellme^{TM} (2000). `http://studio.tellme.com/library/grammar/`.

Tishby, N.Z. and A.L. Gorin (1994). Algebraic learning of statistical associations. In *Computer, Speech & Language.*

Tomita, Masaru (1987). An Efficient Augmented-Context-Free Parsing Algorithm. In *Computational Linguistics*, 13(1-2), pp. 31–46.

Tomita, Masaru, and Harry Bunt (eds.) (1996). *Recent Advances in Parsing Technology.* Kluwer Academic Publishers.

Turing, Alan (1950). Computing machinery and intelligence. In *Mind*, 59, pp. 433–460.

Universität des Saarlandes (1994). The Verbmobil Semantic Specification. Verbmobil Report 1994-6.

Waibel, Alex, Michael Finke, Donna Gates, Marsal Gavaldà, Thomas Kemp, Alon Lavie, Lori Levin, Martin Maier, Laura Mayfield, Arthur McNair, Ivica Rogina, Kaori Shima, Tilo Sloboda, Monika Woszczyna, Torsten Zeppenfeld, and Puming Zhan (1996). JANUS-II: Translation of Spontaneous Conversational Speech. In *Proceedings of ICASSP-1996.*

Waibel, Alex. (1996) Interactive Translation of Conversational Speech. In *IEEE Computer*, 29(7).

Walker, Marilyn, et al. (1998). Learning Optimal Dialogue Strategies: A Case Study of a Spoken Dialogue Agent for Email. In *Proceedigns of COLING/ACL-1998.*

Ward, Wayne (1994). Extracting information in spontaneous speech. In *International Conference for Spoken Language Processing*, Yokohama, Japan.

Woszczyna, Monika, Matthew Broadhead, Donna Gates, Marsal Gavaldà, Alon Lavie, Lori Levin, and Alex Waibel (1998). A Modular Approach to Spoken Language Translation for Large Domain. In *Proceedings of AMTA-1998.*

Zue, Victor, et al. (2000). JUPITER: A Telephone-Based Conversational Interface for Weather Information. In *IEEE Transactions on Speech and Audio Processing*, 8(1).