Genesis-II:
A Language Generation Module for Conversational Systems

by

Lauren M. Baptist

B.A., Dartmouth College (1999)

Submitted to the Department of Electrical Engineering and Computer Science
in Partial Fulfillment of the Requirements for the Degree of

Master of Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

September 2000

Author . . . .
Department of Electrical Engineering and Computer Science
August 28, 2000

Certified by . . . .
Stephanie Seneff
Principal Research Scientist
Thesis Supervisor

Accepted by . . . .
Arthur C. Smith
Chairman, Departmental Committee on Graduate Students

Genesis-II:
A Language Generation Module for Conversational Systems

by

Lauren M. Baptist

Submitted to the Department of Electrical Engineering and Computer Science
on August 28, 2000, in Partial Fulfillment of the
Requirements for the Degree of
Master of Science

Abstract

Language generation is a fundamental component of dialogue systems. Over the past year, we have developed a new generation module for conversational systems developed at MIT using the Galaxy architecture. In this thesis, we discuss how our system, Genesis-II, bridges non-linguistic and linguistic generation with an original framework that is simple, yet powerful. We have created a set of mechanisms that allow domain experts to rapidly develop template-like rules for simple domains, while giving them the power to carefully construct linguistic rules for complex domains. In particular, Genesis-II provides domain experts with flexible tools for overcoming difficult linguistic challenges, like word-sense disambiguation and wh-movement. Throughout the thesis, we shall illustrate the Genesis-II framework with examples from many domains (flight status, weather information) and languages, both natural (English, Chinese, Japanese, Spanish) and formal (SQL, HTML, speech waveforms).

Thesis Supervisor: Stephanie Seneff
Title: Principal Research Scientist

Acknowledgments

First, I would like to thank my thesis advisor, Stephanie Seneff, for all of the help she has given me in the past year. I simply couldn't have done it without her guidance and insight.

I would like to acknowledge the members of the Spoken Language Systems group for all of their help. I am especially grateful to our domain experts, some of whom began creating knowledge bases within Genesis-II when the system was still in its earliest stages of development. Their feedback was invaluable to me. In particular, I would like to thank Alcira Gonzalez, Mikio Nakano, Joe Polifroni, and Chao Wang.

I am grateful to my mother and father for encouraging me from afar, and I am grateful to my friends here—Abigail Marsh, Marty Vona, April Rasala, and Trusty—for making this year at MIT a memorable one. I appreciate the continued support of my undergraduate advisor, Tom Cormen. Finally, I would like to thank Ross Wilken, for his endless patience—through this thesis and the last.

This work was supported by a fellowship from Nippon Telegraph & Telephone.

Contents

List of Figures

Chapter 1

Introduction

Language generation is the task of automatically converting an abstract meaning representation into a string. A simple example would be the conversion of the meaning representation, {`$destination_city = Vienna, $departure_time = noon`}, into the string, `"The flight to Vienna leaves at noon."` Meaning representations range in complexity, from flat sets of key-value pairs, as shown in the example, to hierarchical tree-like structures containing clauses, predicates, topics, and lists, in addition to key-value pairs. Target languages also vary in complexity and form, *e.g.,* English, Spanish, French, Chinese, Japanese, HTML, SQL, speech waveforms.

Naturally, language generation systems differ in complexity, as well, depending upon the meaning representations and target languages that the systems must handle. Moreover, systems often grow in complexity over time, as their scopes widen. GENESIS is one such system. The Spoken Language Systems Group at MIT created GENESIS in the early 1990's to serve as the generation component of their GALAXY conversational systems [GPS94]. Over the years, GALAXY grew to encompass a broader range of domains (flight reservations, weather information, local directions) and languages (Chinese, Japanese, HTML, SQL, speech waveforms). Although the original GENESIS system has adapted well to some of its new roles, it is poorly suited to many of the tasks it must handle.

Over the past year, we have developed a new implementation of GALAXY's generation component. Our system, which we call GENESIS-II, resolves many of the original system's idiosyncrasies and shortcomings, and, in this thesis, we shall attempt to touch on every aspect of our new system. We begin with an introduction that offers some motivation for our research, as well as a description of the paradigms on which we built GENESIS-II. We conclude our introduction by outlining the remaining chapters of the thesis.

1.1 Motivation

When the original GENESIS system was implemented about ten years ago, its function in the GALAXY system was to generate paraphrases of user queries and to generate responses to the queries. While the system engineers anticipated multilingual generation, they designed the system primarily for English generation. Therefore, they did not thoroughly consider the difficulties associated with generating other natural languages, not to mention formal languages, such as SQL and HTML. As a consequence, it is difficult to use GENESIS for multilingual generation, and sometimes it is actually impossible to specify correct generation.

We may attribute these deficiencies to a couple of flaws in the GENESIS design. The first culprit is the paradigm of specificity that underlies the original system's framework. In many cases, GENESIS exerts too much control, by "hardwiring" certain aspects of generation, such as wh-query movement and conjunction and predicate generation. This lack of generality forces domain experts to work with prescribed generation formats, that may well be inappropriate for their domains and languages. Another reason for GENESIS's deficiencies is that it simply lacks many of the mechanisms necessary for handling multilingual generation. For example, GENESIS contains no mechanisms for disambiguating word senses or for rearranging meaning representations. As we shall see later in this thesis, such mechanisms are essential for correct multilingual generation.

As the role of generation in the GALAXY conversational system expanded over time, GENESIS evolved, as well. Especially as the range of domains and languages increased, engineers added functionality to GENESIS as needed. However, engineers found it difficult to modify GENESIS, because its framework was so narrow. Furthermore, the system lacked underlying principles, such as generality, consistency, and simplicity. Therefore, ten years after its inception, it is no longer a carefully architected system, but rather, a metaphoric snowball.

GENESIS's framework is confusing and constraining—difficult to learn and difficult to use. For example, GENESIS has different generation mechanisms for every type of constituent—clauses, predicates, topics, lists, and keywords. Therefore, a domain expert must learn several metalanguages in order to use Genesis. Genesis's method for ordering these constituents also poses problems for domain experts. For one, Genesis employs a mixed metaphor for ordering constituents. The order of predicates is governed by the order of rules in the grammar file, whereas the order of other constituents is governed within each rule. Not only is this approach confusing, but it is also constraining, since it renders context-dependent ordering of predicates impossible. As a consequence, realizations in the target language are often awkward or simply incorrect.

1.2 Design Paradigms

Roughly a year ago, we began to design Genesis-II, a new generation module for the Galaxy system. After ten years of generating within Galaxy, we had an understanding of generation that was both wide and deep. So, in many ways, our task was simpler than the task of the original Genesis engineers. Yet, we could not for a moment believe that we knew every function our system would ever have to fulfill. Our system would certainly have to solve the many known problems, but, furthermore, it would have to be powerful enough to handle those we could not have anticipated. Therefore, when designing our framework, it was essential for us to establish a foundation of design paradigms—principles that would make our system both simple to use and simple to extend.

We built Genesis-II on three paradigms: generality, consistency, and simplicity. In this section, we examine each of these principles in some depth.

1.2.1 Generality

As mentioned before, the original Genesis system was built on an implicit paradigm of specificity, and many of the system's deficiencies were directly caused by this underlying principle. For example, the system determines exactly when to generate conjunctions, and so, a domain expert cannot specify a domain-specific or language-specific generation pattern for conjunctions. Predicate generation is also hard-wired. Genesis generates all of the predicates in a frame at one time and concatenates the results based on the order of rules in the grammar file. Therefore, it is impossible to interleave predicates and keywords, and, as you might imagine, interleaving is essential for correct generation in some domains and languages.

Given the difficulties caused by GENESIS's paradigm of specificity, we were determined to build its successor on a foundation of generality. Therefore, when designing GENESIS-II's framework, we strove to strike the difficult balance between solving known problems and anticipating those still unknown, by devising a set of mechanisms that were powerful yet flexible. We carefully considered each proposed mechanism to determine whether it was too specific, and whenever possible, we generalized the mechanism to encompass a broader range of generation.

Over the past few months, domain experts with sophisticated generation requirements have tested GENESIS-II's range by exercising our system's most advanced features. As hoped, the mechanisms inspired by our paradigm of generality have proven useful time and again in many unexpected ways. For example, we devised a certain group of mechanisms in order to address the challenge of word-sense disambiguation, and our foreign language experts have since been able to generate strings that were impossible to generate in the original system. Additionally, because we designed this group of mechanisms in a general fashion, they were useful in a way we had not anticipated: they were ideal for prosodic selection in speech-waveform generation. GENESIS-II's mechanisms for frame manipulation have also been useful in unexpected ways. Originally, we devised them for handling the complex linguistic phenomenon of wh-movement and for handling differences in word order between languages. However, the generality of these mechanisms ensured that their utility would extend beyond the known uses, and, in fact, domain experts have used GENESIS-II's frame-manipulation techniques to solve a wide range of generation problems.

1.2.2 Consistency

At first glance, consistency seems like a commonsensical paradigm in software engineering. However, over time, as a system evolves, it may lose its foundation in such principles. Sometimes, the loss is a consequence of modifications made by a series of mostly unrelated engineers. At other times, it is the consequence of rapidly solving problems as they arise, with little regard for how the modifications affect the system as a whole. Over the past ten years, the original GENESIS system has lost its sense of consistency. As mentioned before, its framework is confusing. This is due in large part to its many inconsistencies, one of which is the mixed metaphor for ordering described above.

Consequently, the second of our three pillars is the paradigm of consistency. Throughout the design process, we evaluated each proposed mechanism not only for generality, but also for consistency. If a mechanism seemed to conflict with existing mechanisms, we sought ways in which to eliminate the dissonance. Sometimes this entailed revising the proposed mechanism. Occasionally, though rarely, it entailed revising existing mechanisms.[1]

GENESIS-II embodies the paradigm of consistency in many ways. In particular, it unifies the generation mechanisms for all constituents—clauses, predicates, topics, lists, and keywords. As mentioned before, the original GENESIS system has a different generation mechanism for each, and so, domain experts must learn several metalanguages, which can be a daunting and often confusing challenge. When designing GENESIS-II, we focused on creating a set of mechanisms that encompassed all of the constituents, and so, domain experts learn just one metalanguage in GENESIS-II.

We also recognized the importance of establishing consistency across the components of GENESIS-II. As we shall discuss in the following chapter, each GENESIS-II knowledge base comprises a grammar, lexicon, and set of rewrite rules. When we designed the grammar and lexicon, specifically, we were careful to construct each mechanism, such that it adhered to the conventions of its component, without conflicting with similar mechanisms in the other component. For example, the semantics of a caret (^) in one component should not vary drastically from the semantics of a caret in another component.

1.2.3 Simplicity

The final paradigm we discuss is the paradigm of simplicity—yet another commonsensical principle that is buried all too quickly in software systems. Over the years, GENESIS became a complex and confusing system. Its idiosyncrasies are difficult to remember, and few people have a thorough understanding of why it generates some of the strings it does. We strove to make GENESIS-II a simpler, yet more powerful, system.

Generation is an inherently difficult task, and yet, a well-designed framework can greatly simplify the work of domain experts. In designing GENESIS-II, we examined the inelegant constructions of the original system and devised cleaner mechanisms for achieving the same

[1]Because domain experts began developing knowledge bases in GENESIS-II early on, backwards-compatibility was always a consideration. Therefore, we did not take revisions to existing mechanisms lightly.

results. A prime example is the SQL domain. Some years back, engineers expanded GENESIS to handle SQL. However, because of the system's limitations, the modifications were at best inelegant. Clunky as they were, they nonetheless sufficed in generating accurate, if somewhat over-constrained, SQL strings. When porting the SQL domain to GENESIS-II, our domain expert successfully used the frame-manipulation mechanisms mentioned above for a task we had not anticipated. The rules in her knowledge base were much simpler: they were more concise, and they were fewer in number. As a result, she reduced the size of the SQL knowledge base by 50%, and the database queries she generates are cleaner than those generated before.

1.3 Outline

We have divided the body of this thesis into six chapters, book-ended between the Introduction and Conclusion. We intend for this work to provide the reader with a complete introduction to our system, and so, we will devote considerable attention to laying the groundwork for understanding the most sophisticated aspects of GENESIS-II. It is our hope that the early chapters will provide the reader with a firm foundation on which we can build. They are absolutely essential for understanding the more interesting aspects that we shall cover in the latter chapters. The chapters are divided as follows:

- **Chapter 2: Background**

  We present an overview of language generation research. In this chapter, we define a scheme for classifying the different approaches to language generation. Next, we discuss the classic debate in the generation community. We then describe several language generation systems. Finally, we position our own system within the broad spectrum of generation research.

- **Chapter 3: Generation in Galaxy**

  We describe the role of generation in the GALAXY conversational system. In this chapter, we focus on external features, such as the tasks the GALAXY generation module must fulfill and the input it must recognize. We conclude with some examples of generation in various languages and domains.

- **Chapter 4: Overview of Genesis-II**

We present a high-level overview of GENESIS-II. We begin by outlining the system's architecture and introducing the system's linguistic components. We then describe the general algorithm with which GENESIS-II converts a meaning representation into a string

- **Chapter 5: Basic Generation in Genesis-II**

  We describe GENESIS-II's basic commands.

- **Chapter 6: Advanced Generation in Genesis-II**

  We describe GENESIS-II's advanced commands.

- **Chapter 7: Evaluation of Genesis-II**

  We provide some metrics for evaluating GENESIS-II, and we present several examples of sophisticated generation in GENESIS-II.

We have also included a "Quick Reference Guide to GENESIS-II" as Appendix A.

Chapter 2

Background

We preface our description of GENESIS-II with an overview of research in natural language generation (NLG). In the first section of this chapter, we define a scheme for classifying the different approaches to generation, by drawing on and uniting the classification schemes of some of the leading generation researchers. We then consider the classic debate in generation (NLG vs. Templates). Next, we present an in-depth examination of five surface generation systems. Finally, we place our system, GENESIS-II, within the context of this field.

2.1 Approaches to Generation

As one might imagine, there are many approaches to language generation, and, at times, the forum for discussion has been filled with derisive voices, each resolute in the belief that its own school of thought is the best one. In the next section, we delve into the smoldering embers of what was once a raging debate. However, we must first define a classification scheme and describe the varying approaches to generation.

We derive our classification scheme by composing and expanding the classification schemes of Robert Dale (Macquarie University, Australia), Eduard Hovy (University of California), and Ehud Reiter (University of Aberdeen, England)—three of the foremost researchers in this field. Dale and Reiter have collaborated often in conducting NLG research and in leading NLG workshops, and our interpretation of their classification scheme is based on an overview of building natural language generation systems that they coauthored a few

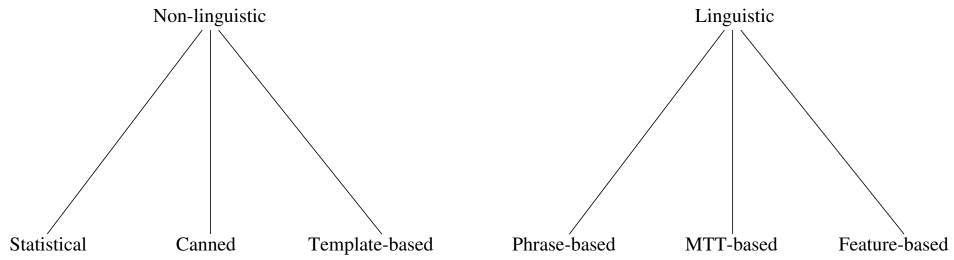

Figure 2-1: A schematic representation of our classification scheme.

years ago [RD97].[1] Eduard Hovy, the other researcher whose writings we consider in this section, is indisputably one of the leaders in generation research, and we base our interpretation of his classification scheme on a chapter [Hov96] he contributed to the *Survey of the State of the Art in Human Language Technology* [CMU$^+$96]. We duly cite Dale, Hovy, and Reiter throughout this section, whenever we include categories from their classification schemes and summarize their descriptions. Of even more interest, perhaps, are their omissions of certain categories, and we note those, as well.

In our classification system, we identify six specific approaches to generation, and we divide them evenly into two categories—"linguistic" and "non-linguistic." As depicted schematically in Figure 2-1, we consider statistical, canned, and template-based approaches to be "non-linguistic", and phrase-based, MTT-based, and feature-based approaches to be "linguistic."[2] Let us look more closely at each of these generation methods, from simplest to most sophisticated.

2.1.1 Statistical

We define the statistical approach to be one which relies primarily on statistical means for generation. Typically, a system that falls into this category must be trained on annotated corpora. Afterward, the system can produce generation strings for its input by applying a probabilistic model to what it has learned. A clear advantage of statistical systems is that they do not require hand-crafted rules, which are undeniably time-consuming to write. A disadvantage of statistical systems, however, is that the strings they generate are of a

[1]We should also note that they fleshed out the subject in a recently published book [RD00].

[2]Some NLG researchers prefer to use the categories, "shallow" and "deep", instead of the categories, "non-linguistic" and "linguistic." We prefer the latter two, because they are less colored.

poorer quality than the strings generated by other types of generators, as even the engineers of statistical generators sometimes concede [Rat00]. We postpone a deeper exploration of statistical systems' advantages and disadvantages until Section 2.3, in which we describe two statistical generators in detail [OR00, Rat00].

The inclusion of the statistical approach is one of our classification system's augmentations. Neither Hovy's overview nor Dale and Reiter's overview includes the statistical approach. It is tempting to chalk up these omissions to the generation community's scorn for non-linguistic methods. However, the absence may be more accurately explained by the fact that the statistical approach is relatively young in the field of language generation.

2.1.2 Canned

The canned approach encompasses systems that print static strings with no changes at all. For example, many programs have standard error and warning messages that are triggered by certain events and are therefore event-specific; thus, the text of the messages need not change. According to Hovy, the majority of software programs use canned text systems for generation.

Hovy includes this category in his overview. Dale and Reiter do not, however—most likely because their definition of generation does not encompass the task of outputting canned responses. In Section 2.2, it will become clear why canned generators might be excluded by some in a field that deigns to acknowledge even the more sophisticated template-based approach.

2.1.3 Template-based

The template-based approach is defined as one in which much of the generated text is static, but some of the text is dynamic and depends on the current environment or state of the program. For example, a program may wish to greet a user upon initialization with the words `"Welcome, <Insert user name>!"`, where `"<Insert user name>"` is replaced by the user's name. Traditionally, template-based systems have minimal linguistic capabilities, although some can handle simple tasks, like subject-verb agreement and the substitution of pronouns for nouns. Hovy mentions several template-based generators, both in commercial systems and in research systems. CoGenTex (Ithaca, NY) and Cognitive Systems Inc. (New Haven, CT) have template-based generation components in commercially available systems.

On the other hand, the early generator ANA [Kuk83] and the "sophisticated", multisentence generator in TEXT [McK85] are products of the research community.

Both overviews we surveyed include the template-based approach. In the next section, we examine the generation community's evolving view on templates. At that time, we also discuss the advantages and disadvantages of this approach to language generation. In Section 2.3.3, we describe a specific template-based generation system in some detail.

2.1.4 Phrase-based

Both overviews [Hov96, RD97] place exactly one category between the template-based and feature-based approaches. Hovy identifies that category as the phrase-based approach, whereas Reiter and Dale consider it to be the MTT-based approach. Their descriptions of these two approaches, as well as their references to systems that take each of these approaches, imply that the phrase-based and MTT-based approaches are distinct, and so, we distinguish between them, as well.

According to Hovy, phrase-based systems employ "generalized templates." Such systems select a phrasal pattern to match the top-level input and then expand each part of the pattern to match a portion of the input. This cascade of pattern-matching halts when each part of the input string has been replaced by a terminating substring. Hovy cites MUMBLE [McD80, MMA$^+$87] as a sophisticated example of a phrase-based system for single-sentence generation, and he cites his own RST text structurer [Hov88] as an example of a phrase-based system for multisentence structure generation.

2.1.5 MTT-based

Reiter and Dale define the MTT-based approach to be one that uses "meaning-text grammars." Based on Meaning-Text Theory (MTT) [Mel88], such systems use dependency grammars to convert a deep syntactic structure into text. Typically, MTT-based systems have several stages; for example, the RealPro MTT-based generator [LR97], which we feature in Section 2.3.1, has the following five stages:

- deep syntactic component
- surface syntactic component
- morphological component

- graphical component
- formatter

Reiter and Dale cite several other examples of MTT generators, including AlethGen [Coc96] and GhostWriter [MCM96]. They also mention FoG, the remarkable bilingual Forecast Generator deployed in Canada for generating weather forecasts in both French and English [GDK94]. The reference is worth following; the article, which appeared in a 1994 issue of *IEEE Expert*, provides a clear explanation of this multilingual generation system and of MTT-based systems in general. Incidentally, both RealPro and FoG were developed at CoGenTex, a small text generation company and Reiter's former employer.[3]

2.1.6 Feature-based

The final approach described by Hovy, Dale, and Reiter is the feature-based approach, also known as the systemic approach, due to its foundations in Halliday's Systemic Functional Linguistics [Hal85]. In systems that take such an approach, each sentence is described by a unique set of features. In one of Hovy's examples, a sentence is either `POSITIVE` or `NEGATIVE`, it is a `QUESTION`, an `IMPERATIVE`, or a `STATEMENT`, and its tense is `PRESENT` or `PAST`, etc. In Hovy's words, "any distinction in language is defined as a feature, analyzed, and added to the system"; in Dale and Reiter's words, "the central task is not viewed as the finding of a chain of grammar rules which convert an input structure into a sentence, but rather that of making a series of increasingly fine-grained choices which taken together determine the syntactic characteristics of the sentence being constructed." By incrementally collecting such features for each part of the input, a final generation string is eventually determined. The most widely-used feature-based single-sentence generators are PENMAN [Mat83, MM85], its descendent KPML [BMTW91, Bat96], and SURGE [ER96], which uses the Functional Unification Grammar Framework (FUF) [ER92]. Of PENMAN, KPML, SURGE/FUF, and the other "sophisticated" feature-based single-sentence generators Hovy lists, not one is in commercial use; he also notes that there are no feature-based multisentence generators. We shall discuss FUF in more detail in Section 2.3.2.

[3] Recall that Hovy also mentions CoGenTex, in the context of commercially-available template-based generators. It is unclear whether he is referring to the same CoGenTex products that Dale and Reiter classify as MTT-based.

2.2 NLG vs. Templates

Years ago, researchers in the language generation community regarded template-based approaches with great disdain. As evinced in the borrowed title of this section [Rei95], the extent of the disdain was so great that some researchers even refused to place template-based systems under the umbrella of generation.

Over the years, however, the animosity has dissipated. At a Workshop on Natural Language Systems held in 1999, at least four of the papers advocated the use of templating in generation systems [BH99, CEMR99, vDKT99, Hei99]. Perhaps it would be more accurate to say that the acceptance of templates by many in the NLG community was a foregone conclusion by this point, for as one of the authors wrote, "At this workshop, it really would be carrying coals to Newcastle to argue for the necessity of an integration of both 'free', fully linguistic generation and template-based approaches" [Hei99].

In this section, we sift through various writings on the subject of template-based generation vs. linguistic generation. We begin with an exploration of the advantages and disadvantages of each of the two approaches. Throughout this exploration, we include the opinions of other generation researchers. We conclude with a discussion of hybrid systems, such as the ones advocated by Heisterkamp [Hei99] and others in the workshop mentioned above.

2.2.1 Why use template-based generation?

The template-based approach to generation has some advantages over linguistic approaches. After all, there must be a reason that most software developers choose template-based generation. In the words of Heisterkamp, a researcher at the Daimler Chrysler Research Center in Germany, "the benefits [of templates] by far outweigh the shortcomings" [Hei99].

The most often cited advantage of the template-based approach is simplicity. When developing a software product that needs generation capabilities, the "quick and dirty" solution is to implement a template-based generation module. Linguistic approaches require much more planning and research. Furthermore, as Ehud Reiter notes, "there are very few people who can build NLG systems, compared to the millions of programmers who can build template systems" [Rei95]. Moreover, it is extremely difficult to find domain experts who are also experts in developing grammars for linguistic generators [RD97]. (Imagine explaining a complex feature-based system to a foreign-language expert, whose understanding of Computational Linguistics and English is minimal.)

Additionally, linguistic approaches can be utilized only if the system uses an intermediate meaning representation for the information to be generated. According to Reiter, few systems use such representations, and the cost of implementing them is high. In *NLG vs. Templates*, Reiter provides an excellent example of a situation in which template-based generation seems to be more cost-effective than linguistic generation [Rei95]. Consider a scientific program that prints out the number of iterations of an algorithm. At first glance, the following structures are all possible outputs that the system has to generate correctly:

- *No iterations were performed.*
- *1 iteration was performed.*
- *2 iterations were performed.*

Given morphological issues, such as the subject-verb agreement problem in the example, a template-based approach appears to be insufficient for this program. However, as mentioned above, the linguistic approach presupposes a meaning representation. Therefore, this system would need a component to translate concepts, like algorithms and iterations, into some intermediate syntactic or conceptual representation, which it could then send to the linguistic generation component. Clearly, adding both components is a time-consuming task for the developer. Moreover, since the morphological problem in the example is avoided by displaying the result in the form *"number of iterations performed: N"*, implementing a linguistic generation component for this program seems even more wasteful.

Another advantage of templates is that their behavior is predictable [Hei99]. In theory, linguistic generation systems have predictable behavior, as well, but often, it is difficult to trace the system's path through the long list of rules to determine why the generated text is not what the developer expected.

2.2.2 Why use linguistic generation?

Linguistic generation clearly has benefits, as well. Ease of maintenance is often cited as an advantage of linguistic methods over non-linguistic methods [RM93, Rei95]. For example, consider a situation in which a system needs to change its generation of time constructs from a military format to an "o'clock" format. In a template-based system, this would require making changes to every template with a time component, whereas in a linguistic-based system, it should require changing just one rule.[4] As both Reiter [RM93] and Goldberg [GDK94] point out, the development period of software is relatively short in comparison with its maintenance period. Therefore, investing extra time in developing a robust and manageable system often saves time and increases the system's lifespan.

Another clear advantage of linguistic systems over non-linguistic systems is quality [Rat00, Rei95]. Even proponents of non-linguistic approaches concede that the output of linguistic systems is more often correct and of a higher quality than output of non-linguistic systems [Rat00].

In a paper on her group's spoken dialogue system [San99], Lena Santamarta, a researcher at the Linköping University in Sweden, makes several astute observations on desired quality of generation output in dialogue systems. For example, it would be desirable for such a generation system to adapt word choice in its output based on the word choices of the user. This is very difficult to do with canned-text and template-based systems, but manageable in linguistic systems. Also, given the channel restrictions of dialogue systems, it may be important to limit the information generated to only that which answers the user's question; to achieve this in a template-based system could require an excessive number of templates.

Multilingual generation appears to be another arena in which linguistic approaches surpass non-linguistic approaches, and in the current atmosphere of rapid technological development and information dispersal, this arena is coming to the forefront [Hei99]. There are many examples of multilingual systems that take a linguistic approach to generation [GPS94, GDK94], but considerably fewer examples of non-linguistic approaches to multilingual generation. According to Reiter [Rei95], developers of template-based systems have two options when porting their systems to a new language. Either they must replace all of the original format strings with strings in the new language or they must build separate systems. Neither approach seems desirable; well-constructed linguistic systems, on the other hand, have robust kernels that can be applied to knowledge bases in many domains and languages.

In his paper *NLG vs. Templates* [Rei95], Reiter asserts that 99.9% of commercial systems use a template-based approach (or, in our terminology, the even more primitive canned

[4] This could achieved in even a template-based system, if the system permits "subroutine calls."

approach). He therefore urges NLG researchers to honestly assess the advantages and disadvantages of their linguistic systems. Such analyses will be essential in proving that there are reasons for developers to choose the sophistication of linguistic generation over the simplicity of templates. In an interesting twist, Reiter actually opted for template-based generation in a recent project [Rei99], while lamenting the dearth of adequate linguistic generation packages. He chose to implement his own "shallow" (template-based) approach to generation in the STOP system, a program for generating personalized pamphlets to encourage individuals to quit smoking. In his words,

> Using shallow techniques for syntactic processing in STOP was a disappointment. I hope that in the future some NLG group does develop a realisation component which is well-documented, well-engineered as a software artifact, and has a wide-coverage grammar; this would allow future STOP-like projects to use deep techniques for realisation.

We revisit Reiter's assessment in Section 2.3.2.

2.2.3 Hybrid generation

Ehud Reiter appears to have been a pioneer in the integration of template-based and linguistic approaches to language generation. In this section's eponym [Rei95], now several years old, Reiter advocates the use of hybrid techniques in NLG systems, and he cites his own IDAS system as an example [RML95]. In that overview, he describes IDAS as a system that can "embed NLG-generated fragments into a template slot, or that [can] insert canned phrases into an NLG-generated matrix sentence." From his more detailed paper on the subject, however, I gleaned that IDAS is a "hybrid" system in that meaning representations can be hybrids [RM93]. The realization process seems to be strictly rule-based.

Van Deemter et al. describe a hybrid system of sorts in their paper *Plan-based vs. template-based NLG: a false opposition?* [vDKT99]. In the paper, they describe D2S, a data-to-speech method that has been applied to several generation applications. The input to D2S is a syntactic tree which contains some "open slots", analogous to template slots. The system generates all possible trees with the open slots filled in. It then checks each possible tree to determine whether it obeys Chomsky's Binding Theory and is compatible with the Context Model, "a record containing all the objects introduced so far and the anaphoric relations among them." With such non-trivial components, D2S does not seem to be a "template"-based system that one can easily dismiss. Claiming that their template-based method is as theoretically well-founded, robust, and maintainable as any other generation method, Van Deemter and his colleagues find the NLG/Template debate to be a "caricature." Their work certainly counters the claim that "template-based NLG systems are always linguistically less interesting" than other NLG systems.

Since there is far too much literature on hybrid systems to cover in this overview, we direct the reader to the following selected papers: [BH99, Bus96, CEMR99, Hei96, MTD96].

2.3 Examples

In this section, we examine several generation systems in detail. We begin by describing two generation *packages*—"off-the-shelf" surface generation components that software developers can incorporate into their systems. Both packages are linguistic-based generators that we mentioned briefly in Sections 2.1.5 and 2.1.6; the first package, REALPRO, is an MTT-based system, and the other package, FUF, is a feature-based system. After discussing REALPRO and FUF, we examine the generation modules of two conversational systems. We look at a template-based generator and a statistical generator for IBM's flight information system and at a statistical generator for Carnegie Mellon's flight information system.

2.3.1 RealPro

As noted before, REALPRO is a product of CoGenTex, a small software company that specializes in generation applications. In their paper describing REALPRO [LR97], Benoit Lavoie and Owen Rambow state that, although REALPRO is derived from previous work [IKP88, IKK+92, RK92], the REALPRO system has an original design and an original implementation. Like GENESIS-II, REALPRO has a core C/C++ kernel that interprets ASCII files specifying domain information at runtime. A set of such files, known as a Linguistic Knowledge Base (LKB), comprises grammar rules, lexical entries, and feature defaults.

The REALPRO generator takes as input a syntactic dependency structure known as the Deep-Syntactic Structure (DSyntS), which is based on a structure in Mel'čuk Meaning-

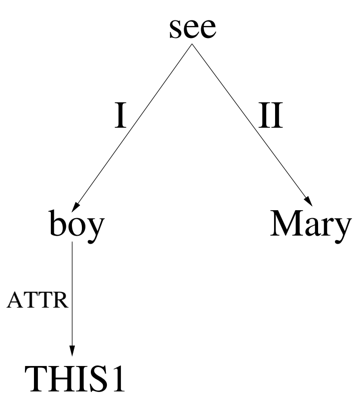

Figure 2-2: DSyntS for the sentence, `"This boy sees Mary."`

Text Theory (MTT) [Mel88]. In their paper, Lavoie and Rambow list five of the DSyntS's "salient" features:

- It is an unordered tree with labeled arcs and nodes.
- The nodes reflect uninflected words in the target language.
- The DSyntS is a dependency structure.
- It is not a semantic structure, and so, all labels reflect syntactic relationships and not conceptual relationships.
- Since it is a deep syntactic representation, only meaning-bearing words are included in the DSyntS. There are no function words.

Lavoie and Rambow proceed to illustrate the concept of the DSyntS with the example shown in Figure 2-2. The tree structure in the figure is the DSyntS for the sentence, `"This boy sees Mary."` Although absent from the picture, the nodes have features that describe attributes of their labels. For example, the node `see` has the feature `word-class:verb`. As Lavoie and Rambow point out, adding the feature `question:+` to `see` and the feature `number:pl` to `boy` would result in a tree corresponding to the sentence, `"Do these boys see Mary?"` Furthermore, this second example demonstrates the fifth point in the list above; the function word `"do"` is absent from the DSyntS, because only meaning-bearing words are included in the tree.

The RealPro generation system has five stages, which we listed briefly in Section 2.1.5 and which we flesh out here:

- The Deep-Syntactic Component enhances the input tree by adding function words, based on information in the grammar and lexicon.
- The Surface-Syntactic Component linearizes the nodes of the enhanced tree.
- The Deep-Morphological Component uses the lexicon to inflect the constituents of the linearized string.
- The Graphical Component adds punctuation and formatting information to the string.
- *Ad-hoc* formatters convert it into its final form—ASCII, HTML, or RTF.

Lavoie and Rambow list several selling points of their product. The system is replete with an LKB for English, and one for French is on the way. Furthermore, the LKBs are reconfigurable, and software developers can customize and extend them as necessary. The grammar handles many linguistic forms, including relative clauses, pronouns (personal, possessive, and relative), punctuation, default word order, and various verbal tenses and moods. The system itself is platform-independent, and its socket interface allows it to operate as a standalone server. Generation runtime is linear in the number of nodes in the DSyntS, such that an n-node DSyntS takes about $.015n$ seconds to process. We reserve our critique of RealPro until the end of Section 2.3.2, in which we briefly evaluate both RealPro and FUF.

2.3.2 FUF

Developing a deeper understanding of feature-based systems is a difficult task. Because feature-based systems are much more theoretically-grounded than some of the other NLG systems, papers describing feature-based algorithms tend to lack the implementation details essential to system developers and curious researchers. In a book chapter entitled *How a systemic functional grammar works: the role of realization in realization* [FTL93], Robin Fawcett et al. describe a feature-based generator in detail, because they feel that there is a lack of such descriptions in the literature. In their words,

> In one sense, SFG [systemic functional grammar] needs no introduction to the NLG community. It is increasingly widely referred to in the literature of computational linguistics (CL) in general, and in the literature of NLG in particular,

> and it has a clearly established reputation as a usable model. Yet there is here a serious anomaly. Surprisingly, and despite the plethora of publications, it seems that a relatively small proportion of researchers in NLP (whether in NLU or NLG) have a real understanding of **how a SFG actually works**. We have given demonstrations of GENESYS [Fawcett et al.'s feature-based generator [FT90]] at many conferences throughout the world, and it has been our usual experience to find in many of those present a slightly surprising mixture of interest and ignorance.
>
> When we review the available literature, we find that we have to admit that this lack of understanding is more the fault of the systemic linguists than the inquirers. The surprising fact is that **there appears to be no generally available account of how a large implemented SFG sentence generator works, at a reasonably detailed and technical level**.

That said, we attempt a deeper description of one feature-based approach. We focus on FUF, as described quite clearly by Michael Elhadad, one of FUF's originators, in the FUF user manual [Elh91]. The FUF system receives as input both a grammar and a meaning representation, known as a "Functional Description" (fd). The system processes the two inputs and generates a well-formed English sentence that both reflects the content of the fd and meets the constraints of the grammar. The generation process comprises two major stages, unification and linearization. In the first stage, the input-fd is unified with the grammar to create a more detailed fd. In the second stage, the system linearizes the enriched fd and employs a morphological module to handle the linguistic details.

Let us look at the simple example provided by Elhadad in the FUF manual. Suppose the generator receives as input the fd shown in Figure 2-3 and the simple grammar shown in Figure 2-4. Without a deep understanding of the system or the lisplike language it employs, we can see that the input-fd is of the category `s` (Figure 2-3, Line 1) and that the grammar describes a rule for the category `s` (Figure 2-4, Line 2). Furthermore, we can see that the `number` of the `verb` appears to be related to the `number` of the `prot` (Figure 2-4, Line 6). In fact, the structure `"number {prot number}"` sets the constraint that the `verb` must agree in number with the `prot`.

```
(set ir 01 '((cat s)
             (prot ((n === john)))
             (verb ((v === like)))
             (goal ((n === Mary)))))
```

Figure 2-3: A simple input-fd.

```
1 ((alt MAIN (
2    ((cat s)
3     (prot ((cat np)))
4     (goal ((cat np)))
5     (verb ((cat vp)
6            (number {prot number})))
7     (pattern (prot verb goal)))
8    ((cat np)
9     (n ((cat noun)))
10    (alt (
11      ((proper yes)
12       (pattern (n)))
13      ((proper no)
14       (pattern (det n))
15       (det ((cat article)
16             (lex "the")))))))
17   ((cat vp)
18    (pattern (v dots))
19    (v ((cat verb)))))))
```

Figure 2-4: A simple grammar.

The result of unification in our example is the enhanced fd shown in Figure 2-5. Note that the unification would have failed if the input-fd had been the fd shown in Figure 2-6. The fd's constraints that the `prot` and `verb` be `singular` and `plural` respectively violate the subject-verb agreement constraint of the grammar, and therefore, the unification fails.

During the linearization stage, the `pattern` rule determines the word order of the enhanced fd's constituents. In our example, the pattern is `"prot verb goal"` (Figure 2-4, Line 7). The morphological component ensures that `"like"` is inflected correctly, and the resulting string is `"John likes Mary."`

RealPro and FUF are solidly grounded in linguistic theories—Meaning-Text Theory and Systemic Functional Linguistics, respectively. Given my limited knowledge of both

```
((cat s)
 (prot ((n ((lex "john")
            (cat noun)))
        (cat np)
        (proper yes)
        (pattern (n))))
 (verb ((v ((lex "like")
            (cat verb)))
        (cat vp)
        (number {prot number})
        (pattern (v dots))))
 (goal ((n ((lex "Mary")
            (cat noun)))
        (cat np)
        (proper yes)
        (pattern (n))))
 (pattern (prot verb goal)))
```

Figure 2-5: An enhanced fd.

```
(set ir 02 '((cat s)
             (prot ((n === john) (number sing)))
             (verb ((v === like) (number plural)))
             (goal ((n === Mary)))))
```

Figure 2-6: An input-fd that fails.

theories, I shall not attempt to critique RealPro and FUF's theoretical foundations. It also would be difficult to critique the packages themselves without actually attempting to integrate them into a software system. Ehud Reiter attempted such integration when creating the STOP system, discussed in Section 2.2.2. As described in his paper on the subject [RM93], he found both RealPro and SURGE (which uses FUF) to be unsatisfactory for his needs. His criticism of RealPro was that it lacked grammatical coverage; it was unable to handle the constructs he needed. He found SURGE difficult to use because it lacked enough documentation for him to get an adequate understanding of the system. It is most likely for reasons such as these that software developers prefer to implement template-based approaches. (Recall Hovy's observation that no commercial system uses a feature-based approach [Hov96].)

2.3.3 IBM's flight information system

Researchers at the IBM T.J. Watson Research Center are developing a conversational system in which callers interact with the system to obtain real flight information and to create travel itineraries. In this respect, their system is comparable to the Galaxy System's Mercury domain [SP00]. Furthermore, both systems use the architectural standard of the hub-based DARPA Communicator Project. For these reasons, the generation mechanisms of the IBM flight information system are particularly relevant to our work with Genesis-II.

At a recent conference (ANLP-NAACL2000) sponsored by the Association for Computational Linguistics (ACL), IBM presented two papers on generation in the flight information domain [Axe00, Rat00]. One of the papers discusses a template-based system for natural language generation [Axe00], whereas the other discusses a trainable system for natural language generation [Rat00]. We shall describe both systems presently.

A template-based system for natural language generation

In his paper, Scott Axelrod describes the template-based language generation module of IBM's flight information system [Axe00]. His generation module comprises two components—one for deep generation ("what to say") and one for surface generation ("how to say it")—a division made by many generation systems. The deep generation component of IBM's module outputs a meaning representation, which consists of a set of variables with assigned values[5], such as { [`$AIRLINE "American"`], [`$SOURCE "Chicago"`], [`$ARRIVAL_TIME "noon"`] }. The surface generation component, in turn, receives a meaning representation from the deep generation component and outputs a well-formed English paraphrase. In his paper, Axelrod gives a detailed description of both the deep generation component and the surface generation component. Because the GALAXY System's turn manager handles deep generation and GENESIS-II handles surface generation, we only discuss IBM's surface generation component.

IBM's surface generation module provides experts with a library of routines for generating common phrases, as well as a set of constructs with which experts can create their own templates. These templates can contain references to variables that correspond to values set during deep generation. The surface generation component can also call on a morphology component to generate correct morphological forms, given linguistic information such as number and tense.

Axelrod highlights one feature of the surface generation component in particular; he presents IBM's solution to the problem of combinatorial explosion, which he claims is a formidable challenge in generation. For example, a template with ten variables has 2^{10} realizations, depending on which variables are set during deep generation. To explore this problem, let us consider the following template written in a pseudo-language, where a word delimited by "$" represents a variable reference:

```
There is a flight on $AIRLINE arriving from $SOURCE at $ARRIVAL_TIME.
```

Suppose that the deep generation component specifies the source (`"Chicago"`) and the airline (`"American"`), but no arrival time. A naive surface generation component would produce the ungrammatical target string, `"There is a flight on American arriving from Chicago at."` To generate correctly given this generation component, we would have to replace the template above with 2^3 different templates—each for a case in which a different combination of the three variables are set. Axelrod's system overcomes this combinatorial explosion by allowing experts to specify subphrases within a template that may be "turned on" if a variable is set. In his system, the template above could by modified by specifying the subphrases, `"on $AIRLINE"`, `"from $SOURCE"`, and `"at $ARRIVAL_TIME"`, so that the

[5]The variable-value pair is known as the attribute-value pair in some systems and the keyword-value pair in others, including our own.

word "on" would appear only if there were a value for $AIRLINE and so forth. Using this mechanism, the IBM flight information system prevents exponential blow-up in the number of templates required for generation.

Like other template-based systems, IBM's generation module is clearly limited. For one, it can interpret only meaning representations that consist of one level of variable-value pairs. Both GENESIS-II and its predecessor handle hierarchical linguistic representations with clauses, predicates, topics, and lists, in addition to variable-value pairs. Furthermore, such features as the ability to "turn on" subphrases are not unique to IBM's system. Rather, they are essential mechanisms in all generation modules that handle domains of similar complexity. As evinced in an ICSLP paper from 1994 [GPS94], GENESIS-II's precursor had all of the features that Axelrod discusses in his ANLP—NAACL2000 paper. IBM's approach may be adequate at this time, but I suspect that they will encounter many challenges in scaling their system to handle more complex domains than flight information.

A trainable system for natural language generation

In the other IBM paper, Adwait Ratnaparkhi describes another approach that IBM is taking in the flight information domain [Rat00]. He has developed three surface generation systems that can be used to statistically generate paraphrases of meaning representations. All three surface generation systems must first be trained on large annotated corpora of "generation templates"—phrases in which values have been replace by variables. For example, if we were to replace values with variables in the phrase, "There is a flight on American arriving from Chicago at noon," the result would be the generation template, "There is a flight on $AIRLINE arriving from $SOURCE at $ARRIVAL_TIME."

Once trained, each of the systems operates in two stages. The input to the first stage is a set of variables, such as { AIRLINE, SOURCE, ARRIVAL_TIME }. Each of the systems has a different probability model, which it uses to select one of the generation templates learned during training as output for Stage 1. In our example, all of the following generation templates are possible outputs:

```
"There is a flight on $AIRLINE arriving from $SOURCE at $ARRIVAL_TIME."

"There is a $AIRLINE flight from $SOURCE at $ARRIVAL_TIME."
```

"There is a $ARRIVAL_TIME flight on $AIRLINE arriving from $SOURCE."

In the second stage, the generation component receives a set of variable-value pairs corresponding to the set of variables received in Stage 1, *e.g.,* { [$AIRLINE "American"], [$SOURCE "Chicago"], [$ARRIVAL_TIME "noon"] }. It then substitutes the values in for the appropriate variables in the generation template from Stage 1 and outputs the resulting string, *e.g.,* "There is a noon flight on American arriving from Chicago."

Ratnaparkhi's three systems differ in the probability models they use to select the generation template in Stage 1. The baseline system aptly uses the least sophisticated algorithm; it simply chooses from the training data the most frequent generation template corresponding to the input variables. The second of the systems uses an n-gram model for word generation; it selects the word sequence with the highest probability that includes each of the input variables exactly once. The final system improves upon the linear predictions of the second system by conditioning on syntactic dependencies, a method which requires that the corpus be annotated with tree structure. As might be expected, Ratnaparkhi's evaluations show that, overall, System 3 is slightly better than System 2, which is significantly better than System 1.

Most of the advantages and disadvantages of Ratnaparkhi's systems are common to all trainable, statistical generation systems. Although they are arguably faster to create because they do not require hand-constructed rules, they are less accurate, as Ratnaparkhi willingly concedes in his paper. Furthermore, statistical methods are clearly limited by their training corpora, whereas flexible linguistic-based systems with subroutine calls and recursion may generate correctly for combinations that were not even considered by the expert. Finally, it may be difficult to scale Ratnaparkhi's systems to handle more complex domains, especially those that require hierarchical meaning representations.

2.3.4 Stochastic language generation at CMU

Carnegie Mellon researchers Alice Oh and Alex Rudnicky are also developing a trainable, statistical generation system [OR00], and like IBM's Axelrod and Ratnaparkhi, they selected travel information as the domain in which to test their system. Although Oh's and

Rudnicky's work was published in parallel with Ratnaparkhi's[6], the behavior of the systems is strikingly similar. Once again, we have a system that must first be trained on what are essentially annotated corpora of "generation templates." The surface generation component then operates in the same two stages. First, the generation engine receives a set of variables and randomly generates a generation template for them, using an n-gram language model computed from the training data. In the second stage, variables in the generation template from Stage 1 are replaced with values. The only significant difference between CMU's statistical generator and IBM's statistical generators appears to be that they use different probability models.

Because the systems are so similar, my critique of IBM's system also applies to CMU's system; therefore, I shall not reiterate myself. I would, however, like to respond to some of the claims that Oh and Rudnicky make in their paper [OR00]. I begin with the following comment on the use of templates and rule-based generation in spoken dialogue systems:

> However, there is still the burden of writing and maintaining grammar rules, and processing time is probably too slow for sentences using grammar rules (only the average time for templates and rule-based sentences combined is reported in Busemann and Horacek, 1998) ...

The uncertainty in their claim is well-founded. After all, GENESIS-II, the system I know best, is a rule-based generation system, and it has never been a bottleneck in our dialogue system. Furthermore, in the paper to which Oh and Rudnicky refer, Busemann and Horacek state that their rule-based generator has an average generation time of less than a second—a time span so insignificant that it "can almost be neglected" in their non-dialogue system [BH98]. Their vague figure of "less than a second" in a non-dialogue system hardly translates to "unacceptably slow" in a dialogue system. So, given the sketchy evidence, Oh's and Rudnicky's claim that their system is "much faster than any rule-based system" appears to be unsubstantiated. Furthermore, given recent timed runs of our system, their claim appears to be false, as well. In Chapter 7, we shall compare the speeds of our system and theirs in more detail.

[6] Ratnaparkhi presented his work earlier this year at the Language Technology Joint Conference of Applied Natural Language Processing and the North American Chapter of the Association for Computational Linguistics (ANLP-NAACL2000). Oh and Rudnicky presented theirs at the Workshop on Conversational Systems at the same conference.

Oh and Rudnicky also state in their paper that evaluators of their surface generation component indicated that there was "no significant difference" in quality between the stochastic generator and a template-based generator, which supports their claim that their system works at least as well as template-based systems with considerably less work in creation and maintenance. Yet, they did not compare their system with a rule-based generator, even though they must be well aware that many NLG researchers consider templates to be a weak and simplistic alternative to rules. It will be interesting to see the results of further development and evaluation of their statistical method.

2.4 Genesis-II

We conclude our survey by discussing where we feel our system fits into the broad generation spectrum. Before we formulated GENESIS-II's approach to generation, we considered several important factors, one of which was the input GENESIS-II receives. GALAXY gives its generation component a very wide range of meaning representations. Consequently, we knew that our framework would have to handle a spectrum of input. On one end lies the hierarchical, linguistic meaning representation, consisting of key-value pairs, lists, linguistic clauses, predicates, and topics. Unquestionably, a powerful linguistic generator would be essential for processing such input. On the other end, lies the flattened "*e*-form" (electronic form) representation, consisting of simple key-value pairs and lists. A powerful linguistic generator would be far too unwieldy for processing this type of input.

When we considered these two extremes and the hybrids between them, we decided to formulate an approach that would encompass aspects of both linguistic and non-linguistic generation systems. In particular, we wanted to create a framework that would allow users to rapidly develop knowledge bases for simple domains. However, we wanted to infuse this framework with powerful mechanisms that domain experts could use to develop knowledge bases for more challenging domains. In essence, we wanted to design a framework that could be used for both simple template-based generation and challenging linguistic-based generation.

We feel that the present GENESIS-II system bridges the divide between linguistic and non-linguistic systems, and we hope that the reader will draw a similar conclusion upon reading this thesis. In Chapter 5, we shall describe how domain experts can specify simple rules that interleave canned text with keyword placeholders. Using such basic mechanisms, they can rapidly develop knowledge bases for generating from simple key-value meaning representations. In Chapter 6, we shall describe how domain experts can use Genesis-II's wide range of advanced mechanisms for more sophisticated generation.

As discussed in the Introduction, we built Genesis-II on paradigms of generality, consistency, and simplicity. If we were successful in this endeavor, domain experts will find that they do not need a doctorate in Computational Linguistics to utilize even the most advanced mechanisms in Genesis-II's framework. As a consequence, we hope that a wide range of domain experts will find our system useful.

Chapter 3

Generation in Galaxy

In the previous chapter, we presented an overview of language generation research, in order to provide the reader with a context in which to place GENESIS-II. We would like to further situate GENESIS-II by describing the role of generation in the GALAXY conversational system. Therefore, we will withhold an in-depth examination of GENESIS-II until the following chapters, and we will focus solely on external features, such as the tasks the GALAXY generation module must fulfill and the input it must recognize. We will conclude this chapter with a few examples of generation in various languages and domains.

3.1 Roles

The generation component's many functions within the GALAXY conversational system are mediated by the GALAXY hub. That is, a rule in the hub indicates that the hub should send the generator a meaning representation, along with the name of the domain it represents and the target language of the generation string. The generator then uses the domain and language information to generate a string from the meaning representation. In this section, we focus on the range of generation for which GALAXY's generator is responsible. In the next three paragraphs, we look at paraphrase generation, response generation, and formal-language generation, respectively.

When a user queries one of the GALAXY conversational systems, the system converts the query into a meaning representation. Before querying the database, GALAXY often summarizes the query for the user, so that the user knows if the system understands his question. To summarize the query, GALAXY sends the meaning representation to the generator. The generator then outputs a target string in a marked-up language that the speech synthesizer can interpret and convey to the user. Additionally, if the user has a graphical interface, the system displays a paraphrase of the query in a special window.

When the system produces a meaning representation in response to a user query, GALAXY sends the meaning representation to the generator for conversion into speech and/or text. The generator converts the response frame into a marked-up language the synthesizer can interpret. It must also be able to convert the response frame into text for users with a graphical interface.

Finally, GALAXY sometimes sends a meaning representation to the generator for conversion into a formal language. For example, the generator must be able to convert a hierarchical linguistic structure into a flattened *e*-form (electronic form)—a structure that the dialogue manager understands. The generator must also be able to convert the *e*-form into a database query, such as an SQL string. Additionally, if the user has a graphical interface, the generator may be called upon to generate hyperlinked HTML from a meaning representation.

Depending upon the domain and language, GALAXY may request that the generator perform any number of the tasks above. Presently, domain experts are developing knowledge bases for several domains and languages. The tables in Figure 3-1 list all of these languages and domains. For more information on the role of generation in GALAXY, we refer the reader to a recent conference paper on generation in the MERCURY flight domain [SP00]. The introduction to the paper outlines the generator's tasks within MERCURY, and the body of the paper contains several different examples of generation.

In the following section, we describe the form of GALAXY's input—the meaning representation. In the final section, having described generation input in detail, we shall return to the high-level view of generation by presenting several examples of generation input and output.

3.2 Meaning Representations

By definition, the input to a generation system is a meaning representation. In some systems, especially non-linguistic ones, a meaning representation is simply a set of keyword-value pairs. In more sophisticated systems, meaning representations are hierarchical lin-

Domains	
Jupiter:	Weather information
Mercury:	Flight reservations
Orion:	Off-line task delegation
Pegasus:	Flight status
Voyager:	Navigation assistance

Natural Languages	Other "Languages"
Chinese	*e*-forms
English	HTML
Japanese	Speech mark-up language
Spanish	SQL

Figure 3-1: Domains and languages in the Galaxy system.

```
1 {c weather_event
2    :topic {q weather_act
3              :name "fog"
4              :pred {p time_interval
5                       :topic {q time_of_day
6                                 :name "morning" } } } }
```

Figure 3-2: The meaning representation for `"morning fog"`.

guistic representations. Meaning representations in Galaxy fall into the latter category. Galaxy uses a powerful data structure called a "frame" to construct meaning representations. In this section, we describe the basic properties and components of frames. Throughout this introduction to frames, we will present examples of frames that illustrate the concepts we are discussing. At this point, we present our first example, the meaning representation in Figure 3-2, in order to give the reader a visual idea of the data structure we are describing.

3.2.1 Names

Perhaps the most basic feature of a frame is its name. Every frame has exactly one, and it always appears on the first line of the printed representation. For example, consider the meaning representation in Figure 3-2. The top-level frame in the figure is named `weather_event`, as displayed in Line 1. Additionally, there are three nested frames in this meaning representation. Line 2 is the first line of a frame named `weather_act`, Line 4 is the first line of a frame named time_interval, and Line 5 is the first line of a frame named time_of_day. We shall discuss the structures for nesting frames later in this section.

Frame names tend to be domain-specific. For example, the JUPITER weather domain includes frames named precip_act and to_degrees, whereas the MERCURY travel domain includes frames named destination and airline. There are, of course, some overlapping frame names, such as date.

3.2.2 Types

Another basic feature of a frame is its type. Every frame has exactly one type, and, like the name, the type always appears on the first line of the printed representation. There are three different types GALAXY assigns to frames. They are: clause, predicate, and topic. They can be abbreviated c, p, and q[1], respectively, and, in fact, the printed representation of frames uses these abbreviations. The meaning representation in Figure 3-2 contains at least one of each type of frame. The frame named weather_event is a clause frame, the frame named time_interval is a predicate frame, and the frames named weather_act and time_of_day are topic frames.

The GALAXY System assigns a type to a frame in a meaning representation based on the linguistic properties of the frame. For example, the system typically assigns the clause type to frames that represent linguistic clauses, such as sentences, and it generally uses the topic type to designate frames that represent noun phrases. The predicate type is used more liberally; it identifies frames representing prepositional and adjectival phrases, as well as standard verbal predicates. We shall discuss shortly how predicate frames differ from clause and topic frames in a significant non-linguistic way.

3.2.3 Keyword-value pairs

A frame conveys some meaning through its name and type. However, many frames use two other methods for communicating the majority of their semantic content. The first method we shall examine is the keyword-value pair, which, as we mentioned in Chapter 2, is a concept common to many generation systems. A frame may contain any number of keyword-value pairs, but every keyword in the frame must be unique. Keywords are always

[1]The letter q in topic frames is an abbreviation for "quantifiable set."

Value type	Example keyword-value pair
`String`	`:conditional "unseasonably"`
`Integer`	`:name 3`
`Frame`	`:topic {q date` `:day "wednesday" }`

`List`	`:cities ( "Ottawa"`
	`"Toronto"`
	`"Vancouver"`
	`"Calgary"`
	`"Montreal"`
	`"Ontario" )`

Figure 3-3: Key value types and examples.

strings, and, by convention, they are delimited by colons. The values associated with keywords, on the other hand, have many forms, four of which are legal in GALAXY meaning representations. These are: strings, integers, frames, and lists. Strings and integers are fairly self-explanatory; frames are not, but a description of them could become recursive all too quickly. The final form, the list, is slightly more complex, in that a list is a parenthesized, heterogeneous set of "objects", which, in theory, include strings, integers, frames, and other lists. Figure 3-3 presents a sample keyword-value pair for each of the four forms. Note that the sample list is a list of strings. For further examples of keyword-value pairs, we may refer back to the simple frame in Figure 3-2. It contains two types of keyword-value pairs. Line 2 contains the keyword `:topic` associated with the frame named `weather_act`, and Line 5 contains the keyword `:topic` associated with the frame named `time_of_day`. Line 3 contains the keyword `:name` associated with the string `"fog"`, and Line 6 contains the keyword `:name` associated with the string `"morning"`.

3.2.4 Predicate lists

It may seem that Line 4 of Figure 3-2 also contains a keyword-value pair, in which the keyword `:pred` is associated with the frame named `time_interval`. However, the `:pred` syntax[2] actually demarcates a member of the "predicate list."[3] Predicate lists are the

[2]The notation is admittedly confusing at first, because, as we noted before, the convention is to delimit keywords with colons. However, since this is a GALAXY-wide representation, we ought not to stray from it.

[3]In the linguistic space, a frame can have at most one topic, but it can have any number of predicates. As a consequence, GALAXY's engineers chose to utilize a list representation for predicates.

```
{c what_about
   :topic {q flight
             :pred {p airline
                      :topic {q airline_name
                                :name "united" } }
             :pred {p flight_number
                      :topic 201 }
             :pred {p arrival_time
                      :topic {q time
                                :military 1200 } } } }
```

Figure 3-4: The meaning representation for "United flight 201 arriving at noon".

second method by which frames convey the majority of their semantic information. The predicate list is simply a list of predicate frames. There are no restrictions on how many predicates a frame may have, but every predicate in the frame should have a unique name. The `flight` frame in Figure 3-4, for example, has three predicates, all of which have unique names.

Thus, there are two ways to nest predicate frames in a meaning representation: key value pairs and predicate lists. One might, therefore, wonder how a predicate in this list differs from a keyword-value pair in which the value is a predicate frame. For one, they are different entities in the computer's internal representation of the frame data structure. However, the two forms also differ in a way visible to the user. The primary difference is that the only way to identify a predicate frame in the predicate list is by the frame name, whereas a key value frame can be identified either by the keyword with which it is associated or by the frame name. Let us expound on this by referring again to the frame in Figure 3-4. Because keywords are unique within a frame, the keyword `:topic` can be used to explicitly identify the `flight` frame in the top-level `what_about` frame. On the other hand, if we wish to identify the `airline` predicate frame in the `flight` frame, we cannot refer to it as the value associated with `:pred` for a couple of reasons. The primary reason is that the `flight` frame contains more than one predicate. Therefore, the handle `:pred` does not uniquely identify any of the predicates, and so, the only explicit means of referring to the `airline` frame is by its name. Although there are cases in which a frame contains only one predicate, we shall maintain consistency by always identifying predicates in the predicate list by their frame names and by always identifying frames associated with keywords by the keywords.

3.3 Examples

Having described the GALAXY meaning representation in detail, we can finally present the reader with a few concrete examples of generation in GALAXY. For each example, we shall include the desired generation output for one or more languages. Our sole intention is to provide the reader with a sample of the expected generation input and output, so that the reader may have a more tangible sense for the scope of generation in GALAXY. In the following chapter, we begin an in-depth examination of how our system produces the desired results.

- For our first example, consider the following meaning representation from the JUPITER weather domain:

  ```
  {c weather_event
     :topic {q precip_act
               :qualifier "heavy"
               :name "rain"
               :number "pl"
               :pred {p month_date
                        :topic {q date
                                  :name "tonight" } } } }
  ```

  Domain experts are developing knowledge bases for JUPITER in several languages other than English—namely Spanish, Chinese, and Japanese. Therefore, depending on the target language, GALAXY's generation module would be expected to convert the meaning representation above into one of the following strings[4]:

  - *English*: `heavy rains tonight`
  - *Spanish*: `lluvia intensa esta noche`
  - *Chinese Pinyin*: `jin1_wan3 you3 bao4 yu3`

- As discussed in Section 3.1, GALAXY uses its generation module for a number of tasks. For example, after GALAXY converts a user query into a meaning representation, it

[4]We exclude Japanese, simple Chinese, and traditional Chinese from this example because of their character sets.

might send the meaning representation to the generation component, which produces a paraphrase and a database query. To illustrate these two tasks, we present the following meaning representation for a query in the JUPITER domain:

```
{c truth
   :exist "it"
   :topic {q weather_acting
             :name "rain"
             :pred {p in
                      :topic {q city
                                :name "boston" } } } }
```

When GALAXY gives its generation component the meaning representation above, it expects the paraphrase, "Is it raining in Boston?", and the SQL query:

```
select distinct geography.apt_code, city, state, country, source, day,
dayspeak, rainspeak from weather.event, weather.geography where
geography.city = 'boston' and rainspeak is not NULL and
weather.event.apt_code = geography.apt_code
```

- GALAXY also uses its generation component to flatten hierarchical meaning representations into *e*-forms. Consider this example from the MERCURY domain:

```
{c wh_query
   :mode "finite"
   :num "sing"
   :aux "do"
   :topic {q flight
             :quant "def"
             :pred {p airline
                      :topic {q airline_name
                                :name "united" } } }
   :pred {p arrival_time
```

```
:topic {q time
          :quant "which"} } }
```

Given this meaning representation, the generation component would be expected to output the following *e*-form:

`airline: UA arrival_time: which.`

- For our final example, we present a complex meaning representation from the rich JUPITER weather domain:

```
{c weather_event
   :topic {q weather_act
             :name "cloud"
             :number "pl"
             :and {q precip_act
                     :temp_qualifier "spotty"
                     :name "drizzle" }
             :pred {p time_interval
                      :topic {q time_of_day
                                :name "morning" } } }
   :and {c weather_event
           :conjn "then"
           :and {c weather_event
                   :conjn "with"
                   :topic {q weather_act
                             :quantifier "some"
                             :name "sun"
                             :pred {p time_interval
                                      :topic {q time_of_day
                                                :name "afternoon" } } } }
           :pred {p becoming
                    :topic {q weather_act
                              :conditional "unseasonably"
```

`:name "mild" } } } }`

The correct output for this meaning representation is `"morning clouds and spotty drizzle then becoming unseasonably mild with some afternoon sun"`.

Chapter 4

Overview of Genesis-II

In Chapter 1, we discussed our motivation for building GENESIS-II and some of the design considerations that arose during its implementation. In Chapter 2, we placed GENESIS-II in context by discussing the broad field of language generation, and, in Chapter 3, we further situated GENESIS-II by describing the role of generation in the specific context of the GALAXY System. In this chapter, we increase magnification yet again, with an overview of the GENESIS-II system itself. In the first section of this chapter, we outline the system's architecture and introduce the system's linguistic components. In the next section, we describe the general algorithm with which GENESIS-II converts a meaning representation into a string. We begin that section with a discussion of the structure that represents the linguistic environment during the generation process. We then describe the process itself and carefully trace generation in a simple example.

4.1 System Architecture

The high-level architecture of GENESIS-II consists of a kernel and a linguistic catalog, as depicted in Figure 4-1. The linguistic catalog is a set of knowledge bases for various domains and languages. Each knowledge base has three components—a lexicon, a grammar, and a list of rewrite rules—which together specify generation for a particular domain and language. The kernel is the core of C code that, at run time, converts a meaning representation into a string by accessing the knowledge base for the given domain and language. For example, suppose the kernel receives a meaning representation, along with the information that the meaning representation is from the JUPITER weather domain and that the target string

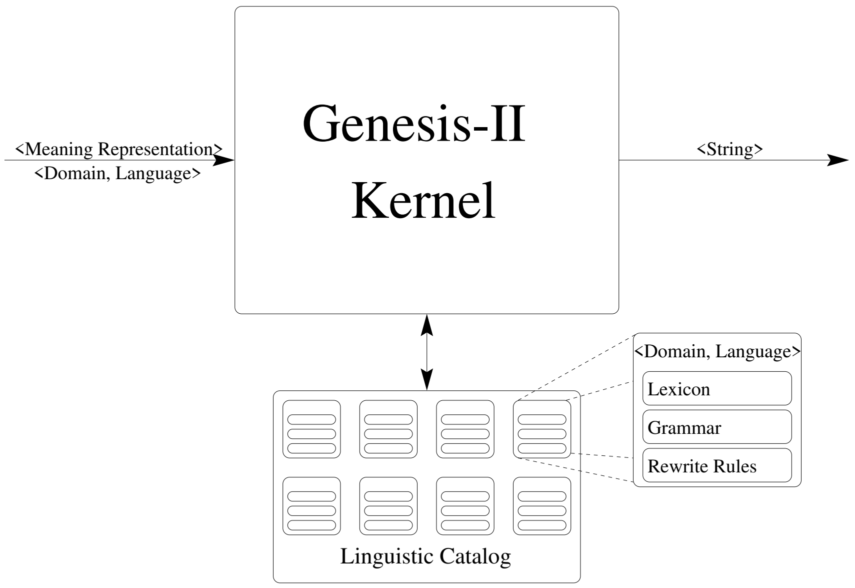

Figure 4-1: GENESIS-II's architecture.

should be in Spanish. The kernel would first search the linguistic catalog for a JUPITER Spanish knowledge base, and, upon finding one, the kernel would use it for generating an appropriate target string for the meaning representation.

The lexicon, grammar, and rewrite rules of a particular knowledge base are defined by the contents of three text files—the vocabulary file, the grammar file, and the rewrite-rule file. Domain and language experts create these files in accordance with the GENESIS-II framework for specifying generation. In the next two chapters, we shall discuss in depth exactly what commands domain experts may include in knowledge-base files. We begin, however, with a brief look at the three components and the general form of the text files that define them.

4.1.1 Lexicon

The lexicon is a set of vocabulary items, each of which contains linguistic information on a particular vocabulary word. The amount of information varies from item to item. For example, in Spanish, noun and adjective entries often contain gender information. In several languages, special pluralized forms are included in vocabulary entries, as well. Regardless, every vocabulary word has at least a part of speech and a default generation string. Let us consider the following two simple vocabulary items from the JUPITER weather domain in

English:

```
thunderstorm          N "thunderstorm"

trade_wind            N "trade wind"
```

The first item states that the vocabulary word "thunderstorm" has the part of speech N (noun) and that the default generation string for "thunderstorm" is "thunderstorm". The second example is similar, but note that the underscore in the vocabulary word is not present in its default generation string. The role of the part of speech (and other forms in the lexicon) will become clear in the next chapter, when we discuss the lexicon in greater detail.

4.1.2 Grammar

The backbone of every knowledge base is its grammar. Essentially, the grammar for a particular domain and language is a list of rules that specify generation for all possible inputs to GENESIS-II in that particular context. Of course, an infinite number of inputs to GENESIS-II is possible, even within the confines of a specific domain and language. However, GENESIS-II combats the problem of combinatorial explosion in several ways. Like the template-based generator from IBM that we discussed in Chapter 2, GENESIS-II has a notion of turning subphrases on and off. Additionally, grammar rules in GENESIS-II can contain recursive calls and "subroutine" calls. We shall discuss all of these mechanisms later in the thesis.

A grammar rule consists of two parts, the rule name and the rule body. We illustrate the form of grammar rules by providing a simple example—a "toy rule" for time_of_day[1]:

```
time_of_day            :name
```

The rule specifies that, when generating a string for a frame named time_of_day, GENESIS-II should select from the frame the key value for the keyword :name. If the value is a string, GENESIS-II should look up the key value in the lexicon and generate the target string for it accordingly. For an example of this rule's application, let us revisit a meaning representation we examined in Chapter 3. For convenience, we include it again as Figure 4-2. Given this

[1]We refer to the rule in that way, because the actual rule for time_of_day is more sophisticated.

```
{c weather_event
   :topic {q weather_act
             :name "fog"
             :pred {p time_interval
                      :topic {q time_of_day
                                :name "morning" } } } }
```

Figure 4-2: The meaning representation for "morning fog".

meaning representation and the rule stated above, GENESIS-II would convert the nested frame named time_of_day into the generation string specified by the vocabulary entry for "morning". In the next chapter, we will describe exactly how GENESIS-II uses the lexicon to generate a string. We shall also describe in more detail the keyword-generation construct introduced above, as well as the many other grammar constructs for specifying generation.

We should note that the order of items in the vocabulary file and rules in the grammar file is, for the most part, unimportant.[2] After GENESIS-II has read the vocabulary file and grammar file, it automatically alphabetizes the items and rules in the lexicon and grammar respectively, to enable GENESIS-II to efficiently look up a string in either of the components by performing a binary search.

4.1.3 Rewrite rules

The final linguistic component we examine is the set of rewrite rules. Aptly, it is the final linguistic component to have an effect on the target string during the generation process. After the system has used the lexicon and grammar to generate a preliminary target string, it uses the rewrite rules to refine the target string. Each rewrite rule contains two strings; the left-hand string specifies the pattern to be matched in the preliminary target string, and the right-hand string specifies a replacement for the pattern. For example, a target string may contain such ungrammatical phrases as "a eastern wind" or "de el", a Spanish phrase that is always contracted to "del". The following two rules correct the problems respectively:

```
" a e"                   " an e"
```

[2]In Chapter 6, we shall discuss the capability to define multiple, "alternate" entries for the same string. When specifying alternate entries, order *does* matter.

```
" de el "              " del "
```

In this final stage of generation, GENESIS-II walks through the set of rewrite rules, in the order in which they appear in the rewrite-rule file, and applies each of them repeatedly to the preliminary target string until it no longer applies. The system then moves on to the next rule. Therefore, in this component, unlike the lexicon and the grammar, order does matter.

We encourage domain experts to specify a minimal number of rewrite rules and to use them exclusively for fixing surface problems, like those in the examples. Whenever possible, domain experts should specify correct generation with the sophisticated mechanisms permitted in the lexicon and grammar, rather than with the primitive pattern-matching facility of the rewrite rules.

4.2 Generation Algorithm

In this section, we describe GENESIS-II's algorithm for converting a meaning representation into a target string. We begin with a lengthy description of the "`info frame`", GENESIS-II's mechanism for transmitting linguistic information among the components of the meaning representation. We then illustrate the process of generation by tracing the conversion of the meaning representation in Figure 4-3 into the target string `"some showers"`. Again, for the sake of simplicity, we use toy rules in our examples.

It is nearly impossible to give the reader a general, but useful, sense for the generation process, without bringing in specific lexical and grammatical forms that we will present in detail later in the thesis. However, in order to keep the present discussion focused on the general algorithm GENESIS-II employs, our examples will only briefly introduce necessary forms and their effects, with the implicit promise to cover them systematically and in more depth in one of the subsequent chapters.

4.2.1 Info frame

When GENESIS-II receives a meaning representation, its first action is to create an `info frame`. During the generation process, GENESIS-II uses this structure to pass linguistic information among the constituents of the meaning representation. The system always uses the `info frame` to pass gender and number information, but, as we shall see in the next two

```
{c weather_event
   :topic {q precip_act
             :quantifier "some"
             :name "shower"
             :number "pl" } }
```

Figure 4-3: The meaning representation for `"some showers"`.

```
{q info
   :gender "f"
   :number "pl"}
```

Figure 4-4: A simple `info frame`.

chapters, domain experts can extend the `info frame` to convey other linguistic information, as well. Figure 4-4 contains a simple `info frame`, with gender and number information.

Motivation

Before describing the way in which information is added to the `info frame` and the way in which information is referenced, let us motivate the necessity for an `info frame` with a linguistic scenario. Suppose we wish to generate a string in Spanish for the meaning representation in Figure 4-3. In this scenario, the frame named `precip_act` establishes two important pieces of linguistic information. First, the key pair [`:number "pl"`] indicates that the noun phrase is plural. Secondly, in the Spanish lexicon, `"shower"` translates to `"aguacero"`, a masculine noun. The quantifier `"some"` modifies `"shower"`, and, in Spanish, an adjective must agree with the noun it modifies in both number and gender. Therefore, the gender and number information must propagate to `"some"`. The `info frame` serves as the conduit of such information.

Updating the info frame

Let us now examine the way in which GENESIS-II automatically updates the `info frame` with gender and number information. GENESIS-II always attempts to update the `info frame` when it receives a frame to process. The way in which GENESIS-II modifies the `info frame` depends upon the frame's type. The system handles topic frames in one way and predicate and clause frames in another way. We begin by considering topic frames.

Topic frames

In the GALAXY System, a topic frame represents a noun phrase. Linguistically, a noun phrase is relatively independent of higher-level constituents. That is, a noun phrase generally determines its own linguistic environment, including gender and number information. Therefore, when GENESIS-II processes a topic frame, it begins by removing all gender and number information from the `info frame`.

GENESIS-II then attempts to update the gender and number information with the linguistic properties of the topic frame's "head." In GALAXY, the head of a topic frame is a key value string for the keyword `:name`. If the topic frame does not contain the keyword `:name`, then the head is the frame's name. For example, in Figure 4-3, the head of the topic frame named `precip_act` is the string `"shower"`. Once GENESIS-II has determined the topic frame's head, it searches the lexicon for the head and uses the matching vocabulary item to update the `info frame`.

To set the gender, GENESIS-II looks for the string `"G"` in the vocabulary item. If it is found, GENESIS-II interprets the string that follows to be the gender and modifies the `info frame` to reflect the new gender. For example, in the following vocabulary item, the gender is `m` (`masculine`):

```
shower                  N "aguacero" G "m"
```

To set the number, GENESIS-II first searches the topic frame (not the vocabulary item) for the keyword `:number`. If it finds the keyword and if the key value is a string, GENESIS-II interprets the string to be the number and updates the `info frame` accordingly. Otherwise, it searches the vocabulary item for the string `"num"`. If the item contains the string `"num"`, GENESIS-II interprets the string that follows as the number and modifies the `info frame` appropriately.

Clause and predicate frames

If the frame is a clause or a predicate, on the other hand, GENESIS-II does not begin by automatically deleting the old gender and number information. Linguistically, clauses and predicates tend to be dependent upon higher-level constituents. As a consequence, GENESIS-II saves the old gender and number information in the `info frame` and replaces it only if necessary. GENESIS-II's first action upon receiving a clause or predicate frame, therefore, is to search for the frame's head. As you may recall, in the case of topic frames, the system preferred to designate as head the key value associated with `:name`. However, clause and predicate frames do not typically contain the keyword `:name`. Therefore, when processing a frame of one of these types, GENESIS-II assumes that the frame's name is the head. GENESIS-II looks up the head in the lexicon, and, if successful, it uses the matching vocabulary item to set the linguistic information by following the same algorithm outlined above. That is, for gender, GENESIS-II defers to the vocabulary item, and, for number, GENESIS-II first searches the frame and then defers to the vocabulary item.

Utilizing the info frame

During vocabulary generation, GENESIS-II automatically references the `info frame` to obtain gender and number information. When a vocabulary item contains several possible generation strings, GENESIS-II uses the `info frame` to select the most appropriate form for the current linguistic environment. For example, consider the following vocabulary entry from the Spanish lexicon:

```
some                     "algun" M "algun" F "alguna" PL_F "algunas"
                         PL_M "algunos"
```

The default generation string in Spanish for the English word `"some"` is `"algun"`. However, if a gender or number is specified, the generation string may be different. For example, if the gender is `F` (`feminine`) and no number is specified, the generation string is `"alguna"`. If the gender is `M` (`masculine`) and the number is `PL` (`plural`), the generation string is `"algunos"`.

Consider again the scenario in which we were generating a Spanish string for the meaning representation in Figure 4-3. Given our algorithm for updating the `info frame`, we know that GENESIS-II updates the `info frame` when it receives the `precip_act` frame for processing. As mentioned before, the gender of the Spanish word for `"shower"` is masculine, and we know from the contents of the `precip_act` frame that the number of this noun phrase is plural. Therefore, by the time GENESIS-II generates vocabulary for `"some"`, the `info frame` contains the following information:

```
{q info
   :gender "m"
   :number "pl"}
```

Given the linguistic information in this `info frame`, GENESIS-II would be able to select from the vocabulary entry the generation string `"algunos"`, the form of `"some"` that agrees in gender and number with `"aguaceros"`, the Spanish noun for `"showers"`.

The info frame's scope

An important issue the reader might be considering at this point is the matter of scoping, since we have not yet specified which constituents are affected by modifications to the `info frame`. The scoping rule is actually quite intuitive and simple. Generally speaking, a lower-level constituent should not affect a higher-level constituent linguistically. Therefore, GENESIS-II makes a fresh copy of the `info frame` each time it begins processing a frame. In this way, all modifications to the `info frame` do not affect the parent frame.

As you might imagine, there are exceptions to the rule. That is, there are situations in which a domain expert may wish to specify propagation from child to parent. Later in the thesis, we shall discuss GENESIS-II mechanisms for achieving backwards propagation.

4.2.2 Generation process

Having completed our discussion of the `info frame`, we now trace the process in which GENESIS-II converts a meaning representation into a target string. As an example, we shall follow GENESIS-II's conversion of the meaning representation in Figure 4-3 into the English string `"some showers"`. Again, we remind the reader that, for the sake of simplicity, we use toy rules in our examples. Furthermore, all lexical and grammatical forms briefly presented in this discussion will be described in much more detail in the next two chapters.

As discussed above, GENESIS-II's first action upon receiving a frame for processing is to update the `info frame`. Afterward, GENESIS-II searches the grammar for a rule specifying how to generate a string from the top-level frame. For now, let us assume that GENESIS-II searches exclusively for a rule with the same name as the top-level frame. (We soon shall see that there are other ways of specifying and determining generation rules for frames.) In our example, GENESIS-II searches for a rule named `weather_event` and finds the rule:

```
weather_event           :topic
```

The string `:topic` in the `weather_event` rule indicates that to generate a string for a `weather_event` frame GENESIS-II must generate a string for the keyword `:topic`. Therefore, GENESIS-II must search for the keyword `:topic` in the `weather_event` frame. In our example, the `weather_event` frame does indeed contain a `:topic`, whose key value is a frame named `precip_act`. GENESIS-II processes the `precip_act` frame in the same way it processed the `weather_event` frame. The system begins by updating the `info frame` with the linguistic information conveyed by the topic frame named `precip_act`. As described earlier in this section, this entails deleting the old gender and number information and replacing it with the current gender and number information. There is no gender information in our English example. However, the `precip_act` frame contains a key value string for `:number`, and so, GENESIS-II updates the number accordingly in the `info frame`. GENESIS-II then searches the grammar for a rule named `precip_act`. Suppose, the rule GENESIS-II finds is as follows:

```
precip_act              :quantifier :name
```

GENESIS-II must first generate a string for the key value of `:quantifier`. In our example, the key value of `:quantifier` is `"some"`, which has the following entry in the English lexicon:

```
some                    A "some"
```

Since there are no special forms, GENESIS-II simply uses the default string, `"some"`. According to the grammar rule for `precip_act`, GENESIS-II should now generate a string for the key value of `:name`. In the `precip_act` frame, the key value of `:name` is `"shower"`, whose entry in the lexicon is as follows:

```
shower                  N "shower" PL "showers"
```

Since the `info frame` specifies that the noun phrase is plural, GENESIS-II selects the generation string `"showers"` from the vocabulary item.[3] The substrings `"some"` and `"showers"` are concatenated to form the string `"some showers"`, and the generation process is complete.

[3] In the next chapter, we shall explain how the part-of-speech information can be used for suffix selection.

Chapter 5

Basic Generation in Genesis-II

In this chapter, we continue our examination of GENESIS-II, by systematically describing the lexical and grammatical commands that domain experts can use to specify generation in GENESIS-II. Because the GENESIS-II framework is rather expansive, we shall reserve a discussion of the more sophisticated generation techniques until the next chapter. Therefore, this chapter will contain the admittedly less interesting commands. However, we must lay the groundwork for the more advanced techniques, and so, the contents of this chapter are essential.

This chapter has two sections; one focuses on the lexicon, and the other focuses on the grammar.[1] In the first section, we flesh out our previous descriptions of the lexicon. Because we described many of the lexicon's basic features in Sections 4.1.1 and 4.2, this section is relatively concise. In the second section, we describe the basic grammar commands permissible in the GENESIS-II framework. Since the grammar is such a complex and powerful component, this section will be considerably longer than the previous section.

5.1 Lexicon

As mentioned in Section 4.1.1, every entry in the lexicon has at least three distinct components: a vocabulary word, its part of speech, and a default generation string for it. The first string in a vocabulary item is considered to be the vocabulary word, by which the entry is indexed. The second string is the part of speech, and the third string, which should

[1]We exhausted the subject of rewrite rules in Section 4.1.3.

appear in double quotes, is the default generation string. For example, the following is a vocabulary entry from the Spanish lexicon:

```
beach                    N "playa"
```

In this rule, the vocabulary word is "beach", the part of speech is N, and the default generation string is "playa".

The part of speech is primarily used to group words with common suffixes. For example, the entry in the JUPITER Spanish lexicon for the part of speech N is:

```
N                        N "N" PL "s"
```

In this entry, the form [PL "s"] specifies that, to construct the plural of all words with part of speech N, one simply needs to add the letter "s". To further tease out this concept, consider the following entries from the same lexicon:

```
A1                       A "o" M "o" F "a" PL_M "os" PL_F "as" PL "es"

another                  A1 "otr"

dangerous                A1 "peligros"
```

Both "another" and "dangerous" have the part of speech A1 (adjective, type 1). The entry for the part of speech A1 specifies how to inflect words with that part of speech, given linguistic information such as number and gender. For example, if the word is masculine plural, then the ending should be "os" ("otros", "peligrosos"). If the word is feminine plural, then the ending should be "as" ("otras", "peligrosas").

Now consider this example from the English lexicon:

```
A                        A "A" PL ""

cloudy                   A "cloudy"

some                     A "some"
```

Both "cloudy" and "some" have the part of speech A (adjective). The entry for the part of speech A specifies that to form the plural of all words with part of speech A, one simply needs to append the empty string, *i.e.,* do nothing. Given these rules, GENESIS-II would never output the strings `"cloudys skies tomorrow"` and `"somes showers expected"`.

Any vocabulary item can specify a pluralization rule directly, as well. For example, the plural form `"these"` cannot be constructed by adding a suffix to the singular form `"this"`. Therefore, the method described above for capturing suffixal similarities does not apply in this situation, and so, the vocabulary entry of `"this"` should be as follows:

```
this                    A "this" PL "these"
```

The `PL` specification in the vocabulary item for `"this"` will override the `PL` specification in the vocabulary item for the part of speech `A`.

We should note that none of the parts of speech and gender designations are "hard-wired." That is, if a domain expert wanted to use the letter `S` to specify nouns or the letter `X` to designate masculine forms, she could do so. Such choices can be completely arbitrary. For the system to work, of course, there must be internal consistency. That is, if all nouns have part of speech `S`, then there should be a vocabulary item for `S`, since the `N` item no longer applies.

GENESIS-II has mechanisms for inflecting not only nominal and adjectival forms, but also verbal forms. In particular, GENESIS-II searches the `info frame` for the keyword `:mode` and uses the key value to select the appropriate verbal form from a lexical entry. This mechanism can be used, for example, to allow an auxiliary verb to control the mode of a main verb. In Section 6.4 of the following chapter, we shall describe how a vocabulary item or grammar rule might set the `:mode`, and at that time, we shall illustrate this concept with a small example. The next chapter also includes descriptions of the other advanced lexical features, including the selector mechanisms.

5.2 Grammar

The rest of this chapter and most of the next will focus on the grammar, the core component of each GENESIS-II knowledge base. In this section, we describe the basic commands domain experts can use to specify generation in grammar rules. For each command, we present a general description, followed by some examples of the command in action. For some of the commands, we also include a few suggestions on how to use them.

5.2.1 Special forms

In the first part of this section, we discuss a few "special" commands. We group these commands because they are similar in many ways. For one, each of these commands has only one form. That is, throughout this chapter we will look at commands that are identified by certain characteristics but that appear in the grammar file in many forms. For example, the commands, `:key1` and `:key2`, are both `keyword` commands (Section 5.2.5), even though they are not identical. Another example would be the strings, `"rain"` and `"snow"`. They are both instances of the `string` command (Section 5.2.2), but they are not identical. "Special" commands, on the other hand, do not change. The `core` command, for example, always has the form `$core`, and the `time` command always has the form `$time`. Special commands also share a common delimiter—the dollar sign. Finally, all of these commands are relatively simple. Subsequently, the descriptions that follow will not include examples.

Capitalization

There are three `capitalization` commands, each of which consists of a parenthesized list. The first string in the list uniquely specifies which of the three `capitalization` commands it is, and the other strings in the list are the commands to which the capitalization rule applies. We define the three types of `capitalization` commands—`$caps`, `$cap`, and `$cap1`—as follows:

- The `$caps` command indicates that GENESIS-II should capitalize every letter of the full string generated by the commands in the parenthesized list.

- The `$cap` command indicates that GENESIS-II should capitalize only the first letter of every whitespace-separated string generated by the commands in the parenthesized list.

- The `$cap1` command indicates that GENESIS-II should capitalize the first letter of the full string generated by the commands in the parenthesized list.

Core

The `core` command has the form, `$core`, and it indicates that GENESIS-II should generate vocabulary for the *name* of the current frame.

Rest

The `rest` command has the form `$rest`, and it indicates that GENESIS-II should search the current frame's predicate list and generate strings for all predicates that have not been processed yet. The strings are concatenated in the order in which the predicates appear in the predicate list.

Time

The `time` command has the form, `$time`, and it specifies that GENESIS-II should preprocess the current frame, which is presumably a time frame. Specifically, if the frame contains information in military time, GENESIS-II will convert it to hours and minutes by adding the key pairs for `:hours` and `:minutes` to the frame. GENESIS-II also adds a key value for `:xm` to indicate whether it is `am` or `pm`. Finally, regardless of whether the information was in military time, GENESIS-II concatenates the hour and minute information and adds it as the key value for `:o+clock`. The domain expert can then arrange the commands for generating strings for the `:hours`, `:minutes`, `:o+clock`, and `:xm` keywords.

5.2.2 Strings

GENESIS-II interprets any string in double quotes to be a `string` command. When GENESIS-II encounters an instance of this command, it strips the quotation marks from the string and adds it "as is" to the full generation string.

Domain experts may take a couple of shortcuts in specifying instances of this command. For one, GENESIS-II interprets any string that begins with a character from the extended character set to be an instance of the `string` command. Thus, quotes are unnecessary in Japanese, Chinese, etc. Furthermore, commas, periods, and question marks are always interpreted as instances of the `string` command. Therefore, quotes are unnecessary around these items, as well.

Examples

- `thanks_very_much        "Thank you very much"`

  When GENESIS-II uses this rule, the generation string is simply `"Thank you very much"`.

- `todays_date`   `"Today is" :topic .`

  This rule specifies that Genesis-II should concatenate the string `"Today is"` with the string generated by issuing the command `:topic`. It also specifies that the full generation string should end with a period.

5.2.3 Lookup

The `lookup` command is defined as any string delimited by an exclamation point. This command indicates that Genesis-II should search for the string in the lexicon and generate vocabulary for it, by using all known linguistic information. If the string is not in the lexicon, Genesis-II uses the string as the root and generates vocabulary for it. Once Genesis-II has generated vocabulary, it adds the generated word to the full generation string.

Examples

- `summarize_one_fare`   `:itineraries !is_good`

  This rule from the Mercury domain specifies that Genesis-II should issue the `:itineraries` command. It should then look up the entry for `"is_good"` in the lexicon and generate vocabulary. Finally, Genesis-II should concatenate the string generated by issuing the `:itineraries` command with the string generated by issuing the `lookup` command.

Usage

We encourage domain experts to use the `lookup` and `$core` commands instead of the `string` command, in order to compartmentalize the functions of the grammar and the functions of the lexicon as much as possible. Not only is modularity a good design paradigm, but, in this case, the practical benefits are immediately evident, as well. Suppose a domain expert were working on knowledge bases for two languages with identical syntactic structures but different vocabularies. If the grammar file were free of `string` commands, she could specify one grammar file to be used with two vocabulary files. Admittedly, such pairs of languages are rare, but, in a conversational system, they do occur. For example, in the Galaxy System's Mercury (travel) domain, the English text and speech "languages" have a common syntax but different lexical realizations. That is, the text vocabulary file contains English words, and the speech vocabulary file contains a mixture of prosodically-marked strings and waveform pointers. However, because of the common syntax, a shared grammar file works perfectly and avoids unnecessary duplication.

For example, the MERCURY English grammar file contains the following grammar rule:

```
speak_one_connecting    :comment_tlist !no_nonstops :on_date .
                        !i_have_a ($if :connection_list "" !connecting)
                        :airline !flight >connection_place >leaving_at
                        !and_arriving_at :arrival_time :arrive_xm
                        :days_later .  >main_preds ?  !would_that_work
```

This rule contains several `lookup` commands (in addition to several commands that we have not yet introduced). By using `lookup` commands instead of `string` commands, the domain expert has ensured that the vocabulary file alone will determine the lexical realization of the string generated by this rule. Let us examine a specific `lookup`—namely, the command `!connecting`. In the MERCURY lexicon for English speech, the entry for `"connecting"` is:

```
connecting              0 "[ mercury/mercury079 54200 62481 1 1 1
                        connecting ]"
```

Therefore, when GENESIS-II executes the `speak_one_connecting` rule in the English speech domain, the target string for `!connecting` will be the default string in the vocabulary item above. On the other hand, the lexicon for English text contains no entry, and so in that domain, GENESIS-II will correctly assume the root to be the string `"connecting"`.

5.2.4 Gotos

One of the most important commands in the GENESIS-II framework is the `goto` command. GENESIS-II considers a command delimited by a greater-than sign to be a `goto`. Such commands have two effects. Primarily, they serve as a signal to GENESIS-II to descend into another grammar rule. Their secondary purpose is to add a method of backing-off for the frame's children. We shall reserve a discussion of backing-off for Section 6.3 of the following chapter, because it is one of GENESIS-II's more sophisticated features. For now, we shall examine only the primary effect of the `goto`.

When GENESIS-II encounters a goto, it strips the ">" from the command and searches for the resulting string in the grammar. If GENESIS-II finds a matching rule, it descends into the rule and continues adding to the generation string by executing the commands in the new rule. After GENESIS-II has executed every command in the new rule, it returns to where it left off in the original rule and continues generation at that point. Once finished, GENESIS-II concatenates the following three substrings:

- the substring generated in the original rule before the goto,
- the substring generated in the rule specified by the goto, and
- the substring generated in the original rule after the goto.

Examples

- `weather_event` `>prefix :topic :aux >main_preds :and :or`

  `prefix` `:expl :city :day :day_tlist :conjn :adverb :conditional :qualifier`

  The weather_event rule contains two instances of the goto command. The first instance, >prefix, indicates that GENESIS-II should search the grammar for a rule named prefix and continue generation there. Thus, GENESIS-II would descend into the prefix rule and generate all of the commands it contains. After completing the prefix rule's final command, :qualifier, GENESIS-II would leave the prefix rule and continue generation where it had suspended execution in the weather_event rule. It would therefore execute the :topic and :aux commands, and then it would execute the other goto command, >main_preds. Finally, after returning from the execution of the main_preds rule, GENESIS-II would execute the weather_event rule's last two commands and then concatenate all of the generation strings to form the full generation string for weather_event.

- `have_sun_rise` `>sunrise_info >info_pred`

  `info_pred` `(>main_preds "nearby") "colon" :city_tlist :db_tlist :continuant`

We include these two rules to illustrate that the rule specified by a goto may itself contain goto commands (and every other type of command). In this example, the first rule contains the goto command, >info_pred, and the second rule, the rule for info_pred, contains yet another goto. Theoretically, any number of nested goto commands is permissible, but domain experts should be careful not to create an infinite loop of goto commands.

Usage

Domain experts find the goto command useful for several reasons. As mentioned briefly, the goto command is essential in creating back-offs for a frame's children. They can also be used to reduce redundancy in grammar files. Suppose, for example, that two rules were identical except for their first commands. Two such rules might be:

```
comment                 :topic not_in >prepreds >postpreds

unknown_city            :name not_in >prepreds >postpreds
```

Instead of duplicating the command list [not_in >prepreds >postpreds], a domain expert might consider creating a third rule of the form:

```
main_preds              not_in >prepreds >postpreds
```

and modifying the other two rules to be:

```
comment                 :topic >main_preds

unknown_city            :name >main_preds
```

Such redundancy often exists in rule files, with identical command lists' appearing in a handful of rules. Therefore, the goto command plays a significant role in reducing the work of Genesis-II domain experts, as well as the size of grammar files. Additionally, goto commands make grammar files more readable.

5.2.5 Keywords

A command delimited by a colon is a `keyword` command. When GENESIS-II encounters a `keyword` command in a grammar rule, it first searches the grammar for a rule that shares its name with the keyword. If such a rule is found, GENESIS-II descends into it and continues generation within it. In this situation, GENESIS-II's behavior is identical to its behavior upon encountering a `goto` command, in every way but two. First, the back-offs briefly mentioned in association with the `goto` commands do not apply to the `keyword` commands. Secondly, if GENESIS-II encounters the command `$value` within a keyword rule, GENESIS-II interprets it as a command to locate the keyword and its value in the current frame it is processing. GENESIS-II then processes the key value based on its type and generates a string for it, as we shall discuss momentarily.

If GENESIS-II cannot find a rule that matches the keyword in the `keyword` command, there are several other ways it can find a rule with which to generate a string for the keyword. We shall discuss them in the following chapter. For now we simply state that if no rule is found, GENESIS-II locates the keyword and its value in the frame it is processing, generates a string for the key value, and adds the string to the full generation string. In other words, every keyword has an implicit rule with the body, `$value`.

As mentioned above, the way in which GENESIS-II processes the key value of a keyword depends upon the key value's type.[2] GENESIS-II has three different approaches—one for strings and integers, one for frames, and one for lists—as outlined below:

- **Strings and Integers:** GENESIS-II processes a string or integer key value by looking it up in the lexicon and generating vocabulary for it. If the value is not in the lexicon, GENESIS-II generates it "as is."

- **Frames:** GENESIS-II processes a frame key value in the same way it processes all frames (Section 4.2). In short, GENESIS-II updates the `info frame` (Section 4.2.1) and then uses the appropriate grammar rule to generate a string for the frame (Section 4.2.2).

- **Lists:** The ability to generate a string from a list is one of GENESIS-II's more sophisticated features. Therefore, we shall describe the process associated with lists in the

[2] For a description of the four types, refer to Section 3.2.3.

```
{c weather_event
   :topic {q weather_act
             :name "fog"
             :pred {p time_interval
                      :topic {q time_of_day
                               :name "morning" } } } }
```

Figure 5-1: The meaning representation for "morning fog".

following chapter.

We should also note that once GENESIS-II generates a string for a particular keyword in a frame, it marks the keyword "silent" and will not generate another string for it. For example, consider the following toy grammar rule:

```
grammar_rule          :key1 :key1
```

Suppose GENESIS-II were using this rule to generate a string for a frame that contained a key pair for the keyword `:key1`. When GENESIS-II encounters the first keyword command in the rule, it processes the key value of `:key1` and marks the keyword silent. Therefore, when GENESIS-II encounters the second keyword command in the rule, the frame no longer contains a legitimate key value for `:key1`, and so, the generation string is not duplicated.

Examples

To illustrate the `keyword` command, we revisit the now-familiar meaning representation for "`morning fog`". We first examined this frame in Section 3.2, and we considered it again in Section 4.1.2. For convenience, we include it again here, as Figure 5-1.

- `time_of_day`   `:quantifier :modifier :name`

  Suppose GENESIS-II were using this rule to generate a string for the `time_of_day` frame in Figure 5-1. When executing the first two commands, it would find that the `time_of_day` frame does not contain the keywords `:quantifier` and `:modifier`. Therefore, it would proceed to the third command, the `keyword` command, `:name`. To execute this command, GENESIS-II would search the grammar for a rule named `:name`. Suppose that no such rule existed; in this situation, GENESIS-II would infer the implicit `$value` rule. It would then search the `time_of_day` frame for the keyword `:name` and find it. In the `time_of_day` frame, the key value of `:name` is the string `"morning"`, and so, GENESIS-II would perform a lookup and add the result to the target string.

- `:name` `($let ^:pos :pos[$core]) $value`

  Suppose that in the previous example, GENESIS-II had found the rule above for `:name`. The first command in the rule is a `let`, which we shall discuss in detail in Section 6.4 of the following chapter. For now, we simply say that this command indicates to GENESIS-II that it should set a "local variable" that contains linguistic information. The second command, `$value`, indicates that GENESIS-II should generate the key value for `:name` in the current frame, given the updated linguistic environment. A special rule for `:name` is necessary in this situation, because the `let` establishes important linguistic information that the default `$value` does not establish.

Usage

In practice, domain experts have found that it is usually sufficient to rely on the default `$value` generation, and so, specialized rules for keywords are often unnecessary.

5.2.6 Or

GENESIS-II provides domain experts with two commands for conditional generation: the `or` and the `if`. We begin with the `or` command, which, syntactically, is a parenthesized list of commands. However, we must be careful in designating parenthesized lists as `or` commands, because, as we have seen in this chapter and as we will see in the next, other commands are parenthesized lists, as well. In every case, though, GENESIS-II can distinguish commands of this form by the first string in the list. For example, the `if` command is also a parenthesized list, but the first string in the list is always `$if`. GENESIS-II identifies as `or` commands all parenthesized lists that do not fall into one of the other categories.

Semantically, an `or` is an indication to GENESIS-II that only one of the commands in the list should contribute a substring to the full generation string. That is, when GENESIS-II encounters an instance of the `or` command, it should proceed through the list of commands and cease generation when one of the commands produces a generation string.

Examples

- `out_of_domain` `"I currently don't have knowledge about" (:topic "that").`

  In this example from the JUPITER weather domain, GENESIS-II first attempts to generate a string for the `keyword` command `:topic`. If unsuccessful, GENESIS-II settles for the vague pronoun `"that"`.

- `weather_template` `>prepreds (:name $core) :units >and_or`

  This example contains a common `or` command in the English grammar files: `(:name $core)`. You might recall from our previous discussion of the `info frame` (Section 4.2.1) that GENESIS-II preprocesses topic frames differently from clause and predicate frames. For one, GENESIS-II searches topic frames for the keyword `:name` and, if possible, uses the lexical item associated with the key value to update the `info frame`. If the frame does not contain a `:name`, GENESIS-II settles for the vocabulary item associated with the frame's name. Similarly, this `or` command indicates that, if possible, GENESIS-II should generate a string from the key value of `:name` in the current frame. If unsuccessful, GENESIS-II should resort to executing the `$core` command.

5.2.7 If

The other conditional command is the `if` command. As mentioned in the description of the `or` command, the `if` is a parenthesized list, beginning with the string `$if`. The second part of the `if` command contains the constituent/s on which GENESIS-II must condition. The third part is the "then" consequent, *i.e.,* the command GENESIS-II executes if the condition is true. The fourth part, which is optional, is an "else" consequent, *i.e.,* the command GENESIS-II executes if the condition is false. In the next few paragraphs, we look at each of these parts in more detail.

Let us first examine the syntax and semantics of the conditional part of the `if` command. GENESIS-II always interprets the first string after the `$if` to be the first conditional. If the string is followed by two ampersands (`&&`), GENESIS-II infers that the `if` contains a series of ampersand-separated conditionals, all of which must be true for GENESIS-II to execute the "then" consequent. If, on the other hand, the string is followed by two bars (`||`), GENESIS-II infers that the `if` contains a series of bar-separated conditionals, one of which must be

```
($if :key1[:key2] :key3 :key4)
($if pred1[:key1 :key2 pred2] >goto1 >goto2)
```

Figure 5-2: Two examples of bracketed `if` commands.

true for GENESIS-II to execute the "then" consequent. In other words, domain experts can "AND" and "OR" conditionals. If the string is followed by neither ampersands nor bars, GENESIS-II infers that the `if` contains one conditional, which must be true for GENESIS-II to execute the "then" consequent. We shall provide some examples after we have described the other components of the `if` command.

GENESIS-II divides conditional strings into two groups: those that contain square brackets and those that do not. If a conditional does not contain square brackets, GENESIS-II interprets it to be one of the simpler types, either a keyword or predicate `test`. In this case, GENESIS-II searches the `current frame` it is processing for a keyword or predicate that matches the `test`. If GENESIS-II finds a match, it considers the conditional to be true.

Conditionals that contain square brackets indicate that GENESIS-II should search deeper in the frame structure for the constituent on which to condition. The format of a bracketed conditional is always a bracketed list of predicate and/or keyword constituents, preceded by a predicate or keyword `test`. Figure 5-2 contains two artificial examples of `if` commands that contain conditionals with brackets. In the first example, the `test` is the constituent `:key1`, and the only bracketed constituent is `:key2`. In the second example, the `test` is the constituent `pred1`, and the bracketed constituents are `:key1`, `:key2`, and `pred2`. When GENESIS-II encounters a conditional with a bracketed portion, it searches for the `test` in the frame's children, as specified in the bracketed list. That is, GENESIS-II first searches the current frame for a keyword or predicate that matches a string in the bracketed list. If it finds a matching keyword and the key value is a frame or if it finds a matching predicate frame, GENESIS-II searches this child frame for the `test`. If it is present, GENESIS-II considers the conditional to be true.

Consider again the two example commands in Figure 5-2 . The first `if` command specifies that GENESIS-II should search the current frame for the keyword `:key2`. If the frame contains the keyword `:key2` and if the key value is a frame, then GENESIS-II should search the key value frame for the keyword `:key1`. If there is a match, the conditional is true, and GENESIS-II should execute the "then" consequent, `:key3`. Otherwise, GENESIS-II should execute the "else" consequent, `:key4`. The second example in Figure 5-2 indicates that GENESIS-II should search the current frame for the keywords `:key1` and `:key2` and for a predicate named `pred2`. If GENESIS-II finds one of the keywords and its key value is a frame or if GENESIS-II finds a matching predicate, it should search the child frame for a predicate named `pred1`. If successful, the conditional is true. We will consider a couple more examples shortly, after we finish describing the format of the `if` command.

The final components of an `if` command are the "then" consequent and the optional "else" consequent. Either consequent may be any command that contains no whitespace. For example, a `goto` command or a `keyword` command may appear in the body of an `if` command. On the other hand, an `or` command or another `if` command cannot. However, `goto` commands can be used in place of illegal commands to achieve the same effect. For example, suppose a domain expert wished a "then" consequent to be the string, `"Not currently available."` She could achieve this by specifying as the "then" consequent the `goto` command, `>avail_rule`, and by defining the rule `avail_rule` to be `"Not currently available."` So, the whitespace restriction on consequents has no effect on the power of the `if` command.

Examples

- `limited_future` `"I only have predictions" ($if :city "for") :city >upto_day`

  In this example, the word `"for"` only appears in the full generation string if the keyword `:city` is in the current frame. By using an `if` command, the domain expert guards again such ungrammatical generation strings as `"I only have predictions for up to Tuesday."`

- `current_humidity` `"The current humidity" ($if :city "in") :city "is" :humidity`

  This example is similar to the last one, in that the `if` command is used to ensure that a preposition is used only if it will indeed be followed by a noun. Given this rule for `current_humidity`, GENESIS-II might generate `"The current humidity is 86 percent"` or `"The current humidity in San Diego is 86 percent"`; an impossible generation string is "The current humidity in is 86 percent".

- `and` `>--and ($if :verb[:and] $:verb) !and --and`

  In this example, the `if` command contains a bracketed conditional. When GENESIS-II executes this command, it searches the current frame for the keyword `:and`. If successful and if the key value for `:and` is a frame, GENESIS-II searches the `:and` frame for the keyword `:verb`. If it finds a `:verb`, it executes the "then" consequent, `$:verb`—a construct we shall discuss in the next chapter.

- `wind_chills` `>topic_core :qualifier ($if from_degrees && to_degrees >--between) month_date by_time >postpreds`

  This example from the JUPITER Spanish grammar contains an `if` command with an "ANDed" set of conditionals, as indicated by the ampersands. GENESIS-II executes this command by searching for `from_degrees` and `to_degrees` in the current frame. If it finds both, GENESIS-II executes the "then" consequent, `>--between`. Otherwise, GENESIS-II continues executing the other commands in the rule.

- `loc` `$:loc :conditional :loc_qualifier :particle ($if :particle || :loc_qualifier >loc1 >--loc) :topic direction >postpreds >and`

  Like the previous example, this rule is from the JUPITER Spanish grammar. In this case, the `if` command contains an "ORed" set of conditionals, as indicated by the bars. GENESIS-II executes this `if` command by searching for both `:particle` and `:loc_qualifier` in the current frame. If it finds either, GENESIS-II executes the "then" consequent, `>loc1`. Otherwise, if it finds neither, GENESIS-II executes the "else" consequent, `>--loc`.

5.2.8 Predicates

The final command we examine in this chapter is the `predicate` command. We reserved it for the end, despite its relative simplicity, because we cannot use positive attributes to describe its syntax. In other words, we can only say what it is not, and the system identifies it in a similar manner. That is, GENESIS-II determines that a command is a predicate command when it has ruled out all other possibilities. Therefore, it seemed more appropriate to introduce the predicate command after we had described a handful of basic commands.

When GENESIS-II concludes that a command is a predicate command, it searches the current frame's predicate list for a predicate whose name matches the predicate command. If it finds such a predicate, GENESIS-II processes it as it processes all other frames. (See 4.2 for details.) As in the case of keywords, once GENESIS-II has generated a string for a predicate, it marks the predicate "silent" and will not generate a string from it again.

Examples

- `measure_preds          high_value celsius_value low_value percent_chance`

  If GENESIS-II were using this rule to generate a string from a meaning representation, it would conclude that all four commands are predicate commands. Therefore, it would search the current frame for each of the four predicates and generate strings from them if possible.

5.2.9 Combinatorial explosion

Early in this thesis, we introduced the problem of combinatorial explosion in the context of IBM's template-based generator, primarily because their conference paper highlights their solution to the problem [Axe00]. We now present GENESIS-II's approach.

As described in Section 2.3.3, a template with n variables has 2^n realizations, depending on which variables have values. Therefore, if the framework of a particular generation system requires separate templates for each of the realizations, then the size of the corresponding rule libraries will be exponential in the number of variables. As a consequence, most generation systems have mechanisms for combating this type of combinatorial explosion. In Section 2.3.3, we described IBM's mechanism for "turning off" subphrases. In this section, we discuss our algorithm for "turning off" grammar rules.

Let us begin with the following example from a paper on GENESIS-II [BS00]:

```
flight_leg          >airline_flight >leaves_from

airline_flight      :airline "flight" :flight_number
```

```
leaves_from          "leaves" >from_source >at_dpt_time

from_source          "from" :source

at_dpt_time          "at" :depart_time
```

Suppose for a moment that GENESIS-II were using the `flight_leg` rule for generating from a frame that contained only the key pairs, { [`:airline "American"`], [`:flight_number` 998], [`:source "San Francisco"`] }. Given our description of the GENESIS-II framework thus far, the result would be the ungrammatical string, `"American flight 998 leaves from San Francisco at"`. However, in actuality, GENESIS-II would have "turned off" the `at_dpt_time` in this case, and it would have generated the *grammatical* string, `"American flight 998 leaves from San Francisco"`.

When we say that GENESIS-II "turns off" a grammar rule, we are referring to GENESIS-II's decision to exclude from the target string the string generated by executing the grammar rule. In the example above, GENESIS-II would have decided not to include the result of executing the `at_dpt_time` rule, and so, the target string would not have included the string, `"at"`.

GENESIS-II turns a grammar rule on only if it is able to generate a string for a "weighty" constituent, such as a keyword or predicate. There is also provision for rules that contain no weighty constituents. In this way, domain experts can express themselves concisely and thereby avoid the problem of combinatorial explosion. An explicit formulation of the rule is as follows:

> ***Rule:*** GENESIS-II turns the grammar rule on only if a command other than a `string` or `lookup` command produces a string.
>
> ***Exception:*** GENESIS-II always turns the rule on if it contains only `string` and `lookup` commands, as well as any of the commands that never produce strings (`set-selector` [Section 6.1]; `set`, `let`, `clone` [Section 6.4]; `push` [Section 6.6.2]).

Chapter 6

Advanced Generation in Genesis-II

In this chapter, we build on the description of GENESIS-II from the previous chapter, by focusing on the advanced features that make our system a formidable generator for conversational systems. Arguably, both GENESIS-II and its predecessor distinguish themselves simply by virtue of the fact that they are able to process hierarchical meaning representations. As you may recall, we examined three generators for conversational systems in Chapter 2, and, in each case, the generator was a non-linguistic system that was able to handle only flat sets of key-value pairs. In this chapter, we describe the advanced features that distinguish GENESIS-II from its predecessor and make it a more powerful and accurate generation tool.

6.1 Selectors

Conceptually, a selector is a variable that a grammar rule or a vocabulary item sets, for reference by grammar rules and vocabulary items later in the generation process. Syntactically, a selector is any string delimited by a dollar sign (`$`), and when either setting or referencing a selector, this syntax must be used. For example, consider the following grammar rule:

```
particle                $:amount :particle
```

In this rule, the command `$:amount` is a `set-selector` command, which indicates that GENESIS-II should set the selector `$:amount`.

Motivation

Before we describe selectors in more detail, we motivate the need for them by discussing the problem of word-sense disambiguation—a formidable challenge in both machine translation and language generation. Word-sense disambiguation may be defined as the process of resolving semantic ambiguities that arise during translation. That is, a word in one language may have several context-dependent translations in another language, and a translator—machine or otherwise—must select the correct translation. In GENESIS-II, this problem manifests itself when the system converts a meaning representation into a string in a language other than English. Theoretically, GALAXY meaning representations are interlingual, but experience with multilingual generation has demonstrated that, in fact, the meaning representations are best suited to generation in English. Specifically, the constituents of meaning representations can be semantically ambiguous and difficult to translate correctly. A good general example is the translation of prepositions from one language to another. Quite often, a preposition in one language has several translations in another language. GALAXY resolves some of these ambiguities by making the constituents more specific. For example, meaning representations for JUPITER English contain predicates named `in_loc`, `in_time`, `in_value`, and `in_effect`, in addition to the simple `in`. However, there are many cases in which GALAXY constituents are less specific, and, consequently, word-sense disambiguation is a serious challenge.

Setting selectors

Domain experts use selectors to resolve these semantic ambiguities. In particular, domain experts set selectors to indicate context, and the system subsequently references the selectors for context-specific vocabulary generation. In GENESIS-II, there are two ways to set selectors with the `set-selector` command. The first is to use a `set-selector` command in a grammar rule, as shown in the `particle` example above. The other way to set a selector is to use a `set-selector` command in a vocabulary item. The syntax is similar in that the command is simply the selector itself. However, the difference is that `set-selector` commands may appear only at the end of vocabulary items, and they must be preceded by one semicolon. For example, the following vocabulary item contains two `set-selector` commands:

```
increasing            0 "zeng1_qiang2" ; $:incre1 $:incre2
```

Once GENESIS-II has finished generating with a vocabulary item that contains some number of `set-selector` commands, GENESIS-II adds the selector/s to the list of active selectors in the generation environment.

Referencing selectors

GENESIS-II references the generation environment for selector information when generating vocabulary. In other words, when GENESIS-II is using a lexical item to generate vocabulary, it compares the list of selectors with strings in the lexical item and selects the most specific generation string possible. Since GENESIS-II maintains a list of active selectors, it begins with the selector most recently set and continues backwards through the list. It compares each selector in the list with each selector in the vocabulary item. If there is a match, GENESIS-II examines the string following the matching selector. If the string is a `lookup` command (*i.e.,* delimited by an exclamation point), GENESIS-II executes it by generating vocabulary for the specified word. Otherwise, GENESIS-II interprets the string to be the root string, with which it generates vocabulary in the standard way.

Domain experts can also use selectors as conditionals in `if` commands, as in the following rule from the Chinese JUPITER grammar:

```
direction            ($if $:wind >direction1 >direction2)
```

Info frame

The generation environment to which we refer is, in fact, the `info frame`. That is, GENESIS-II uses the `info frame` to maintain a list of active selectors. For example, the state of the `info frame` after executing the `set-selector` commands in the `increasing` vocabulary item shown above might be:

```
{q info
   :selector {q selector
                 :pred {p $:incre1}
                 :pred {p $:incre2} } }
```

The scoping of selectors differs from the scoping of the `info frame`, however. As you may recall, the `info frame`'s scope is governed by the relationship between constituents. That is, modifications made to the `info frame` during the generation of a child frame do not affect the parent. A selector's scope, on the other hand, is governed by the relationship between grammar rules. Specifically, a selector only applies until GENESIS-II finishes executing the grammar rule in which the selector was set. For example, consider the following two grammar rules:

```
wind_speed          ($if to_value >topic_core >speed) :particle
                    :adverb >prepred >qualifiers :topic >conjn
                    >postpreds >and

speed               $:reaching >topic_core
```

Suppose that the conditional in the `wind_speed` rule's `if` command evaluated to false. GENESIS-II would execute the "else" consequent, the `goto` command, `>speed`, by descending into the grammar rule for `speed`. The first command in the `speed` rule is a `set-selector` command, which GENESIS-II would execute by adding the selector `$:reaching` to the `info frame`. The selector would apply until GENESIS-II had finished executing the entire rule, at which point, GENESIS-II would remove the selector from the `info frame`. In other words, the `$:reaching` selector would apply to generation during the execution of the `goto` command, `>topic_core`, and, when GENESIS-II returned control to the `wind_speed` rule, the selector `$:reaching` would no longer apply.

Examples

- `particle                $:amount :particle`

  Suppose we were generating a target string in Spanish with the `particle` grammar rule from above. Also suppose that the key value for `:particle` in the current frame were the string `"up"` and that its lexical entry were:

  ```
  up                  O "ma's" $:amount "hasta"
  ```

  In this situation, GENESIS-II would recognize that the selector `$:amount` had been set, and it would select the generation string `"hasta"` over the default generation string "ma's". Hence, the semantic ambiguity of whether "up" translates into "más" or "hasta" would be resolved by using selectors.

- higher A "ma's alto" $:wind "!strong" F "ma's alta" PL_M "ma's altos" PL_F "ma's altas"

  Again we have an item from the Spanish JUPITER lexicon. In this item, the selector $:wind is associated with a lookup command. The domain expert selected a lookup command over a simple string, because the Spanish word for "strong" has a different set of endings than the Spanish phrase for "higher". Therefore, GENESIS-II must pass control to the vocabulary item for "strong" in order to generate a string with the correct inflection.

6.2 Alternates

GENESIS-II allows domain experts to avoid repetition by permitting them to define multiple grammar rules for the same item. That is, GENESIS-II queues all rules that are indexed by the same name, and each time it needs to generate with a rule by that name, GENESIS-II dequeues the head of the queue, uses it for generation, and then enqueues it at the tail. In this way, GENESIS-II cycles through the rules and avoids repetition.

Example

The GALAXY conversational system must often ask users whether there is anything more the system can do for them. Of course, some variety in this type of query is preferable to repeating the monotonous, "Can I help you with something else?" A domain expert can vary the system's query by defining the following grammar rules:

```
something_else      "Can I help you with something else?"

something_else      "What else would you like to know?"

something_else      "Is there anything else?"

something_else      "Is there something else?"

something_else      "What else?"
```

The first time GENESIS-II uses a rule for `something_else`, it will generate the string, `"Can I help you with something else?"` The next time it uses a rule for `something_else`, it will generate the string `"What else would you like to know?"` The sixth time it uses a rule for `something_else`, it will wrap back to the first generation string, `"Can I help you with something else?"` In this way, GENESIS-II cycles through the grammar rules for `something_else` and avoids monotony.

6.3 Grouping Grammar Rules

One of our objectives in developing GENESIS-II was to reduce the work of domain experts. We hoped to achieve this primarily through a consistent and flexible framework that was simple to learn and use. In addition, we sought to reduce the actual number of rules domain experts needed to construct by eliminating redundancy within the linguistic components themselves. In this section, we discuss the GENESIS-II "grouping" mechanisms that make this type of reduction possible.

The original GENESIS system has an idiosyncratic specification that forces domain experts into patterns of repetition for several reasons. For one, GENESIS determines the order in which it processes a frame's predicates by the relative ordering of the corresponding rules in the grammar file. That is, if the rule for "Predicate A" appears before the rule for "Predicate B", then GENESIS will always process Predicate A before it processes Predicate B. This framework for ordering predicates essentially demands that there be a specific grammar rule for every possible predicate, even if the generation patterns for sets of predicates are identical. Another reason for the redundancy in rule files is that GENESIS does not contain an equivalent for the `goto` command. As mentioned in Section 5.2.4 of the last chapter, domain experts can use `goto` commands to collapse grammar rules and thereby reduce the size of grammar files.

GENESIS-II does not share its predecessor's idiosyncratic specification for ordering predicates. As we discussed in the previous chapter, each grammar rule can specify the ordering of predicates, either by explicitly listing them or by using the `$rest` command. Therefore, a grammar rule's position in the grammar file is unimportant in our framework. Given this added degree of freedom, we were able to infuse GENESIS-II with additional mechanisms for reducing the size of grammar files. Specifically, we have given GENESIS-II the capability to recognize more than one particular grammar rule for generating a string from a given constituent. GENESIS-II has a series of rules it can use for generating a string from a given constituent, and it "backs off" from the most specific to the most general until it finds a rule to use.

In this section, we examine the ways in which GENESIS-II backs off from the specific grammar rule to find other applicable grammar rules for specifying generation. We shall discuss the back-off forms from most specific to most general, and we shall provide some intuition into our motivation for creating them and into their utility within the GENESIS-II framework.

Specific rules

When processing a frame, GENESIS-II considers the rule whose name matches the frame's name to be the "specific rule." By paging through the previous chapters of this thesis, we can find several examples of such rules. In fact, until this point, we have used only specific rules in our examples. We include here yet another example. Consider the following meaning representation:

```
{c weather_event
   :topic {q precip_act
             :name "sleet"
             :pred {p expected } } }
```

It consists of three frames—named `weather_event`, `precip_act`, and `expected`. The following three toy grammar rules of the same names would suffice for specifying the generation string, `"sleet expected"` (assuming we have a reasonable English lexicon):

```
weather_event        :topic

precip_act           :name expected

expected             $core
```

Because the names of the grammar rules match the names of the frames, we would designate each rule as the specific rule for the corresponding frame.

Goto rules

In Section 5.2.4 of the last chapter, we introduced the `goto` command and described at length its primary effect in the GENESIS-II framework. We also mentioned that `goto` commands have a secondary effect, in that they add a method of backing-off for the frame's children. Before we describe this aspect of the `goto` command in detail, we provide some motivation.

Over the years, domain experts have found that constituents appearing in the same position in a sentence often have the same generation pattern. For example, predicates that appear before the topic of the sentence often have a common generation pattern, which differs from the common generation pattern shared by predicates that appear after the topic of the sentence. When designing GENESIS-II, we decided to provide a mechanism that would allow domain experts to capture these commonalities in a convenient framework and thereby reduce the repetition of common generation patterns.

Thus, we introduced the secondary effect of the `goto` command. As we described in the previous chapter, the primary effect of the `goto` command is to descend into another grammar rule. As we shall discuss in this chapter, the secondary effect is to add a grammar rule to a list of back-offs. The grammar rule GENESIS-II adds to the list is simply the rule indexed by the concatenation of the `goto` command (stripped of the `>`) and the string, `"_template"`. GENESIS-II applies this list of back-offs to all child frames processed before GENESIS-II finishes executing the `goto` command. Let us chase this description with a solid example.

Consider the following rule from the Spanish grammar:

```
clause_template        >conjn >prepred >prepred2 :conditional :adverb
                       :topic >postpreds :and
```

This example contains four `goto` commands.[1] When executing each of them, GENESIS-II would first search the grammar for the appropriate back-off (*e.g.,* `conjn_template`, `prepred_template`, etc.), and, if found, GENESIS-II would add it to the list of back-offs for the current frame's children. GENESIS-II would then descend into the rule specified by

[1] Note that the `goto` commands—`>prepred`, `>prepred2`, and `>postpreds`— imply that the Spanish domain expert found the pattern we described above: predicates in similar positions have similar generation patterns.

the goto and continue generation there. Let us look at one of these goto commands in more depth. The rule specified by >prepred2 and the corresponding back-off are as follows:

```
prepred2             continuing ending falling expected clearing

prepred2_template    :conjn :conditional :particle :qualifier
                     >topic_core :topic :adverb ($if :or !or) :or
                     >and >postpreds
```

To execute the goto command, >prepred2, GENESIS-II would first add the back-off rule, prepred2_template, to the list of back-offs and then descend into the rule for prepred2. All of the commands in the prepred2 rule are predicate commands. Therefore, to execute each of them, GENESIS-II would search the current frame's predicate list, and if there were a match, GENESIS-II would process the child. As you know, GENESIS-II's first step when processing a frame is to find a rule with which to generate. In this particular case, GENESIS-II would find the prepred2_template rule in the back-off list, and so, it would be a viable candidate for specifying generation. In particular, if there were no specific rule, GENESIS-II would select the prepred2_template as the rule with which to generate a string for the child in question.

Grouping rules

When designing the goto rules, we observed that they were inherently dependent upon the goto commands. That is, GENESIS-II only adds a goto rule to the back-off list by executing the appropriate goto command. For example, the prepred2_template from above can be used only when a >prepred2 command is executed.

We therefore decided to include another mechanism for utilizing the generational similarities captured by the goto rules. Specifically, we added "groups" to the GENESIS-II framework. The groups are intended to provide GENESIS-II with even more options for backing off. Every grammar file may contain one grouping rule for each type of constituent—clause, predicate, topic, key, and list. The name of a grouping rule must consist of the constituent type concatenated with the string, "_groups". In other words, the following five rule names identify grouping rules:

- clause_groups

- `predicate_groups`
- `topic_groups`
- `key_groups`
- `list_groups`

Each group should consist of a list of items. For example, the English JUPITER grammar contains the following predicate group:

```
predicate_groups        pred_no_core postpred_without postpred_with
```

The grammar should also contain a rule for each of the items in a group. So, the grammar from our example contains rules for `pred_no_core`, `postpred_without`, and `postpred_with`. Additionally, the grammar should contain a goto back-off for each of the rules. Therefore, the grammar in our example also contains the goto back-offs: `pred_no_core_template`, `postpred_without_template`, and `postpred_with_template`.

Consider yet another example. The following rules are from the `Mercury` English domain:

```
predicate_groups        predless_pred

predless_pred           nth flight_interval superlative
                        superlative_size superlative_time
                        superlative_arrival_time relative_time forward
                        nweeks flight_mode flight_cycle fare_class
                        airline month_name month_day at

predless_pred_template >prepreds :negate :topic >postpreds >and
```

Suppose GENESIS-II were searching for a rule with which to generate a predicate frame named `superlative_size`. Further suppose that GENESIS-II were unable to find a specific rule or a goto rule with which to generate. In this case, GENESIS-II would look for the appropriate grouping rule, which would be the rule named `predicate_groups`. According to this rule, the only predicate group is named `predless_pred`. Therefore, GENESIS-II would search for a rule named `predless_pred` and would find that in fact superlative_size was in the list of predless_preds. GENESIS-II would then search for the corresponding goto rule—predless_pred_template—and use it for generating a string from the superlative_size frame. In this example, the power of grouping is evident; it allows GENESIS-II to utilize the predless_pred_template, even though the generation chain did not explicitly invoke the predless_pred rule.

Default rules

At the opposite end of the spectrum from the specific rules lie the default rules, the most general back-offs in the GENESIS-II framework. When a domain expert specifies a default rule, she should attempt to capture the most common generation protocol for a particular type of constituent. By doing so, she can reduce the number of specific rules necessary for correct generation, because GENESIS-II can always fall back on the defaults.

Every grammar file may contain one default rule for each type of constituent—clause, predicate, topic, key, and list. The name of a default rule must consist of the constituent type concatenated with the string, "_template". In other words, the following five rule names identify default rules:

- clause_template
- predicate_template
- topic_template
- key_template
- list_template

As a small example, consider the following three default rules from the Spanish JUPITER grammar:

```
clause_template        >conjn >prepred >prepred2 :conditional :adverb
                       :topic >postpreds :and

predicate_template     :particle :adverb >prepred :quantifier
                       >qualifiers :topic >conjn >and >postpreds
```

`topic_template`	`>prepreds >pre_topic2 :conditional >topic_core :loc_qualifier :qualifier --continuando :units :and :or >postpreds`

These rules indicate what the Spanish domain expert found to be the most common generation patterns for clauses, predicates, and topics, respectively.

6.4 Set, Let, Clone

In Chapter 4, we introduced the concept of an `info frame`, and we discussed the ways in which GENESIS-II automatically updated and referenced the `info frame`. Earlier in this chapter, we discussed the concept of selectors—the environment variables that could be added to the `info frame` and referenced as needed during the generation process. In this section, we describe another way in which grammar rules and vocabulary items can update and reference the `info frame` during generation. Domain experts can use the three commands—`set`, `let`, and `clone`— for just this purpose and also for the purpose of updating and referencing a frame GENESIS-II is processing.

Set

Let us begin with the `set` command. The syntax of the `set` command is: `($set target source)`, where the target is always a keyword and the source may have one of several formats. We discuss the different formats and their semantics below:

- `($set :target "source")`

  In the simplest case, the source is a quoted string, and GENESIS-II's behavior is to add the key pair [`:target, "source"`] to the frame it is currently processing.

- `($set :target !source)`

  If the source is a `lookup` command, GENESIS-II executes it, using the lexicon, and adds the result to the current frame as the key value for the keyword `:target`.

- `($set :target :source)`

If the source is a keyword, GENESIS-II searches the current frame for the `:source`. If successful, GENESIS-II copies its key value and adds it to the current frame as the key value for the keyword `:target`.

- `($set :target :source[predicate])`

  `($set :target :source[:key])`

  If the source contains a bracketed portion and the bracketed string is not `$core`, then GENESIS-II searches the current frame for the predicate or keyword in the brackets. If it finds a matching predicate or if it finds a matching keyword with a frame key value, then GENESIS-II searches that child frame for the `:source` and sets the key value of `:target` in the current frame to be the key value of `:source` in the child frame.

- `($set :target :source[$core])`

  If the source contains the bracketed string `$core`, then GENESIS-II searches the lexicon for the vocabulary item indexed by the current frame's name. If there is a match, GENESIS-II searches the item for the keyword `:source` and sets the key value of `:target` in the current frame to be the key value of `:source` in the vocabulary item. If the source is one of the linguistic keywords—`:gender`, `:number`, or `:pos`—GENESIS-II finds the appropriate linguistic information in the vocabulary item and sets the key value of `:target` accordingly.

As mentioned before, domain experts can also use the `set` command to modify the `info frame`. The syntax is identical to the syntax described above, except that the target should be preceded by a caret to indicate that GENESIS-II should make the modifications to the `info frame` and not to the frame it is processing. For example, the following command would indicate that GENESIS-II should add the key pair [`:target, "source"`] to the `info frame`:

```
($set ^:target "source")
```

Domain experts can also specify `set` commands in the lexicon. The syntax differs slightly from `set` commands in the grammar. Whenever GENESIS-II encounters a string delimited by a colon or a caret in a vocabulary item, it interprets the string to be a `set` command. To add a key pair to the current frame, domain experts should include the key pair in the vocabulary item by using the form:

```
{c truth
   :aux "will"
   :topic {q pronoun
            :name "they" }
   :pred {p serving
           :topic {q meal
                     :quantifier "indef"
                     :name "meal" } } }
```

Figure 6-1: The meaning representation for "`Will they serve a meal?`"

```
:key "value"
```

To add a key pair to the `info frame`, domain experts should insert a caret before the key, as follows:

```
^:key "value"
```

In Section 5.1, we briefly mentioned the role of the keyword `:mode` in GENESIS-II. Now that we have presented the `set` mechanism, we can illustrate the function of the `:mode` with an example. Consider the meaning representation in Figure 6-1. Suppose that the lexicon contained the following three items:

```
will              X "will" ^:MODE "root"

serving           V2 "serv"

V2                V "V" ROOT "e" PL "e" THIRD "es" ING "ing"
                  PP "ed"
```

Further suppose that GENESIS-II were using a rule for `truth` in which the `keyword` command, `:aux`, preceded the `predicate` command, `serving`. In this case, GENESIS-II would first look for the string "`will`" in the lexicon. The entry for "`will`" contains the `set` command:

```
^:MODE "root"
```

Therefore, GENESIS-II would add the key pair [`:MODE "root"`] to the `info frame`, and it would add the default string "`will`" to the target string. Later, when generating vocabulary for "serving", GENESIS-II would search for an entry corresponding to its part of speech V2. GENESIS-II would then use the :mode to determine which suffix to use. Because the info frame contains the key pair [:MODE "root"], GENESIS-II would select the suffix "e" and generate the correctly inflected form, "serve".

Let

The let command is very similar to the set command. They differ in only two ways—one syntactic and one semantic. The syntactic difference is that the first string of a set command is $set, and the first string of a let command is $let. The semantic difference concerns the scoping of commands that affect the info frame. All changes made to the info frame in a set command persist for the life of the info frame; as discussed in Section 4.2.1 of Chapter 4, all modifications to the info frame affect the current frame and its children, but not the parent frame. The scope of changes made by the let command is even narrower. They only apply until GENESIS-II finishes executing the rule which contained the let.

Clone

The clone command is simply shorthand for a very specific type of set command. If a domain expert wishes to specify a set command that has one of the following forms:

```
($set :target :source[:key])

($set :target :source[predicate])

($set :target :source[$core])
```

and the :target and :source are identical, then the domain expert can use the following clone commands instead:

```
($clone :source[:key])

($clone :source[predicate])

($clone :source[$core])
```

```
:fl_list ( {c departing_flight
              :departure_time "6:15"
              :depart_xm "a.m."
              :arrival_time "8:03"
              :arrive_xm "a.m." }
           {c departing_flight
              :departure_time "8:50"
              :depart_xm "a.m."
              :arrival_time "10:45"
              :arrive_xm "a.m." }
           {c departing_flight
              :departure_time "11:20"
              :depart_xm "a.m."
              :arrival_time "1:11"
              :arrive_xm "p.m." } )
```

Figure 6-2: A sample list from the MERCURY domain.

6.5 Lists

List structures are prevalent in database retrievals, and so, the GALAXY system's list type is a common constituent of meaning representations. In Figure 3-3 of Chapter 3, we provided an example of a list in the GALAXY framework; for reference, we provide another example in Figure 6-2 of this chapter.

Generating a string from a list is not as straightforward as it might seem at first. It is a challenging task particularly because the position of an item in a list often influences its surface realization. For example, in many lists, all but the first and last entries are preceded by a comma, and the last is preceded by the word `"and"`. GENESIS-II handles these distinctions by tagging list items with positional information. In this section, we shall describe the way in which GENESIS-II processes lists and the language that domain experts can use to specify list generation.

GENESIS-II generates a string from a list in a multifold process. First, it unbundles the list into distinct items by calling upon list manipulators in the GALAXY library. Next, for each list item, GENESIS-II creates a wrapper frame, into which it inserts the item as the key value for the keyword `:nth`. Then, depending upon the item's position in the list, GENESIS-II reinserts it into the wrapper by tagging it with positional information. If the item is the first or last item in the list, it receives the `:first` or `:last` tag respectively. If the item is any item but the last, it receives the `:butlast` tag. If there is only one item in the list, it receives the `:singleton` tag. For example, if GENESIS-II were processing the list in Figure 6-2, it would produce the three wrapper frames in Figure 6-3.

Once GENESIS-II has created the wrappers, it attempts to generate a string for each list item. First, by using the back-offs described in Section 6.3, GENESIS-II searches for an appropriate rule with which to generate. That is, GENESIS-II first searches for the specific rule, *i.e.,* the rule indexed by the list's key. The most specific rule for the list presented in Figure 6-2, for example, would be named `:fl_list`. Goto rules do not apply to lists, and so, GENESIS-II would next search for a grouping rule by looking at the `list_groups` rule. Finally, GENESIS-II would back off to the default rule for lists—`list_template`.

Given a rule with which to generate, GENESIS-II proceeds to process each wrapped and tagged list item sequentially by executing the rule in the standard fashion. Every command we have discussed retains its meaning in this context, and `keyword` commands such as `:first` and `:nth` result in the extraction and generation of the list items themselves. We should note that when GENESIS-II generates a string for one of the special list keywords (`:nth`, `:first`, `:butlast`, `:last`, or `:singleton`), it removes all other special keywords from the wrapper so that it does not generate the list item twice.

Example

As an illustration, we return to the example in Figure 6-2. Suppose that GENESIS-II were using the following rules to generate a string from a `:fl_list` in the MERCURY English domain:

```
:fl_list          >singleton >first >but_last >last

singleton         :singleton

first             "one" :first

but_last          , "another" :butlast

last              "and the last" :last
```

Let us walk through the generation of a string from the `:fl_list`. As mentioned before, Figure 6-3 contains the wrappers GENESIS-II would create upon preprocessing the list in

```
{p wrapper
   :nth {c departing_flight
           :departure_time "6:15"
           :depart_xm "a.m."
           :arrival_time "8:03"
           :arrive_xm "a.m." }
   :first {c departing_flight
             :departure_time "6:15"
             :depart_xm "a.m."
             :arrival_time "8:03"
             :arrive_xm "a.m." }
   :butlast {c departing_flight
               :departure_time "6:15"
               :depart_xm "a.m."
               :arrival_time "8:03"
               :arrive_xm "a.m." } }
{p wrapper
   :nth {c departing_flight
               :departure_time "8:50"
               :depart_xm "a.m."
               :arrival_time "10:45"
               :arrive_xm "a.m." }
   :butlast {c departing_flight
               :departure_time "8:50"
               :depart_xm "a.m."
               :arrival_time "10:45"
               :arrive_xm "a.m." } }
{p wrapper
   :nth {c departing_flight
               :departure_time "11:20"
               :depart_xm "a.m."
               :arrival_time "1:11"
               :arrive_xm "p.m." }
   :last {c departing_flight
               :departure_time "11:20"
               :depart_xm "a.m."
               :arrival_time "1:11"
               :arrive_xm "p.m." } }
```

Figure 6-3: Wrappers for the list items in Figure 6-2.

Figure 6-2. After wrapping and tagging the list items, GENESIS-II would use the `:fl_list` grammar rule above to generate a string from each of the list items. For the first list item, the `>singleton` command would evaluate to the empty string, but the `>first` command would evaluate to the substring `"one"`, followed by the generation string for the list item. For the second item, the `>but_last` command would evaluate to the substring `", another"`, followed by the generation string for the second list item. For the third item, the `>last` command would be the only command in the `:fl_list` rule to produce a result, and that would be the substring `"and the last"`, followed by the generation string for the last list item. The concatenated result would therefore be something of the form, `"one departing at 6:15 a.m..., another departing at 8:50 a.m...and the last departing at 11:20 a.m..."`.

6.6 Reorganizing the Frame Hierarchy

The meaning representation is theoretically an interlingual structure. However, our experience with multilingual generation has shown that meaning representations are often difficult to interpret when working with a language other than English. Because word order differs greatly from language to language, a representation that seems logical for English might be quite illogical for a language like Chinese or Japanese. In particular, we have found that a constituent deep in the meaning representation may need to appear at an unusual location in the generation string for some languages.

Interestingly, there is an English phenomenon that also requires this type of movement. The problem of "wh-movement" is the well-defined linguistic phenomenon of moving an embedded noun phrase to the beginning of a wh-query.[2] For example, in Figure 6-4, the wh-quantified noun phrase `"what time"` appears sentence-initially in the English generation string, despite the phrase's location in the meaning representation.

When architecting GENESIS-II, we knew that its framework would have to resolve these movement issues in order for our system to be an effective multilingual generator. Therefore, we devised a handful of mechanisms for reorganizing the frame hierarchy. In keeping with our paradigm of generality, none were designed specifically for handling the known problems, since domain experts had found the original GENESIS system's hard-wired mechanisms to

[2]A "wh-query" is a question beginning with a word like "who" or "what."

```
{c truth
   :mode "finite"
   :number "third"
   :aux "do"
   :topic {q flight
            :quantifier "def" }
   :pred {p arrival_time
           :topic {q time
                     :quantifier "what" } } }
```

Figure 6-4: The meaning representation for `"What time does the flight arrive?"`

be inflexible and unintuitive. Rather, we designed general mechanisms that solved the problems at hand, and we hoped that they would be general enough and powerful enough to solve problems that we had not anticipated. Over the past few months, these mechanisms have in fact been useful for solving unexpected difficulties as they have arisen.

There are four general commands for reorganizing the frame hierarchy in GENESIS-II, and the semantics of these commands suggest a natural division into two categories. We group the first two commands—`tug` and `yank`—because they both indicate to GENESIS-II that it should extract a lower-level constituent for generation and inclusion at a higher level. These two commands differ only in scope, and we shall discuss this difference presently. The other two commands—`push` and `pull`—work together to generate and defer inclusion of a string until a higher level. Specifically, the `push` command indicates that GENESIS-II should generate a string and then defer its inclusion in the target string, and the `pull` command indicates that GENESIS-II should include a deferred string in the target string. In this way, a lower-level constituent can "push" a generation string aside, and a higher-level constituent can "pull" it into the target string.

We shall divide this section into two parts, one for each of the categories described above. In the first part, we shall discuss the `tug` and `yank` commands, and in the second part, we shall discuss the `push` and `pull` commands.

6.6.1 Tug and yank

The `tug` command and the `yank` command share the function of extracting lower-level constituents for generation and inclusion at a higher level. As their names suggest, however, they differ in strength. We will first examine the `tug` command, because the behavior of the `yank` command will be simple to explain once the behavior of the `tug` command is clear.

Tug

The delimiter of a `tug` is a less-than sign, followed by two dashes (`<--`), and it can precede several different types of strings. Let us examine each `tug` form.

- `<--:keyword`

  When GENESIS-II encounters a `tug` of this form, it searches the frame's children (predicate children, then keyword children) for the keyword specified in the command. If such a keyword is found, GENESIS-II moves the key pair from the frame's child into the frame, generates a string for the keyword, and then includes the string in the full target string.

  Example

  The JUPITER weather system produces the following frame for the English phrase `"some areas of fog"`:

  ```
  {c weather_event
     :pred {p areas_of
              :topic {q weather_act
                        :quantifier "some"
                        :name "fog" } } }
  ```

  The structure of the frame suggests that the quantifier modifies `"fog"`. In the original sentence, however, the quantifier modifies `"areas"`. Thus, we have an undesirable frame hierarchy, which we can correct using the `tug` command. The Spanish grammar file makes such a correction in the rule

  ```
  areas_of            <--:quantifier >prepreds >topic_core
                      :topic >postpreds
  ```

  When using this rule to generate a string for a frame, GENESIS-II `tug`s the quantifier from one of the frame's children, generates a string for it, and places it before the rest of the target string. Therefore, the resulting Spanish target string is "algunas áreas de neblina", where "algunas" means "some" and "áreas" means "areas". The quantifier has been restored to its correct position in the phrase.

- `<--predicate_name`

  In the case of a tug delimiter followed by no colon, GENESIS-II interprets the string stripped of the delimiter to be a predicate name. It searches the frame's children (predicate children, then keyword children) for a predicate with the specified predicate name. If such a predicate is found, GENESIS-II moves it from the frame's child into the frame, generates a string for it, and then includes the string in the full target string.

Examples

Consider the following rule from the Chinese JUPITER grammar file:

```
periods_of              <--month_date <--time_interval
                        >pre_chance_of :topic :conditional
                        :adverb :loc_qualifier >chance_preds >and
```

Suppose GENESIS-II were using this rule to generate a string for a frame named periods_of. GENESIS-II would first search all of the frame's children until it found one that contained the predicate named month_date. If it found one, GENESIS-II would move the predicate into the parent periods_of frame and generate a string for it. Note that this rule contains a tug of the predicate time_interval, as well. In Section 7.4 of the next chapter, we shall describe the linguistic motivation for tugging in instances such as this one.

- `<--:keyword[predicate_name1 predicate_name2 keyword1]`

  When GENESIS-II encounters a tug consisting of a keyword followed by bracketed substrings, it interprets the bracketed substrings as specific frame children in which to look for the specified keyword. GENESIS-II therefore searches only in the predicate children and keyword children in the brackets for the keyword specified by the command. It searches these children in the order in which they appear in the command.

If it finds the keyword in one of these children, it moves the key pair into the frame and generates a string for the keyword.

Examples

The rule

```
--pm_test              <--:xm[minutes] :xm
```

in the Chinese grammar file contains a `tug` of the form just discussed. When GENESIS-II uses this rule, it first searches the frame for a predicate name `minutes`. If it finds such a predicate and that predicate contains a key pair for the keyword `":xm"`, then GENESIS-II moves the key pair into the parent frame and generates a string for it. GENESIS-II also looks inside the frame itself for the keyword `":xm"` and generates a string for it. Note that if the frame starts out with a key pair for the keyword `":xm"` and the `tug` is successful, then GENESIS-II overwrites that key pair with the key pair from the `minutes` predicate. Thus, the second command in the rule generates nothing.

- `<--predicate_name[keyword1 predicate_name1 keyword2]`

  When GENESIS-II encounters a `tug` of this form, it interprets the bracketed substrings as specific frame children in which to look for the specified predicate name. It searches only in the predicate children and keyword children in the brackets for the predicate specified by the command. It searches these children in the order in which they appear in the command. If such a predicate is found, GENESIS-II moves it into the frame and generates a string for it.

Examples

The Chinese grammar file also contains the following rule:

```
complex_by_time        (by_time <--by_time[:and]
                        <--by_time[:or]) .
```

The `or` command in the rule signals to GENESIS-II that it should try a few different things in order to generate a string for the predicate `by_time`. The first command in the `or` command is to look for a predicate named `by_time` in the frame itself and to generate a string for it, if it exists. If not, Genesis-II must look for the keyword `:and` in the frame. If there is such a keyword and its key value is a frame, then Genesis-II must search that frame for the predicate `by_time`. If it finds it, Genesis-II generates a string for it. Otherwise, Genesis-II should try one last time, this time looking inside the key value frame for the keyword `:or`.

- `<-->rule_name[predicate_name1 keyword1 keyword2]`

  This final `tug` form is somewhat different from the others. When Genesis-II encounters a `tug` delimited by "`<-->`", it interprets the substring before the bracketed substrings to be a rule name and the bracketed substrings to be specific frame children for which to look. In the previous commands, Genesis-II was looking *within* the frame children for a predicate or a keyword. In this case, Genesis-II looks *for* particular frame children. If Genesis-II finds a matching child, it searches for the rule specified in the command and uses it to generate the frame child.

Yank

The `yank` differs syntactically from the `tug` in that every dash is replaced by the more "forceful" equal sign, *e.g.,* `<==:adverb[probability]`. Semantically, they differ in that the `yank` command is not restricted to looking at the current frame's children, but rather, it has the power to look at all of the current frame's descendents, from children down to great^n-grandchildren. Genesis-II uses a breadth-first search algorithm for `yank` commands.

6.6.2 Push and pull

We now examine the other two commands for frame manipulation. As mentioned in the introduction to this section, these two commands must be used together first to generate a string and defer it ("push") and then to include it in the full target string ("pull"). We begin by discussing the `push` command.

Push

Syntactically, the `push` command is a `goto` command, in which the embedded rule name begins with two dashes, *e.g.,* `>--auxes`. Semantically, the two commands are similar, as well. When GENESIS-II encounters a push command, it strips the greater-than sign from the command string and searches for the resulting string in the grammar. If GENESIS-II finds a matching rule, it descends into the rule and executes the commands in the new rule. The difference between the push command and the goto command is that when GENESIS-II executes a goto, it continues adding to the generation string by executing the commands in the new rule. When GENESIS-II executes a push command, on the other hand, it defers inclusion of the substring it generates while executing the new rule. Rather, once GENESIS-II has finished executing the new rule, it adds the generated substring to the info frame as the key value for the name of the new rule. It then returns to where it left off in the original rule and continues generation at that point.

Let us look at a small example. The JUPITER Spanish grammar file contains the following two rules:

```
accumulation            >prepreds $:accum :quantifier >--temp
                        <==:adverb[probability]
                        <==:temp_qualifier[probability] --temp
                        :probability :qualifier :quant
                        ($if accumulating $:no_core) >topic_core
                        :temp_qualifier :loc_qualifier :adverb
                        >postpreds

--temp                  :temp_qualifier
```

The grammar rule for accumulation contains the command, >--temp. When GENESIS-II encounters this pull command, it descends into the grammar rule for --temp, in which it executes the :temp_qualifier command and defers inclusion of it. In other words, GENESIS-II extracts the :temp_qualifier from the current frame, generates vocabulary for its key value, and saves it as the key value for --temp in the info frame. GENESIS-II then returns to the accumulation rule and executes the following two yank commands, the second of which might overwrite the :temp_qualifier key pair in the current frame. Fortunately, the value has been saved as the key value for --temp in the info frame, and so, it can be pulled in later in the generation process.

Pull

The counterpart of the `push` command is the `pull` command. Syntactically, the `pull` command consists of two dashes, followed by a string, *e.g.,* `--temp`. When GENESIS-II encounters a `pull` command, it searches the `info frame` for a keyword that matches the command. If it finds such a keyword, it extracts the key pair and adds the key value to the target string. If GENESIS-II does not find the keyword, it marks the location in the target string, finishes executing the rule, and then executes the `pull` command again. If successful this time, it adds the result to the appropriate location.

To illustrate the `pull`, let us return to where we left off in the example above. Once GENESIS-II has executed the two `yank` commands, it executes the `pull` command, `--temp`. To execute this command, GENESIS-II searches the `info frame` for the keyword `--temp`. We know that if the `info frame` contains `--temp`, the key value for `--temp` is the string GENESIS-II generated from the keyword command `:temp_qualifier`. If GENESIS-II finds a match, it adds this key value to the target string and continues executing the commands in the `accumulation` rule.

The `push` and `pull` commands can also be used to solve the wh-movement problem, which we introduced at the beginning of this section. We shall illustrate this solution in Section 7.2 of the following chapter, in which we discuss GENESIS-II's strengths in an attempt to evaluate the system.

Chapter 7

Evaluation of Genesis-II

The task of evaluating a generation system remains a challenging research problem. It is fairly simple to assess some of the quantitative aspects of a system, such as speed and memory requirements.[1] However, it is considerably harder to assess the more interesting, qualitative aspects, such as generation quality. For example, we considered conducting an experiment in which participants compared the quality and correctness of output from GENESIS and GENESIS-II, but we felt that the test would be unfair to both systems for several reasons. For one, domain experts spent years perfecting some of the GENESIS knowledge bases, whereas, they began creating GENESIS-II knowledge bases just a few months ago. Given the fine-tuning that comes with maturation, such a comparison seemed unfair. On the other hand, there are also knowledge bases that are essentially unique to GENESIS-II. As mentioned throughout the thesis, GENESIS's narrow framework renders correct generation impossible in some circumstances, and in particular, in certain languages. Therefore, it would be relatively useless to compare the two systems' outputs in Chinese or Japanese, for example.

We finally decided on a structure for this chapter that we felt would allow the reader to make the most meaningful assessment of GENESIS-II. The first section will contain a brief

[1] Although such considerations may seem peripheral, they are important, especially when the generator must perform in real-time, as in a conversational system. As mentioned in Section 2.3.4 of Chapter 2, the engineers of the statistical generation system at CMU have considered the issue of speed, and they emphasize it as an advantage of their system over other systems [OR00]. However, the data they reference in their paper are hardly conclusive. Additionally, when we timed GENESIS-II , the results indicated that our system was faster than theirs by a factor of five. Admittedly, the difference between 200 milliseconds and 40 milliseconds is fairly insignificant, even in real-time. However, we mention it in an attempt to dispel the myth that their statistical system is "much faster than any rule-based system."

"status report", in which we outline current domain development within the GENESIS-II framework. In the remaining sections, we will embark on a detailed examination of the most interesting examples of generation in GENESIS-II. We shall highlight our system's most sophisticated capabilities and their application in real domains and languages, in an attempt to demonstrate GENESIS-II's versatility and power. In particular, we will showcase examples of generation in which GENESIS would have failed. To avoid repetition, we will not make an explicit comparison between the two systems in every example we present. We shall divide this portion of the chapter by domain-language pairs.

7.1 Status Report

One indication of a system's quality is the collective experience of those who use it. Since we began implementing GENESIS-II, researchers in our group have overwhelmingly favored our system over its predecessor. The GENESIS-II framework appears to be easier to learn than the GENESIS framework, although the learning curve is still by no means minute. We have trained a handful of domain experts, including foreign-language experts with a precarious grasp of English. In fact, the many challenges of multilingual generation demand that foreign-language experts attain "fluency" in GENESIS-II, and the examples in this chapter will demonstrate the sophistication that they have already achieved in their knowledge bases.

The range of domains, languages, and meaning-representation styles that GENESIS-II already handles strongly indicates its power and flexibility. The following is a list of the domain-language pairs that experts are developing within the GENESIS-II framework:

- JUPITER Chinese
- JUPITER English
- JUPITER Japanese
- JUPITER Spanish
- JUPITER SQL
- MERCURY Dialogue

- Mercury English
- Mercury Envoice
- Mercury HTML
- Orion Dialogue
- Orion English
- Orion HTML
- Voyager English

The rest of this chapter contains the most interesting examples of generation from some of the domains and languages listed above.

7.2 Mercury English

We begin this series of examples by detailing our solution to the challenging phenomenon of wh-movement. When we introduced the problem in Section 6.6, we provided a small sample frame from the Mercury domain, and we promised a complete description later in the thesis. In this section we shall describe exactly how domain experts can specify wh-movement within the Genesis-II framework.

7.2.1 Wh-movement

Let us first examine the meaning representation in Figure 7-1. As you can see, the constituents for `"what"` and `"time"` are deep in the frame. Moreover, the top-level `truth` frame is "unaware" that the target string will begin with a phrase generated from within the `departure_time` constituent. In fact, one of the complexities of wh-movement is that only the lower-level wh-constituent "knows" its place at the head of the query. Therefore, a `tug` or even a `yank` is powerless in this situation.

The `push` and `pull` commands, on the other hand, are quite powerful. Consider the following grammar rule for `truth`:

```
truth                    --trace (:aux !do) (:topic "it") :not :aux2
                         >main_preds >and
```

```
{c truth
   :aux "will"
   :topic {q flight
            :pred {p flight_number
                     :topic 645 } }
   :pred {p departure_time
           :topic {q time
                     :quantifier "what" } } }
```

Figure 7-1: The meaning representation for "What time will flight 645 depart?"

The rule begins with the `pull` command, `--trace`. Essentially, this command indicates to GENESIS-II that, at this point, it is unclear what wh-phrase (the "trace") should begin the query. Therefore, GENESIS-II marks this place, finishes executing the rule, and then attempts once more to include the deferred value of `--trace`.

Suppose now that, while GENESIS-II was executing the rest of the rule, it encountered the following grammar rule:

```
time                    :quantifier ($if :trace >--trace)
```

GENESIS-II's first action would be to search the current frame for a `:quantifier`. If it were searching the `time` frame in Figure 7-1, it would find the key value `"what"`, and it would look it up in the lexicon. Suppose then that it found the following entry:

```
what O "" :trace "what"
```

Through this entry, the lexicon indicates that it "knows" that the wh-word `"what"` is part of the `trace`. When generating from this entry, GENESIS-II would add nothing to the target string, but it would add the key pair [`:trace "what"`] to the current frame.

When GENESIS-II returned to the `time` rule, it would execute the `if` command, which would evaluate to true. Consequently, GENESIS-II would search the grammar for a rule named `--trace`, and it would find:

```
--trace                 :trace ($if :trace $core)
```

GENESIS-II would then concatenate the `:trace` and the `$core` to produce the string, `"what time"`, which would be placed in the `info frame` as the key value of `--trace`.

Finally, when GENESIS-II finishes making its first pass through the original `truth` rule, it would look at the `info frame` one last time, for the keyword `--trace`. This time, it would meet with success, and so, it would place the wh-phrase, `"what time"`, in its rightful place at the beginning of the sentence.

In this way, domain experts can use the general frame-manipulation mechanisms in GENESIS-II for specifying wh-movement.

7.3 Mercury Envoice

The other MERCURY "language" we examine is the mark-up language that the ENVOICE speech synthesizer understands. As we discussed in Section 5.2.3, domain experts have developed a knowledge base for ENVOICE within the GENESIS-II framework. The knowledge base shares a common grammar file with the knowledge base for English text, but it utilizes a distinct vocabulary file, which contains a mixture of prosodically-marked strings and waveform pointers. In this section, we describe how the selector mechanisms were useful to the developers of this system.

7.3.1 Prosodic selection

An important aspect of speech synthesis is prosody. To produce speech that sounds natural, a synthesizer must vary the pitch of words, depending upon their positions in the utterance. The developers of the MERCURY ENVOICE knowledge base used the selector mechanisms for just this purpose. Consider, for example, the following grammar rule:

```
last                    $LOW !and :last
```

This rule contains the `set-selector` command, `$LOW`. If GENESIS-II executed this rule, it would set the selector `$LOW` before executing the commands, `!and` and `:last`. Suppose that the key value for `:last` included the word `"p.m."`, as in, `"the last flight leaves at 8:30 p.m."` Further suppose that GENESIS-II found the following entry for `"p.m"` in the lexicon:

```
p.m.                    0 "[ mercury/mercury069 108189 123185 1 1 1
```

```
pm ]"
$LOW "[ mercury/mercury092 89189 96446 1 1 1
p_m ]"
```

The first generation string is the default, and, presumably, the second has a lower pitch. Thus, when generating vocabulary for "p.m." in the scenario described above, GENESIS-II would select the value for $LOW, and it would have effectively selected the prosody.

7.4 Jupiter Chinese

Researchers in our group have been developing a conversational system for providing up-to-date weather information to speakers of Mandarin Chinese. The system is called MUXING, which is the Chinese name for the planet Jupiter. In a recent conference paper [WCM+00], the developers of MUXING discuss various aspects of the system, including response generation. In their words, "response generation was probably the most challenging aspect of MUXING. This is because the weather reports are available mainly in English, making response generation essentially a translation task." Moreover, our group uses weather forecasts that have been manually prepared by meteorologists at the U.S. National Weather Service. The forecasters tend to use rich, expressive language in their reports, and so, the weather domain is an interesting, but formidable challenge.

In this section, we provide several interesting examples of generation in the MUXING system. We briefly note that GENESIS-II generates textual output for MUXING in three distinct forms—Pinyin, Simplified Chinese, and Traditional Chinese—by using one grammar and three distinct lexicons.[2] For simplicity, we have chosen to use Pinyin in this section.

7.4.1 Reorganizing the frame hierarchy

In Section 6.6, we discussed how our experience with multilingual generation has shown us that the hierarchy of a meaning representation is sometimes illogical for languages other than English. As you might imagine, we encountered many word-ordering challenges in creating a Chinese knowledge base. Some of the difficulties were resolved outside of GENESIS-II.

[2]In Section 5.2.3 of Chapter 5, we emphasized that domain experts should compartmentalize the functions of the grammar and the functions of the lexicon as much as possible, so that such sharing is possible.

```
{c weather_event
   :pred {p becoming
           :topic {q weather_act
                    :name "sunny"
                    :pred {p in_time
                            :topic {q time_of_day
                                     :modifier "late"
                                     :name "evening" } } } } }
```

Figure 7-2: The meaning representation for "becoming sunny in the late evening".

In particular, it seemed appropriate at times to actually reorder the meaning representation before sending it to GENESIS-II. At other times, it was appropriate to make the constituents of a meaning representation more specific before sending the meaning representation to GENESIS-II. For example, the system now uses the predicates `in_time` and `in_loc` to distinguish between the conditions of being "in a time" an "in a location."

In some cases, we used mechanisms in GENESIS-II to resolve the challenges we encountered. For example, an important distinction between English and Chinese is that time and location information must appear first in a Chinese clause, whereas, in English, the position of time and location information is not as fixed. We illustrate this distinction with the meaning representation in Figure 7-2. In Chinese, the phrase for "in the late evening" must appear first, and so, the rule for `becoming` contains the `yank` command, `<--in_time[:topic]`. The Chinese grammar contains many such `tug` and `yank` commands for specifying the reorganization of time and location predicates.

7.4.2 Keyword-renaming

Another interesting problem that arose in designing a Chinese knowledge base concerned the keyword labels for linguistic modifiers. That is, GALAXY engineers had created a small set of groups to which modifiers were assigned according to their meaning. For example, the keyword `:temp_qualifier` is paired with temporal qualifiers, whereas the keyword `:loc_qualifier` is paired with location qualifiers. Unfortunately, some pairings are inappropriate given the positional constraints expected in Chinese. Therefore, the Chinese-language experts used the `set` command to assign new keywords to some of the modifiers. In their words, "this mechanism leads to essentially post-hoc editing of the frame to reassign inappropriately labelled keys, and, consequently, to reposition their string expansions in the surface form generation." Consider for example the `:qualifier "brief"`. In Chinese, its categorization as a `:time_qualifier` is important for determining placement in the target string. Therefore, the vocabulary entry for `"brief"` first generates a null string. It then uses a `set` command to place the Chinese translation for `"brief"` into the current frame as the key value for `:time_qualifier`. The rule is as follows:

```
brief                   "" :time_qualifier "zai4 duan3 shi2 jian1 nei3"
```

A grammar rule can then include the translation for `"brief"` in the appropriate place, *i.e.*, the position for `:time_qualifiers`.

7.4.3 Word-sense disambiguation

Word-sense disambiguation is another important issue in Chinese generation, and, as expected, the selector mechanisms have been essential in this task. At last count, the lexicon for Chinese Pinyin contained over 70 selector commands, and that number is most likely growing. Consider, for example, the problem of generating the Chinese phrase for `"late evening"`. In Chinese, it is impossible to select one translation for `"late"` that is correct for `"morning"`, `"afternoon"`, and `"evening"`. Conversely, it is impossible to select one translation for `"evening"` that is correct for both `"early"` and `"late"`. Therefore, the Chinese-language experts have used selectors to establish context and translate accordingly. Figure 7-3 contains an excerpt from the Chinese Pinyin lexicon that illustrates the use of selectors for solving this particular problem. Note that the vocabulary entries for `"early"` and `"late"` indicate that GENESIS-II should first generate a null string and then set the appropriate selectors. The vocabulary entry for `"evening"` accesses the selectors to generate the correct translations for `"early evening"` and `"late evening"`.

7.5 Jupiter Japanese

For the past year, researchers in our group have been developing MOKUSEI, a Japanese version of Jupiter [ZSP+00]. At the time of this writing, MOKUSEI already has a substantial generation knowledge base, with over 400 grammar rules and 3000 vocabulary items. Generation has been a challenging aspect in the development of MOKUSEI, because, like

```
early                  0 "" ; $:early

late                   0 "" ; $:late

evening                0 "wan3_jian1" $:early "bang4_wan3"
                       $:late "ye4_jian1"
```

Figure 7-3: An excerpt from the Chinese Pinyin lexicon.

the Chinese system MUXING, it essentially requires automating the translation of weather reports. In this section, we provide a few examples of advanced generation in MOKUSEI. (For convenience, we will transliterate Japanese throughout this section.)

7.5.1 Word-sense disambiguation

In Section 6.1, when we introduced the concept of selectors, we mentioned that the task of translating prepositions is often an exercise in word-sense disambiguation. That is, the mapping of a preposition from one language into another is often a one-to-many mapping, and so, the task of the translator is to use context to determine the appropriate translation. For example, the English word `"near"` has several translations in Japanese, depending on the context. This one-to-many mapping is evident in the following list of English-to-Japanese translations, in which the Japanese words for `"near"` are italicized.

`low near -5`	→	`saitei kion mainasu 5 do` *`kurai`*
`near record high`	→	*`hotondo`* `kirokuteki na saikoo kion`
`near 100 percent chance of rain`	→	`ame no kakuritu` *`yaku`* `100 paasento`

To distinguish between these translations, our Japanese-language expert uses selector mechanisms in both the grammar and lexicon. Specifically, the grammar rules for `record` and `percentage` contain the set-selector commands, `$:record` and `$:percentage`, respectively, as shown below:

```
record                 $:record :qualifier >predicate_template

percentage             $:percentage :qualifier :topic "paasento"
```

```
{c weather_event
   :pred {p percent_chance
            :topic {q precip_act
                        :name "rain" }
            :pred {p percentage
                       :qualifier "near"
                       :topic "100" } } }

{c weather_event
   :topic {q high
             :pred {p record
                        :qualifier "near" } } }
```

Figure 7-4: The meaning representations for "near 100 percent chance of rain" and "near record high", respectively.

When executing one of these rules, Genesis-II would first set the selector and then issue the keyword command `:qualifier`. If the key value of `:qualifier` is "near", then Genesis-II would perform a vocabulary lookup and find the following entry:

```
near                   O "kurai" $:record "hotondo"
                       $:percentage "yaku"
```

Genesis-II would then select the appropriate translation for "near" from the entry. We illustrate this process with the two meaning representations in Figure 7-4. When generating a Japanese paraphrase for the upper frame, Genesis-II would execute the `percentage` rule above, and by selecting on `$:percentage`, Genesis-II would translate "near" correctly, as "yaku". Likewise, when generating a paraphrase for the lower frame, Genesis-II would execute the `record` rule above, and by selecting on `$:record`, Genesis-II would translate "near" correctly, as "hotondo".

7.5.2 Reorganizing the frame hierarchy

Our domain experts have found that sometimes the hierarchy of a meaning representation is illogical for Japanese, just as it is for Chinese. For example, consider the Japanese translation for "rain possible in the morning". The Japanese phrase that reflects this meaning and sounds most natural is,

```
{c weather_event
   :topic {q precip_act
            :name "rain"
            :pred {p probability
                     :temp_qualifier "possible"
                     :pred {p in_time
                              :topic {q time_of_day
                                        :name "morning"
                                        :quantifier "def" } } } } }
```

Figure 7-5: The meaning representation for "rain possible in the morning".

```
 "gozenchuu  ame   no  kanoosee     ga  arimasu"
 ("morning   rain      possibility      exist")
```

However, in the meaning representation for this phrase, "morning" (incorrectly) appears within the "probability" frame, as depicted in Figure 7-5. Like its Chinese counterpart, the Japanese grammar file uses the tug mechanisms to extract the time information. Specifically, the rule for precip_act contains the tug command, <--in_time. In this way, the system resolves this word-ordering challenge and others.

7.6 Jupiter Spanish

Throughout this thesis, we have used more examples from Spanish than from any other language besides English. In Section 6.1, for example, we discussed how the Spanish knowledge base uses selector mechanisms for distinguishing between distinct translations for such words as "up" and "higher". In Section 6.3, we described the grouping rules it contains for capturing generational patterns and subsequently reducing redundancy within the Spanish knowledge base. Finally, in Section 6.6, we studied the tug mechanism it uses for correcting inappropriate hierarchy, and we also examined a way in which it utilizes push and pull commands. Since we have already discussed such a breadth of advanced generation in Spanish, we present no further examples of Spanish generation in this chapter.

7.7 Jupiter SQL

The formal language SQL is the final JUPITER language we shall examine. SQL is a challenging language to generate, because one constituent of a meaning representation may determine several substrings that must appear nonsequentially in the final SQL string. For example, the presence of the key pair, [`:name "snow"`], indicates that the SQL query should include the text:

```
geography.apt_code, city, state, country, source, day, dayspeak, snowspeak
from weather.event, weather.geography where...weather.event.apt_code =
geography.apt_code and snowspeak is not NULL.
```

In the text above, the ellipsis holds the place of text that other constituents will determine. The original GENESIS system had a very difficult time with generation sequences such as this one. Domain experts could achieve such generation only by using an idiosyncratic mechanism that engineers had inserted into the GENESIS framework for solving this problem specifically. The nature of this mechanism required the SQL vocabulary file to be extremely long and filled with duplicated text. Furthermore, the queries generated were overspecified. That is, a query would often include the same database table multiple times.

As a consequence of these shortcomings, SQL became one of the first languages we converted to the new GENESIS-II system. The resulting grammar and vocabulary files are cleaner and more compact than the originals. In fact, we literally halved the size of the knowledge base by converting it to GENESIS-II.

7.7.1 Nonsequential generation

In this section, we shall discuss the ways in which GENESIS-II simplifies and improves SQL generation in GALAXY. In particular, we look at two mechanisms for specifying nonsequential generation. We begin with the `push` and `pull` pair and conclude with the `set` command.

Push and pull

Once again, the `push` and `pull` mechanisms were adept in solving a problem for which we had not anticipated using them. In particular, we are able to use the `push` and `pull` mechanisms for performing "table joins"—the database term for including several tables in a query. We achieve this with GENESIS-II in two stages:

- In the first stage, GENESIS-II executes a grammar rule that contains a `push` command. For example, consider the following excerpt from the grammar file:

```
event_group            accumulation weather_act weather_acting
                       crisis crisis_region precipitation
                       severe_weather

event_group_template   >event_selections >event_np >--event

--event                "and weather.event.apt_code =
                       geography.apt_code"
```

  In this excerpt, the grouping rule, `event_group`, indicates that GENESIS-II can generate any of the constituents in its list by executing the `event_group_template`. Whenever GENESIS-II executes the `event_group_template`, it finishes by issuing the `push` command, `>--event`, which entails adding to the `info frame` the key pair [`--event "and weather.event.apt_code = geography.apt_code"`].

- In the second stage, GENESIS-II executes a grammar rule that contains a `pull` command. For example, consider the following excerpt from the grammar file:

```
clause_template        "select distinct" :topic >table_joins

table_joins            --update --event --weather --wunder_stats
                       --state
```

  GENESIS-II finishes executing the `clause_template` by issuing the `goto` command, `>table_joins`. It descends into the `table_joins` rule and searches the `info frame` for the deferred strings: `--update`, `--event`, `--weather`, `--wunder_stats`, and `--state`. If GENESIS-II has already executed the `event_group_template`, then the `info frame` contains a value for `--event`, and GENESIS-II can include it in the target string at this time. In this way, domain experts can specify the nonsequential generation that characterizes SQL. Moreover, this mechanism ensures that the "event table" is included only once in the query, even if multiple constituents request its inclusion.

Sets

We found that we could specify nonsequential generation by using the `set` command, in addition to the `push` and `pull` commands. For example, the vocabulary item for `"snow"` is:

```
snow                    O ", snowspeak" :post "snowspeak"
```

When GENESIS-II uses this entry, it includes `", snowspeak"` in the target string and places the key pair [`:post "snowspeak"`] in the current frame. Consequently, a grammar rule can specify its inclusion later in the generation process, as in the following rule:

```
post                    "and" :post "is not NULL"
```

So, in this way, as well, GENESIS-II renders nonsequential generation possible.

Chapter 8

Conclusion

In this thesis, we have attempted to provide a thorough introduction to our new language generation system. We began by discussing GENESIS-II's place in the broad field of language generation and in the specific context of the GALAXY conversational system. We then gave an overview of GENESIS-II and described its basic features. Finally, we discussed the mechanisms that set GENESIS-II apart, and we illustrated the system's power through several examples of sophisticated generation in the GENESIS-II framework.

When designing GENESIS-II, we approached language generation from an unusual perspective. Meaning representations in the GALAXY conversational system take on a range of forms—from simple *e*-forms to hierarchical, linguistic structures. Consequently, our challenge was to design a single framework that could handle each type of input appropriately, *i.e.,* with both simplicity and power. We wanted to create a set of mechanisms that allowed domain experts to rapidly develop template-like rules for simple domains, while giving them the power to carefully construct linguistic rules for complex domains and languages.

We knew that our framework would be used for multilingual generation, and so, we infused GENESIS-II with advanced mechanisms for propagating and using linguistic features. Furthermore, we knew that GENESIS-II would have to serve as not only a language generation system, but also as an actual translation system. In the JUPITER weather domain, for one, the GALAXY system converts English weather reports into meaning representations, which GENESIS-II must then translate into three natural languages—Chinese, Japanese, and Spanish. Therefore, we had to equip GENESIS-II with mechanisms for facilitating translation—mechanisms for word-sense disambiguation and frame reordering, for example.

In the previous few chapters, we presented the reader with a detailed description of the GENESIS-II framework itself. In Chapter 5, we described GENESIS-II's simpler commands, and in Chapter 6, we described the more sophisticated ones. Basic features, like the `keyword` and `string` commands, give domain experts the ability to specify simple template-like generation in an intuitive and straightforward fashion. On the other hand, advanced features, like the `push` command and the selector mechanisms, give domain experts the power to specify complex linguistic generation, again in an intuitive and straightforward fashion. In Chapter 7, we illustrated GENESIS-II's sophisticated generation capabilities with examples from a variety of domains and languages. We demonstrated the ways in which domain experts use GENESIS-II to surmount the linguistic challenges of word-sense disambiguation, prosodic selection, and wh-movement, to name a few. Because our framework gives domain experts such a range of capabilities, we feel that the present GENESIS-II system succeeds in bridging the divide between linguistic and non-linguistic systems.

8.1 Future Work

At this point, I can identify several ways in which I would enhance or extend GENESIS-II, if given more time. In this section, we identify and describe a few of them.

- When designing GENESIS-II's framework, we focused primarily on reworking the original system's grammar component. We made some changes to the lexicon, but we wanted to keep the lexicon backwards-compatible with the original GENESIS system. As a consequence, we inherited some of GENESIS's less intuitive syntactic forms. Given more time, I would split the vocabulary file into old and new versions and redesign some of the mechanisms. In particular, I would focus on verbal forms, such as mode and tense.

- As we expand to new domains and languages, it may become appropriate to automate the propagation of more linguistic features. Presently, GENESIS-II automatically propagates only gender and number information. However, we may need to extend GENESIS-II to handle linguistic characteristics such as case, as well.

- We also may need to implement a simpler mechanism for backwards propagation in GENESIS-II. Presently, the `push` mechanism is the sole way in which a lower-level constituent can convey information to a higher-level constituent. However, the `push` mechanism is sometimes inappropriate for such communication. For one, the `push` mechanism propagates two bits of information—a keyword (delimited by `--`) and a key value string. In some situations, a domain expert may wish to backward-propagate only one piece of information, similar to the way in which she can forward-propagate only one piece of information by using the selector mechanisms. Therefore, it may be appropriate to implement a "sticky" selector mechanism—one whose scope is wider than that of the original selector mechanism.

After a few more months of development within the GENESIS-II framework, we will be able to identify several more ways in which we might improve GENESIS-II, and our system will be tested once again. One of our primary goals in developing GENESIS-II was to create a system that was powerful and flexible, a system that could be used to overcome present challenges and that could be extended to handle those we had not anticipated. Whether we have succeeded will become clear in time.

Appendix A

Quick Reference Guide to Genesis-II

This appendix contains a quick reference guide to commands in GENESIS-II. Almost all of the commands in this section are from the grammar component; we have explicitly marked the few that are from the lexical component. We have grouped the commands into alphabetized categories.

- **Alternates**

  - *grammar_rule command1_a...commandL_a*
    grammar_rule command1_b...commandM_b
    grammar_rule command1_c...commandN_c

    Cycles through *grammar_rule*s.

- **Capitalization**

  - ($cap *command1...commandN*)

    Capitalizes the first letter of every whitespace-separated substring in the full string generated by *command1...commandN*, and adds the result to the target string.

  - ($cap1 *command1...commandN*)

    Capitalizes the first letter of the full string generated by *command1...commandN*, and adds the result to the target string.

– (`$caps` *command1...commandN*)

Capitalizes every letter in the full string generated by *command1...commandN*, and adds the result to the target string.

- **Clone**

  – (`$clone` *:source*[*:keyword*])

  Equivalent to (`$set` *:source :source*[*:keyword*]).

  – (`$clone` *:source*[*predicate*])

  Equivalent to (`$set` *:source :source*[*predicate*]).

  – (`$clone` *:source*[*$core*])

  Equivalent to (`$set` *:source :source*[*$core*]).

- **Core**

  – `$core`

  Generates vocabulary for the current frame's name, and adds the result to the target string.

- **Gotos**

  – >*other_grammar_rule*

  Adds the *other_grammar_rule_template* to the list of back-offs, descends into the *other_grammar_rule*, executes it, adds the result to the target string, and continues generation in the original rule.

- **Grouping**

  – `clause_template` *command1_a...commandN_a*

  `predicate_template` *command1_b...commandN_b*

  `topic_template` *command1_c...commandN_c*

  `key_template` *command1_d...commandN_d*

  `list_template` *command1_e...commandN_e*

  Default rules for generating from clauses, predicates, topics, keys, and lists, respectively.

- `clause_groups` *`group1_a...groupN_a`*

  `predicate_groups` *`group1_b...groupN_b`*

  `topic_groups` *`group1_c...groupN_c`*

  `key_groups` *`group1_d...groupN_d`*

  `list_groups` *`group1_e...groupN_e`*

  Grouping rules for clauses, predicates, topics, keys, and lists, respectively.

- **If**

In each case below, if the `if` command evaluates to true, then the *`then_command`* is executed, and the result is added to the target string. Otherwise, the *`else_command`* is executed, and the result is added to the target string.

N.B.: The `else_command` is optional in every case.

- `($if` *`:keyword then_command else_command`*`)`

  If the current frame contains the *`:keyword`*, then the `if` is true.

- `($if` *`predicate then_command else_command`*`)`

  If the current frame contains the *`predicate`*, then the `if` is true.

- `($if` *`:keyword`*`[`*`:key1 pred1 pred2...`*`]` *`then_command else_command`*`)`

  If the current frame contains a child frame matching one of the predicates or keywords in brackets and if the child frame contains the *`:keyword`*, then the `if` is true.

- `($if` *`predicate`*`[`*`:key1 pred1 pred2...`*`]` *`then_command else_command`*`)`

  If the current frame contains a child frame matching one of the predicates or keywords in brackets and if the child frame contains the *`predicate`*, then the `if` is true.

- `($if` *`$:selector then_command else_command`*`)`

  If the *`$:selector`* is set, then the `if` is true.

- `($if` *`condition1 &&...&& conditionN then_command else_command`*`)`

  If all of the conditions are true, then the `if` is true.

  N.B.: The conditions may have any of the forms described above.

- (`$if` *`condition1`* `||...||` *`conditionN then_command else_command`*)

  If any of the conditions are true, then the `if` is true.

  N.B.: The conditions may have any of the forms described above.

- **Keywords**

  - `:`*`keyword`*

    Searches the current frame for the *`:keyword`*, processes its key value, and adds the result to the target string.

- **Let**

  - (`$let` *`:target source`*)

    Identical to the `set` command except that changes to the `info frame` apply only until the execution of current rule is finished.

- **Lists**

  - `:nth`

    Identifies each list item.

  - `:first`

    Identifies the first item in the list.

  - `:butlast`

    Identifies each item but the last in the list.

  - `:last`

    Identifies the last item in the list.

  - `:singleton`

    Identifies the item in a singleton list.

- **Lookups**

  - `!`*`string`*

    Generates vocabulary for the *`string`*, and adds the result to the target string.

- **Pull**
  - `--`*`deferred_string`*

    Searches the `info frame` for the `--deferred_string`. If found, adds the key value of the `--deferred_string` to the target string.
- **Push**
  - `>--`*`other_grammar_rule`*

    Descends into the *`other_grammar_rule`*, executes it, defers the resulting string by adding it to the `info frame` as the key value for `--`*`other_grammar_rule`*, and continues generation in the original rule.
- **Or**
  - `(`*`command1...commandN`*`)`

    Executes each of the commands sequentially *until* one produces a string, then adds the result to the target string.
- **Predicates**
  - *`predicate`*

    Searches current frame for the *`predicate`*, processes it, and adds the result to the target string.
- **Rest**
  - `$rest`

    Generates a string for all predicates in the current frame that have not yet been processed, and adds the result to the target string.
- **Selectors**
  - `grammar_rule` *`command1...$:selector...commandN`*

    Sets the *`$:selector`* in the `info frame`.

- *lexical_entry POS "default_string" ; $:selector1 $:selector2*

  Sets *$:selector1* and *$:selector2* in the `info frame`. (*Lexicon*)

- *lexical_entry POS "default_string" $:selector "specific_string"*

  Selects *specific_string* if `$:selector` is set in `info frame`. Otherwise, selects *default_string*. (*Lexicon*)

- **Set**

  - `($set` *:target "source"*`)`

    Adds the key pair [*:target, "source"*] to the current frame.

  - `($set` *:target !source*`)`

    Generates vocabulary for the *source*, and adds the result to the current frame as the key value for *:target*.

  - `($set` *:target :source*`)`

    If the current frame contains the keyword *:source*, its key value is copied and added to the current frame as the key value for *:target*.

  - `($set` *:target :source*`[`*predicate*`])`

    If the current frame contains the *predicate* and it contains the keyword *:source*, the key value of *:target* in the current frame is set to be the key value of *:source* in the child frame.

  - `($set` *:target :source*`[`*:keyword*`])`

    If the current frame contains a child frame for *:keyword* and it contains the keyword *:source*, the key value of *:target* in the current frame is set to be the key value of *:source* in the child frame.

  - `($set` *:target :source*`[`*$core*`])`

    Searches the lexicon for the current frame's name. If found, searches the vocabulary entry for the *:source* and sets the key value of *:target* in the current frame to be the key value of *:source* in the vocabulary entry. If the *:source* is one of the linguistic keywords—`:gender`, `:number`, or `:pos`—searches for the appropriate linguistic information in the vocabulary item, and sets the key value of *:target* accordingly.

- ($set ^*:target source*)

  Copies the key value from the *source*, and adds it the info frame as the key value for *:target*.

  N.B.: The source *may have any of the forms described above.*

- *lexical_entry POS "default_string" :target "source"*

  Adds the key pair [*:target, "source"*] to the current frame. (*Lexicon*)

- *lexical_entry POS "default_string" ^:target "source"*

  Adds the key pair [*:target, "source"*] to the info frame. (*Lexicon*)

- **Strings**

  - "*string*"

    Adds the *string* to the target string.

- **Time**

  - $time

    Preprocesses the current frame as a time frame and attempts to add key values for :hours, :minutes, o+clock, and :xm.

- **Tug**

  - <--*:keyword*

    Searches the current frame's children for the *:keyword*. If found, moves the *:keyword* and its key value into the current frame, generates a string for them, and adds it to the target string.

  - <--*predicate*

    Searches the current frame's children for the *predicate*. If found, moves the *predicate* into the current frame, generates a string for it, and adds it to the target string.

  - <--*:keyword*[*key1 pred1 pred2...*]

    If the current frame contains a child frame matching one of the predicates or keywords in brackets and if the child frame contains the *:keyword*, moves the

:keyword and its key value into the current frame, generates a string for them, and adds it to the target string.

- <--*predicate*[*key1 pred1 pred2...*]

  If the current frame contains a child frame matching one of the predicates or keywords in brackets and if the child frame contains the ***predicate***, moves the ***predicate*** into the current frame, generates a string for it, and adds it to the target string.

- <--*grammar_rule*[*key1 pred1 pred2...*]

  If the current frame contains a child frame matching one of the predicates or keywords in brackets, generates a string for the child by using the ***grammar_rule***, and adds the result to the target string.

• **Yank**

- <==*command*

  Identical to the `tug` command except that the `yank` is not restricted to searching only the children of the current frame. The `yank` command performs a breadth-first search of all descendants.

Bibliography

[Axe00] S. Axelrod. Natural language generation in the IBM flight information system. In *Proceedings of the Workshop on Conversational Systems at the First Language Technology Joint Conference of Applied Natural Language Processing and the North American Chapter of the Association for Computational Linguistics (ANLP—NAACL2000)*, pages 21–26, Seattle, Washington, May 4, 2000.

[Bat96] J. Bateman. KPML development environment. Technical report, IPSI, GMD, Darmstadt, Germany, 1996.

[BH98] S. Busemann and H. Horacek. A flexible shallow approach to text generation. In *Proceedings of the Ninth International Workshop on Natural Language Generation (INLG1998)*, Niagara-on-the-Lake, Canada, August 1998.

[BH99] J. Bateman and R. Henschel. From full generation to 'near templates' without losing generality. In *Proceedings of the Workshop on Natural Language Systems at the German Annual Conference on Artificial Intelligence*, Bonn, Germany, September 13-15, 1999.

[BMTW91] J. Bateman, E. Maier, E. Teich, and L. Wanner. Towards an architecture for situated text generation. In *Proceedings of the International Conference on Current Issues in Computational Linguistics (ICCICL1991)*, Penang, Malasia, 1991.

[BS00] L. Baptist and S. Seneff. GENESIS-II: A versatile system for language generation in conversational system applications. In *Proceedings of the Sixth International Conference on Spoken Language Processing (ICSLP2000)*, October 2000.

[Bus96] S. Busemann. Best-first surface realization. In *Proceedings of the Eighth International Workshop on Natural Language Generation (INLG1996)*, Herstmonceux, England, 1996.

[CEMR99] J. Calder, R. Evans, C. Mellish, and M. Reape. “Free choice” and templates: how to get both at the same time. In *Proceedings of the Workshop on Natural Language Systems at the German Annual Conference on Artificial Intelligence*, Bonn, Germany, September 13-15, 1999.

[CMU+96] R.A. Cole, J. Mariani, H. Uszkoreit, A. Zaenen, and V. Zue, editors. *Survey of the State of the Art in Human Language Technology.* Studies in Natural Language Processing. Cambridge University Press, Cambridge, England, 1996. Commissioned by NSF (Washington, DC) and LRE (Brussels, Belgium).

[Coc96] J. Coch. Evaluating and comparing three text production techniques. In *Proceedings of the 16th International Conference on Computational Linguistics (COLING1996)*, 1996.

[Elh91] M. Elhadad. FUF*: the Universal Unifier User Manual Version 5.0.* Columbia University, 1991.

[ER92] M. Elhadad and J. Robin. *Controlling content realization with functional unification grammars*, pages 89–104. Springer-Verlag, Heidelberg, Germany, 1992.

[ER96] M. Elhadad and J. Robin. An overview of SURGE: a reusable comprehensive syntactic realisation component. In *Proceedings of the Eighth International Workshop on Natural Language Generation (INLG1996) (demos and posters)*, pages 1–4, 1996.

[FT90] R. Fawcett and G. Tucker. Demonstration of GENESYS: A very large, semantically based systemic functional grammar. In *Proceedings of the 13th International Conference on Computational Linguistics (COLING1990)*, volume 1, pages 47–49, 1990.

[FTL93] R.P. Fawcett, G.H. Tucker, and Y.Q. Lin. *How a systemic functional grammar works: the role of realization in realization*, chapter 6, pages 114–186. Pinter Publishers, London, England, 1993.

[GDK94] E. Goldberg, N. Driedger, and R. Kittredge. Using natural-language processing to produce weather forecasts. *IEEE Expert*, 9(2):45–53, 1994.

[GPS94] J. Glass, J. Polifroni, and S. Seneff. Multilingual language generation across multiple domains. In *Proceedings of the Third International Conference on Spoken Language Processing (ICSLP1994)*, Yokohama, Japan, September 18-22, 1994.

[Hal85] M. Halliday. *An Introduction to Functional Grammar*. Edward Arnold, London, England, 1985.

[Hei96] P. Heisterkamp. Natural language analysis and generation. Materials of the course held at the Fourth European Summer School on Language and Speech Communication—Dialogue systems, 1996.

[Hei99] P. Heisterkamp. Time to get real: Current and future requirements for generation in speech and natural language from an industrial perspective. In *Proceedings of the Workshop on Natural Language Systems at the German Annual Conference on Artificial Intelligence*, Bonn, Germany, September 13-15, 1999.

[Hov88] E.H. Hovy. Planning coherent multisentential text. In *Proceedings of the 26th Annual Meeting of the Association for Computational Linguistics*, SUNY, Buffalo, NY, 1988.

[Hov96] E.H. Hovy. *Language Generation*, chapter 4, pages 161–188. In Cole et al. [CMU+96], 1996. Commissioned by NSF (Washington, DC) and LRE (Brussels, Belgium).

[IKK+92] L. Iordanskaja, M. Kim, R. Kittredge, B. Lavoie, and A. Polugère. Generation of extended bilingual statistical reports. In *Proceedings of the 14th International Conference on Computational Linguistics (COLING1992)*, Nantes, France, 1992.

[IKP88] L. Iordanskaja, R. Kittredge, and A. Polugère. Implementing the Meaning-Text Model for language generation. In *Proceedings of the 12th International Conference on Computational Linguistics (COLING1988)*, Budapest, Hungary, August 22-27, 1988.

[Kuk83] K. Kukich. *Knowledge-Based Report Generation: A Knowledge-Engineering Approach*. PhD thesis, University of Pittsburg, 1983.

[LR97] B. Lavioe and O. Rambow. A fast and portable realizer for text generation systems. In *Proceedings of the Fifth Conference on Applied Natural Language Processing (ANLP1997)*, pages 265–268, 1997.

[Mat83] C.M.I.M. Matthiessen. Systemic grammar in computation: The Nigel case. In *Proceedings of the First Conference of the European Chapter of the Association for Computational Linguistics*, Pisa, Italy, 1983.

[McD80] D.D. McDonald. *Natural Language Production as a Process of Decision Making Under Constraint*. PhD thesis, Massachusetts Institute of Technology, 1980.

[McK85] K. McKeown. *Text Generation: Using Discourse Strategies and Focus Constraints to Generate Natural Language Text*. Studies in Natural Language Processing. Cambridge University Press, Cambridge, England, 1985.

[MCM96] B. Marchant, F. Cerbah, and C. Mellish. The GhostWriter Project: A demonstration of the use of AI techniques in the production of technical publications. In *Proceedings of Expert Systems 1996: Applications Stream*, pages 9–25, 1996.

[Mel88] I. Mel'čuk. *Dependency Syntax: Theory and Practice*. State University of New York Press, Albany, NY, 1988.

[MM85] W.C. Mann and C.M.I.M. Matthiessen. *Nigel: A systemic grammar for text generation*. Ablex Publishing, Norwood, NJ, 1985.

[MMA+87] M. Meteer, D.D. McDonald, S. Anderson, D. Foster, L. Gay, A. Huettner, and P. Sibun. Mumble-86: Design and implementation. Technical report, University of Massachusetts at Amherst, 1987. COINS-87-87.

[MTD96] M. Milosavljevic, A. Tulloch, and R. Dale. Text generation in a dynamic hypertext environment. In *Proceedings of the Nineteenth Australasian Computer Science Conference (ACSC1996)*, Melbourne, Australia, January 31 - February 2, 1996.

[OR00] A.H. Oh and A.I. Rudnicky. Stochastic language generation for spoken dialogue systems. In *Proceedings of the Workshop on Conversational Systems at the First Language Technology Joint Conference of Applied Natural Language Processing and the North American Chapter of the Association for Computational Linguistics (ANLP—NAACL2000)*, pages 27–32, Seattle, Washington, May 4, 2000.

[Rat00] A. Ratnaparkhi. Trainable methods for surface natural language generation. In *Proceedings of the First Language Technology Joint Conference of Applied Natural Language Processing and the North American Chapter of the Association for Computational Linguistics (ANLP—NAACL2000)*, pages 194–201, Seattle, Washington, April 29 - May 3, 2000.

[RD97] E. Reiter and R. Dale. Building applied natural-language generation systems. *Journal of Natural-Language Engineering*, 3:57–87, 1997.

[RD00] E. Reiter and R. Dale. *Building Natural-Language Generation Systems*. Studies in Natural Language Processing. Cambridge University Press, Cambridge, England, 2000.

[Rei95] E. Reiter. NLG vs. templates. In *Proceedings of the Fifth European Workshop on Natural-Language Generation (ENLGW1995)*, Leiden, The Netherlands, 1995.

[Rei99] E. Reiter. Shallow vs. deep techniques for handling linguistic constraints and optimisations. In *Proceedings of the Workshop on Natural Language Systems at the German Annual Conference on Artificial Intelligence*, Bonn, Germany, September 13-15, 1999.

[RK92] O. Rambow and T. Korelsky. Applied text generation. In *Proceedings of the Third Conference on Applied Natural Language Processing*, pages 40–47, Trento, Italy, 1992.

[RM93] E. Reiter and C. Mellish. Optimising the costs and benefits of natural language generation. In *Proceedings of the 13th International Joint Conference on Artificial Intelligence (IJCAI1993)*, volume 2, pages 1164–1169, 1993.

[RML95] E. Reiter, C. Mellish, and J. Levine. Automatic generation of technical documentation. *Applied Artificial Intelligence*, 9, 1995.

[San99] L. Santamarta. Output generation in a spoken dialogue system. In *Proceedings of the Workshop on Natural Language Systems at the German Annual Conference on Artificial Intelligence*, Bonn, Germany, September 13-15, 1999.

[SP00] S. Seneff and J. Polifroni. Formal and natural language generation in the Mercury conversational system. In *Proceedings of the Sixth International Conference on Spoken Language Processing (ICSLP2000)*, October 2000.

[vDKT99] K. van Deemter, E. Kramer, and M. Theune. Plan-based vs. template-based NLG: a false opposition?. In *Proceedings of the Workshop on Natural Language Systems at the German Annual Conference on Artificial Intelligence*, Bonn, Germany, September 13-15, 1999.

[WCM+00] C Wang, S. Cyphers, X. Mou, J. Polifroni, S. Seneff, J. Yi, and V. Zu. MUXING: A telephone-access mandarin conversational system. In *Proceedings of the Sixth International Conference on Spoken Language Processing (ICSLP2000)*, October 2000.

[ZSP+00] V. Zue, S. Seneff, J. Polifroni, M. Nakano, Y. Minami, T. Hazen, and J. Glass. From JUPITER to MOKUSEI: Multilingual conversational systems in the weather domain. In *MSC2000 Workshop*, Kyoto, Japan, October 2000.