Project Number: GFP-0507

A Natural Language Interface Using First-Order Logic

A Major Qualifying Project Report:

Submitted to the Faculty of

WORCESTER POLYTECHNIC INSTITUTE

in partial fulfillment of the requirements for the

Degree of Bachelor of Science

by

Craig Andrews　　Michael Itz　　Martin Meyer

15 December 2005

Approved:

Professor Gary Pollice, Major Advisor

Professor Dan Dougherty, Co-Advisor

1. natural language

2. first order logic

3. Lojban

Abstract

This paper demonstrates the use of the designed human language Lojban as an interface to a first order logic reasoner. We report on an implementation of a system in which representation of knowledge is constructed from and queried through natural language dialogue with a user, delivering answers verbally through speech synthesis. The implementation handles issues with extracting this representation from Lojban text, storing the resulting information in a logic-based form, and inferring and phrasing appropriate Lojban answers to user queries.

Acknowledgments

We would like to take this opportunity to sincerely thank those who assisted us through the course of this project from inception to completion. Without them, this project could not have been possible. The creators and maintainers of lojban.org provided the inspiration for this project and direction to the resources which made it possible. Professors Dan Dougherty and Gary Pollice, as project advisers, provided valuable insight in scoping the project, organizing our efforts, and presenting it in paper and presentation form.

Contents

Abstract **iii**

Acknowledgments **v**

1 Introduction **1**

2 Background **3**

2.1 Lojban 3

2.1.1 Mission/History 4

2.1.2 Letters and Pronunciation 4

2.1.3 Basic Grammar 7

2.1.4 Advantages 10

2.1.5 Uses 10

2.2 Lojban and Calculus 11

2.3 Theorem Provers 11

2.3.1 Different Approaches to Theorem Proving 12

2.3.2 First Order Logic Theorem Provers 12

2.4 Natural Language Processing 14

2.4.1 How NLP Works 14

2.4.2 Natural Language Parsers for English and Other Evolved Languages . . 15
2.4.3 Lojban Natural Language Processing 16
2.5 Speech Synthesis 16
2.5.1 Possible Applications 16
2.5.2 Terminology 17
2.5.3 Utterance Synthesis 19
2.6 Speech Synthesis Programs 21
2.6.1 Festival 22
2.6.2 FreeTTS 22

3 Methodology 25

3.1 Approach 25
3.1.1 Organization 25
3.1.2 Development Plan 26
3.2 Textual Analysis 28
3.2.1 Parse Phase 28
3.2.2 Structure Analysis Phase 29
3.2.3 Meaning Resolution Phase 30
3.3 Reasoning 31
3.3.1 Reasoning Structures 32
3.3.2 The Knowledge Base 32
3.3.3 Describing the Result 32
3.4 Textual Synthesis 33
3.4.1 Answer Generation 33
3.4.2 Fact Synthesis 33
3.4.3 Subject Synthesis 34

3.5 Speech Synthesis . . . 35
3.5.1 Selecting a Speech Synthesizer . . . 35
3.5.2 Creating a Voice . . . 36

4 Implementation 39

4.1 Textual Analysis . . . 39
4.1.1 Parse Phase . . . 40
4.1.2 Structure Analysis Implementation . . . 41
4.1.3 Meaning Resolution Implementation . . . 44
4.2 Reasoning . . . 45
4.2.1 The Data Structures . . . 46
4.2.2 The Reasoning Engine . . . 47
4.2.3 The Describe Algorithm . . . 48
4.3 Textual Synthesis . . . 50
4.4 Speech Synthesis . . . 50
4.4.1 Tokenizer . . . 50
4.4.2 Pause Generator . . . 50
4.4.3 Lexicon . . . 51

5 Results and Conclusions 53

5.1 Language . . . 53
5.2 Logic and Reasoning . . . 55
5.3 User Interface . . . 57
5.4 Text to Speech . . . 57
5.4.1 Phone Mapping . . . 57
5.4.2 Pronunciation . . . 58

5.4.3 Pausing 58

5.4.4 Other Potential `UtteranceProcessor`s 59

6 Future Work **61**

6.1 Language 61

6.1.1 Additional Analysis Features 61

6.1.2 Enhanced Synthesis 62

6.2 Reasoning 62

6.2.1 Tense 62

6.2.2 Commands 63

6.2.3 Epistemology 63

6.2.4 Mathematics 64

6.3 Text to Speech 64

6.3.1 Part of Speech Tagging 64

6.3.2 Syllables and Stress 65

6.3.3 Additional UtteranceProcessors 65

A Pronouncing Lojban **67**

Glossary **71**

List of Figures

3.1 Layered Organization 26
3.2 Sample Parse Tree 29

4.1 Analysis Package, General Organization 41

5.1 The Final User Interface 56

List of Tables

A.1 List of Lojban vowels 67

A.2 List of Lojban Diphthongs 68

A.3 List of Lojban consonants 69

Chapter 1

Introduction

Natural language processing is a field of increasing importance with growing applications in search technology, machine translation, command and control, and general human-computer interaction. In many areas, natural language affords an intuitiveness and flexibility in human interaction difficult to achieve through specialized command languages or even graphical interfaces. A well-developed natural language interface could drastically reduce the learning curve for an application, allowing a new user to quickly take advantage of features with the same ease that they naturally communicate with other people.

Attempts to realize such natural language interfaces often focus heavily upon the major difficulties produced by the inherent ambiguity and frequent irregularity in grammar and vocabulary in evolved languages such as English, Spanish, or German. Sentences may often yield more than one interpretation, and words may have very different meanings depending upon their context. Solutions to these troubles often require statistical and heuristic methods which must be tailored to the application and, necessarily, introduce some error. [13] The time investment required makes natural language infeasible for applications of even moderate size, while the margin of error introduced renders it unacceptable for critical tasks. These factors erect a considerable ‘barrier to entry’ around the field of language processing and discourage experimentation with natural language in more general applications. While much research has been conducted in the field of natural language processing, most of the effort is absorbed by these barrier problems of ambiguity and irregularity.

Though ambiguity and irregularity plague evolved languages, they need not be inherent in the concept of natural language itself. The designed language Lojban, developed for research in linguistics, eliminates almost all ambiguity and is highly regular in its formation. [8] Because of Lojban’s highly regular nature, the initial step in interpreting statements more closely resembles programming language parsing than it does processing of evolved languages. Using Lojban as a communication medium, experimental work with natural language then can examine a wider range of applications without troubling with statistical or heuristic methods. Translating the parsed statements into a knowledge representation and performing work upon it then becomes the major remaining issue. [7]

To demonstrate one such approach to computing with natural language, this project developed an example application using Lojban as a human interface to a first order logic reasoning engine. Additionally, to stress the potential for intuitive verbal interface through natural language, the project included development of a Lojban speech synthesis library for inclusion in the demonstration system. The result outlines a technique by which a representation of knowledge may be constructed from and queried through natural language dialogue with a user. It considers issues involved with extracting this representation from Lojban text, storing it in a logic-based form, and inferring and phrasing appropriate Lojban answers to queries posed against it in Lojban.

Chapter 2

Background

This section provides background information on the issues and technologies addressed by this project. It explains the history of Natural Language Processing (NLP), principles and structure of Lojban, the principles and types of theorem provers, and the concepts involved in speech synthesis.

2.1 Lojban

Many of the issues commonly experienced in developing natural language systems stem from the complexities, asymmetries, and irregularities which appear in evolved human languages. Many constructions in English, both written and spoken, resolve to multiple unrelated meanings that human speakers correctly choose only from intuition, if at all. Moreover, the methods by which English translates thoughts into utterances are clumsy in their linkages to the underlying implicit logical assertions, often making difficult any determination of precisely what has been said. These difficulties consistently confound systems which aim for NLP. However, these problems belong to English and other common evolved languages and not to the concept of human language itself. Lojban, a carefully designed spoken language, demonstrates language which is highly regular, precise, and, most importantly, unambiguous in construction. [8]

2.1.1 Mission/History

Lojban originated as the Loglan Project in 1955. Developed by Dr. James Cooke Brown, the project was an experimental trial of the Sapir-Whorf hypothesis, which proposes that the grammatical forms of a language influence the understanding and forms of thought of its speakers. By simultaneously allowing for more precise and less constrained grammatical forms, the project hoped to demonstrate that Loglan speakers could clearly form thoughts awkward for English speakers. Loglan was first made public in a Scientific American article [5] in 1960 and has since been revised as Lojban by the founders and members of The Logical Language Group. [8]

Lojban is an artificial language designed for daily use between human speakers, much like Esperanto. However, Esperanto aimed primarily for facilitating international communication with forms based upon traditional languages. The Lojban project instead has modeled its linguistic constructs after the forms of predicate logic and has carefully chosen its vocabulary for unambiguity by computer algorithm. These choices cause Lojban to express many interesting features which make it conducive to human-computer interaction. [8]

2.1.2 Letters and Pronunciation

The Lojban alphabet consists of 26 characters coming primarily from the Latin (Roman) alphabet. Standard ASCII characters were selected to facilitate easier storage on computers. The letters used are **' , . a b c d e f g i j k l m n o p r s t u v x y z**, which is all but three letters from the English alphabet and three English punctuation characters. The letters ‘h’, ‘q’, and ‘w’ are omitted because the sounds they create can be made through other Lojban letter combinations.

The letters ‘'’, ‘,’ and ‘.’ are used to modify the sounds of the surrounding letter. The ‘ch’ in **ch**urch and the ‘j’ in **j**udge are considered to be two consonant sounds and are represented in Lojban by the consonants ‘tc’ and ‘dj’ respectively. [8]

Vowels and Consonants

The Lojban vowel set consists of the five English vowels (‘a’, ‘e’, ‘i’, ‘o’, ‘u’) plus the letter ‘y’. All other letters are consonants.[8]

Special Pronunciations

The majority of the Latin letters in the Lojban alphabet sound like their English counterparts with some exceptions. The International Phonetic Alphabet (IPA) symbol which indicates the pronunciation of each letter can be found in Appendix A. The interesting characters to note are ‘'’, ‘,’, ‘.’, ‘c’, ‘g’, ‘j’, ‘r’, ‘s’, ‘x’.[8]

' Sounds like a short, breathy English ‘h’. It is used for smoothing the separation between vowels.

, Is used to mark syllable breaks in uncommon places. No pause is implied (or allowed) by this character, but the ‘h’ sound of the apostrophe can be used at the speaker's discretion

. Is an optional reminder of a mandatory pause. It exists for the benefit of the reader, since the pause is required regardless of whether the ‘.’ is present or not.

c Always sounds like the **sh** in **sh**ip.

g Always sounds like the **g** in **g**um.

j Always sounds like the **s** in mea**s**ure.

r has a number of possible sounds in various languages, all of which are valid in Lojban.

s Always sounds like the **s** in cat**s**.

x Doesn't have a corresponding phone in English, but it sounds like the **ch** in Ba**ch**.

Pausing

There are seven rules for pausing in Lojban, all of which are outlined in [8]:

1. Any two words may have a pause between them, but it is always illegal to pause in the middle of any word because such a pause would imply a word break.
2. Every word ending with a consonant must be followed by a pause.
3. Every word beginning with a vowel must be preceded by a pause.
4. Every *cmene* must be preceded by a pause unless the preceding word is of one the *cmavo*, *la*, *lai*, *la'i*, or *doi*.
5. If the last syllable bears the stress and the following word is a *brivla* then they must be separated by a pause.
6. A *cmavo* of the type "Cy" (any consonant followed by the vowel 'y') must be followed by a pause unless another "Cy"-form *cmavo* follows.
7. Non-Lojban words must be preceded and followed by pauses.

Syllabication

Syllable boundaries in Lojban are found after a vowed or between two consonants. When a consonant double is encountered the first consonant is part of the previous syllable and the second is the beginning of the next. When a consonant triple is encountered a syllable break is required between the first and second and optional between the second and third (which could leave one consonant as an entire syllable). The apostrophe and comma also imply syllable breaks.[8]

2.1.3 Basic Grammar

Lojban utterances consist of assertions of facts as well-formed formulas of predicate logic. utterances constructed by a speaker are, by default, interpreted as literally true. This rule includes use of metaphor or hyperbole as they appear in English, though forms exist to grammatically express and mark statements as figuratively, not literally, true. Summarized here from [8], John Woldemar Cowan's *The Complete Lojban Language*, are the very basics of forming Lojban sentences. The complete grammar is much more involved.

Vocabulary

Lojban vocabulary is composed of three major classes: root words, called *gismu*, structure words, called *cmavo*, and names (similar to proper nouns in English), called *cmene*. Specialized forms also exist for inclusion of foreign words into the language. Root words are all five characters long, starting with a consonant and ending with a vowel, and possess exactly one pair of consecutive consonants. There are approximately 1350 root words which may be combined through various compound word forms to achieve a wide range of expression. Lojban defines the root words in terms of the relationship they imply between the one to five objects they appear with. Names can be formed along much less constrained rules than root words, but they can only legally appear in a few grammatical forms and must always end with a consonant and a full stop. Structure words are divided into classes which have common formation according to the needs of their function, but they must always end in a vowel. Words of any type which begin with a vowel must be preceded by a full stop.

Sentences

Root words and names may be assembled using structure words to construct the equivalent of sentences, called *bridi*. A *bridi* consists of a subject and objects (called *sumti*) which are related to each other by a *selbri* that is something like a verb in English. The *selbri* may be a single root word or a compound of several words to achieve a more precise meaning. The subject is, in the simplest form, placed before the *selbri*, and up to four objects may be listed after it. With additional structure words, the sentence components can be rearranged to any order to change emphasis, for example, to duplicate the effect of passive voice in English.

It may be helpful to imagine that a *bridi* is much like a function call in a programming language. The *selbri* is the function's name, the subject and objects are its arguments, and the *selbri* will have a signature dictated by its construction. The invocation of the function asserts to the listener that some relationship exists between the objects as defined by its signature.

Subjects and Objects

Much of the technical complexity of Lojban lies in the mechanism to identify the subjects and objects of a sentence, the *sumti*. Lojban has no direct analogue to an English noun. Instead, the four major ways for a speaker to achieve this are by name, by veridical description, by nonveridical description, and by *prosumti* (much like a pronoun).

Identification by name is the simplest. It is achieved by prepending the structure word article *la* to the *cmene* for the object. A speaker might therefore refer to Arthur (using Lojban phonology and morphology) as "*la .artur.*", or Boston as "*la bosten.*". Alternatively, a speaker might use a root word to figuratively refer to an object: as Arthur comes from the Latin word for bear, a speaker might refer to him as "*la cribe*", or Bear (more literally Is-a-bear).

Identification by veridical description is similar to the usage of a noun with the article *a* in English, though carries with it the existential quantification of predicate calculus, and grammatically implies that some object being described really exists. Veridical descriptions are constructed by prepending the Lojban article *lo* to a root word describing the object. A speaker might say "*a bear*" as "*lo cribe*", for example, but the speaker is then literally (unlike above) asserting that this thing being described is a bear. Arthur may be referred to as "*a ruler*" or "*lo turni*".

Identification by nonveridical description, using the Lojban article *le*, is most similar to the English article *the*, though it carries the additional weight of for-all quantification to the sentence within the context of the conversation. "*The bear*", or "*le cribe*", refers to all bears previously mentioned by the conversation or obvious in the environment of the speakers and asserts that for all those bears the sentence is true. Thus, the nonveridical form is useful only in a situation where it is obvious to the listener to what it refers.

Both veridical and nonveridical description can be made more precise through the addition of relative clauses. Relative clauses further restrict the objects to which the sentence applies. In the above example, if there was a black bear mentioned and a brown bear mentioned, "*le cribe*" would refer to both bears. However, by appending to the nonveridical description the requirement that the bear be black (with root word "*xekri*") with the relative clause structure word *poi*, the black bear alone can be indicated by "*le cribe poi xekri*". Similarly, appending relative clauses to veridical descriptions more precisely refines the assertion being made about what exists: "*lo cribe poi xekri*" specifies not just a bear but a bear which is black.

Finally, a speaker can be more concise by making references to objects through pronouns. Lojban has a suite of pronouns of different types, including ones which are explicitly assigned (like variables in programming languages), refer to previously mentioned objects or objects within the same sentence (somewhat like the English *it* but with clearly defined referents), or refer to things in the context of the conversation (like the English *you* or *this*).

Putting these together, a speaker can fill in the places of the sentence to form a complete thought. For example, using the root word "*viska*", which asserts that the first parameter visually perceives the second (and has an optional third parameter omitted here to indicate the conditions of the perception), a speaker might say "*Arthur sees the black bear*" as "*la .artur. viska lo cribe* *poi xekri*".

2.1.4 Advantages

This precise construction has great benefits for human interaction as, while maintaining (or even exceeding) the expressive quality of English, Lojban eliminates all unintended ambiguity and precisely defines areas where ambiguity may remain. Its grammar, vocabulary, and pronunciation are highly regular and thus more easily processed by computer than evolved languages. As Cowan stresses, Lojban is designed both for cultural neutrality and ease of learning. Lojban, thus, has much to recommend it for purposes of human-computer interaction. [8]

2.1.5 Uses

Areas in computer interaction which could benefit from a common interface language such as Lojban include contract languages, interactive dialogs with applications, expert systems, and end user programming and customization. In the business arena, Lojban might be used to specify service contracts which can be computer evaluated and applied and yet easily edited by individuals who can read and write Lojban though have no programming experience. In every day usage, Lojban could be used to facilitate interaction with any number of applications through voice or text interface so that a Lojban speaker might address a software package in human language. A Lojban interface to an expert system would allow individuals to pose queries and receive responses just as they would speak to another human. Lojban may also allow for customization of systems such as home automation packages by allowing the end-users to describe what they would like to occur under which circumstances in plain spoken language. [7]

2.2 Lojban and Calculus

Lojban's grammar is based upon First Order Logic (FOL) as much as possible. The grammar, as explained in Section X, is not entirely FOL, as it includes meta linguistic constructs, as well as other features not found in standard FOL. [7] However, although more complex forms may not be captured, a FOL theorem prover may be used to provide a rudimentary logical back-end for a computer system that uses Lojban for its human interface as a proof of concept.

Propositional calculus is less powerful than FOL. FOL adds the qualified statements "*there exists an x such that and for all x, it is true that...*" to propositional logic. Without these statements, solving propositional logic statements is decidable problem. However, it is NP complete, so propositional logic problems can require up to exponential time to solve. [9]

The set of valid sentences of FOL are, on the other hand, recursively enumerable. Any valid FOL statement may be proven given limitless resources. However, testing validity is not decidable, so the theorem prover is not guaranteed to halt while attempting to determine a statement's validity. This can become a major problem for FOL in comparison to propositional logic. [9]

2.3 Theorem Provers

Combining that knowledge that the expressive power of FOL is so much greater than that of propositional logic and Lojban is specifically designed with FOL in mind, the team decided that the benefits of using FOL far outweigh its downsides. The theorem prover therefore constitutes a viable back-end to a Lojban logic system for, as previously stated, Lojban's grammar is based on FOL and designed to be precise and non-ambiguous. Transforming written Lojban to a computer interpretable form naturally results in FOL statements that can be fed to a theorem prover in order to answer questions and interpolate information as needed in the course of user interactions.

2.3.1 Different Approaches to Theorem Proving

There are two major categories of theorem provers, those that are interactive and those that are fully automated. Interactive theorem provers require the user to guide them in their search for a proof and thus yield clear, precise results. [9] Interactive theorem proving may be a perfectly valid approach for certain applications, however in the pursuit of modeling more human-like interaction with the system, the group focused on fully automated theorem provers.

Fully automated theorem provers do not interact with the user at all. They yield far less elegant proofs and explore proof paths that the human user of an interactive prover would intuitively know to be fruitless, potentially wasting time. However, for the vision of the human-computer interaction pursued in this project, time and proof elegance are acceptable trade-offs for autonomy. [9]

2.3.2 First Order Logic Theorem Provers

There is an assortment of FOL theorem provers available under various free and commercial licenses. For this project, a free product was necessary so that progress might not be stymied by licensing acquisition and associated issues.

Prolog

Prolog is a logic programming language created by two professors, Alain Colmerauer and Robert Kowlaski, around 1972. The language is very mature and easy to use, and there are many free implementations available. However, a significant restriction built into the language requires that all Prolog statements be formulated as Horn clauses, statements in the form of at most one positive literal combined with negative literals by means of disjunctions. This limitation makes evaluation faster and simpler. Though, while Prolog is faster, more mature, and very available, its Horn clause restriction renders many Lojban constructs difficult, or impossible, to express.

Prolog is a logic programming language, not a theorem prover, and is not powerful enough for this application. [6]

Otter

Otter is an automated FOL theorem prover developed by Argonne National Laboratory. It is a very popular piece of software among logicians because of the power and speed which result from its many years of development and optimization. The software is released into the public domain, thus no copyright issues are present.

Otter, though, proved less than ideal for this project. First, Otter does not maintain state. For each operation, the database must be fed into Otter with one question which will then return one answer. If another query is posed, instead of applying it against an existing knowledge base a new base must be created. Additionally, Otter is written in C, making it difficult to interface with the intended programming language for the project, Java. Finally, in order to gather a complete result, both Otter and its sister program, Mace, must operate upon the knowledge base, adding complexity to the potential implementation. [11]

JTP

Java Theorem Prover (JTP) is developed by Stanford University. It is modular, powerful, and written in Java for easy extensibility, allowing for easier integration of this particular theorem prover than the other candidates. JTP maintains state between queries and thus makes multiple queries against the same data easier and faster. Furthermore, JTP performs the functions of both Otter and Mace, conducting searches for proofs and models through the same interface. The license, which essentially states that the code is readable but not redistributable while compiled object code is redistributable, is acceptable for use in this project. [14]

JTP's primary downside is in its performance. Otter, which is extremely optimized and written in clean C, is reportedly faster than JTP, which did not have speed as a high priority in its design. However, in the project team's opinion, JTP's aforementioned benefits far away this relatively minor downside.

2.4 Natural Language Processing

NLP is a field that has been developing almost since the dawn of computer science. Asking the computer questions and receiving useful answers, or giving it commands and having the computer take action, has appeared in much science fiction. However, the existing NLP systems use evolved languages, and experience a number of setbacks because of this decision. In order to understand why Lojban is a good candidate for a NLP system, one first has to understand why Lojban may be a better language for NLP than other languages.

2.4.1 How NLP Works

Natural language processors first need to break down sentence structures into logical constructs in order for further analysis to be done. This often involves creating some type of formal grammar, such as Backus-Naur form (BNF), to express the sentence structure. Languages tend to be too complex to be expressible as BNF grammars, so extensions are often made or other more powerful grammars are used. The next step in natural language processing requires applying context to the statement to understand what the user meant. This step is very much non-trivial, as much information is obtained from context and there are many ambiguities that may not be easily resolved. Context not only includes the previous utterances stated, such as a pronoun which the computer must dereference, but also information from the environment and cultural conventions. For example, the question, "*Is there water in the fridge?*" may be understood to mean two things: "*is there molecular water in the refrigerator?*" or "*is there at least one bottle* *of water in the refrigerator?*" In specific circumstances, either may be the correct interpretation, but according to typical culture, the second meaning is usually assumed. A computer requires a vast amount of contextual knowledge, specific to regions and conversational topic, in order to be useful for circumstantial disambiguation. [13]

Finally, similar problems recur as the system answers the question and must indicate the subjects within its reply. It must take into account the same cultural and contextual knowledge base as it selects the proper grammatical structures and vocabulary to use to formulate a proper and accurate reply which is not overly verbose. Asked, "*Who is Arthur?*" the system might best answer "*Arthur was a legendary king of England,*" but could also answer, with too much information, "*King Arthur is an important figure in the mythology of Great Britain, where he appears as the ideal of kingship in both war and peace.*" [13]

2.4.2 Natural Language Parsers for English and Other Evolved Languages

Evolved languages, such as Spanish, English, German, Italian, Latin, and all other commonly spoken human languages are riddled with issues that make natural language processing extraordinarily difficult. As above, the first step in natural language processing is to break down sentences into a structure by use of their grammar. However, evolved languages have ambiguous grammars that require context as well as guesswork to complete this breakdown. The classic instance of this problem is the utterance "*pretty little girls' school.*" There are many interpretations of this utterance: the school is little, the girls are little, the girls are pretty, the school is pretty, etc. There is no systematic way to decide if any one interpretation is the intended one. [8]

The vocabulary itself in evolved languages is also ambiguous. "*Pretty*" can be a modifier that means somewhat, such as in "*pretty close,*" or it can mean attractive, such as in "*pretty girls.*" For another example, consider the word "*flies*": "*flies*" can be a verb, indicating movement through the air, or it can be a plural noun, referring to winged insects. [8]

2.4.3 Lojban Natural Language Processing

Lojban was designed from the ground up to be as unambiguous as possible. Each word in the Lojban vocabulary refers to exactly one specific concept or object. No ambiguities such as those previously discussed exist. Clearly, this requires much less work of the natural language processor: it need not apply context to find the meaning for a word, it simply needs to look it up in a dictionary. Likewise, the Lojban grammar was very carefully constructed to be completely unambiguous; optional separators and terminators become mandatory when an utterance would otherwise be ambiguous. In this fashion, a computer can accurately gather the meaning of an utterance without cultural or contextual knowledge. However, on the output end, the computer must still be sure to formulate its reply in a sensible way to provide a useful, concise, answer that does not contain information that the user does not really care about. Clearly, this phase is still very complex; however, Lojban still greatly reduces the problem size of natural language processing when compared to the evolved human languages. [7]

2.5 Speech Synthesis

This section provides background information on the speech synthesis. It explains possible applications, related vocabulary, and common methods of synthesizing speech.

2.5.1 Possible Applications

speech synthesis systems have a number of applications in the real world. They can be used for announcing the next stop on a subway system, to provide auditory feedback at an automated self-checkout line in a supermarket, to read aloud a frequently-changing weather forecast, or to read aloud the text on a computer screen for a visually impaired user, just as a few examples. The advantages of hearing instructions aloud are numerous. Having a subway destination read slowly, clearly, and loudly by a speech synthesizer eliminates hassles for both passengers and operators. Hearing instructions at the same time as they are printed on a touchscreen allows the patron to more quickly respond to the prompt, resulting in faster checkout time overall. Automated reading of weather statements, which can repeat quite frequently and change just as often, makes changing the weather forecast less costly and facilitates more up-to-date weather information. Reading of screen content aloud is one of the few methods of providing textual content to people who are not able to read directly from the screen, and it can also be used for situations where screens are not available and output can be provided by audio only.

2.5.2 Terminology

There is a bit of vocabulary required to understand some of the issues involved in speech synthesis. Several terms will be introduced in this section to explain some of the issues in speech its synthesis.

Utterance

An utterance in speech synthesis is an entire portion of text which is intended for synthesis. The utterance does not need to be a single phrase or sentence; it could be an entire document, a single paragraph, or even a single word. Regardless of the length of the utterance, it is the responsibility of the text to speech system to convert the utterance into pronounceable words (if it needs to) so that the utterance can be read aloud. In English an utterance is normally broken into a series of phrases, each with a pause after it, but this is not strictly required or even necessary. It depends on the implementation of the voice.

Token

Tokenization is the process of converting a string into a series of *tokens*. Each token corresponds to a single whitespace-separated item from the utterance. It is the job of the *tokenizer* to make sure that any tokens that are not pronounceable words are turned into speakable words before being passed off. Each token in the utterance could start as a word, number, abbreviation, or punctuation, and a tokenizer needs to know how to convert these into proper speakable words and pauses. [3]

The tokenizer must convert numbers into readable words in order to make words pronounceable. For example, if "*there are 10 types of people in the world*" was encountered in an utterance, the tokenizer would need to convert it to "*there are ten types of people in the world*" so that the number in the sentence can be read aloud. Abbreviations are similar, since "Mr." would have no proper way to be pronounced unless it was properly converted to the full word "Mister."

Phoneme

A phoneme is a sub-word unit of language that is intended to be communicated. In a purely phonetic language, one phoneme would correspond directly with exactly one phone [4], but in most languages the phoneme can map to one or more phone depending on surrounding letters and the word's part of speech. Lojban is fairly close to a phonetic language with the exception of diphthongs, the 'ch' sound from English, and the 'g' in **g**em. See section 2.1.2 for details on Lojban pronunciation.

Diphthong

A *diphthong* is a sequence of two vowels whose juxtaposition alters the pronunciation. For example, in English the word "*brought*" the diphthong 'ou' could be mapped to the English phoneme 'aw', but considered independently it might be mapped the phonemes 'oh' and 'uw'.

diphthongs present a difficult analysis problem for speech synthesizers because they must try to determine the best phones to apply using complex rules.

Phone

A phone is an actual pronounced sound in a language. It is distinct from a phoneme and diphthong in a very specific way: a phone is the actual sound that is produced to audibly represent a phoneme or diphthong. A phoneme can correspond to different phones (maybe one, maybe more) depending on accent or dialect of the speaker. For example, the word "*car*" has a single syllable, only one vowel, and several possible pronunciations. For standard U.S. English the phones for "*car*" would be something like 'k ah r', while someone from Boston might say 'k ah hh'.

2.5.3 Utterance Synthesis

"Applications for text to speech output fall into four categories:

1. Single word responses
2. Limited set of messages within a strict format
3. Large, fixed vocabulary with normal language syntax
4. Unrestricted text to speech" [2]

Though these categories are fairly broad, they help determine the best method for synthesize the speech for the current application. The first two cases are discussed next, followed by the third and fourth in their own sections.

Simple Words and Messages

Single-word messages can be prerecorded and played back as-needed by a speech synthesis system. In systems which speak a defined set of possible words one at a time, this method will produce perfectly acceptable sounds for all possible utterances with minimal space and processing power required to produce the sounds. The next level of complexity arises when words are concatenated together: although the words may sound well enough independently, stringing them together to create a sentence can result in disjointed and awkward-sounding utterances. When working with prerecorded words it is difficult to simulate speech features such as coarticulation, which is the change in a word's pronunciation based on context, intonation, rhythm, and stress pattern. For systems with a limited number of predictable phrases one possible solution is to prerecord entire phrases for later playback to users. The space required for this solution depends on the number of possible phrases, but it will not be quite as efficient as recording single words because words would need to be re-recorded in each phrase that they are used in. [2]

Large, Fixed Vocabulary

For larger systems which have a fixed vocabulary it may still be feasible to store recorded waveforms for each possible word, but in a normal language syntax it is infeasible because it would require too much storage space. In order to smooth speech output some sort of waveform analysis is required to make the end of the current word blend with the beginning of the next. A number of methods have been developed to accomplish this but they are beyond the scope of this report. Disadvantages of using prerecorded words include the infeasibility of large vocabularies and the fact that it is easier to adjust word stress, length, and pronunciation with smaller segments. [2]

Concatenation

For systems with a large, unbounded vocabulary and full language syntax there are two methods of synthesizing unrestricted utterances: phonemic synthesis-by-rule and prerecording component pieces of words. [2] Although synthesis-by-rule can produce extremely realistic utterances, the difficulty in creating such a synthesizer makes it impractical for most applications. By far the most common method of creating a voice which concatenates prerecord word units for later synthesis.

A number of possible segments have been investigated at length for the basic phoneme unit for prerecording. The three units investigated to the greatest extent are syllables, decasyllables, and diphones. A syllable consists of a vowel root with consonants on either side. This seemed like a natural choice for researchers at first because all words break down into a distinct group syllables, but it becomes difficult to determine which consonant constitutes the actual syllable boundary. Even worse, it is difficult to correct coarticulation across syllable boundaries and the syllable set for English is very large. No syllable systems existed in 1987 when [2] was published. Demisyllables are defined as half of a syllable, and using this type of segment could potentially solve the problem of the large list of syllables since there are reported to be less than 1000 demisyllables required to synthesize any English utterance. [2]

The use of diphones, or half of one phone followed by half of the next phone, was first proposed as a unit for speech synthesis in 1958. diphones constitute a natural unit for synthesis because coarticulatory influence of one phone does not usually extend much further than halfway into the next. [2] Research published in 1968 showed that highly intelligible synthetic speech could be fashioned from about 1500 prerecorded diphones, consisting of forty English phones combined with almost any other phone. Duration, intensity, and fundamental frequencies could be adjusted independently by signal processing to adjust stress, intonation, and rhythm.

2.6 Speech Synthesis Programs

A number of speech synthesis systems exist but not many of them have the full capabilities that were required for creating a new voice to speak Lojban. In this section two systems are introduced that were considered as possible speech synthesis engines. Although some technical specifications are provided in this section, the reasoning behind our final selection is discussed in section 3.5.1.

2.6.1 Festival

“Festival offers a general framework for building speech synthesis systems as well as including examples of various modules. As a whole it offers full text to speech through a number APIs: from shell level, though a Scheme command interpreter, as a C++ library, and an Emacs interface. Festival is multi-lingual, we have developed voices in many languages including English (UK and US), Spanish and Welsh, though English is the most advanced.

The system is written in C++ and uses the Edinburgh Speech Tools for low level architecture and has a Scheme (SIOD) based command interpreter for control. Documentation is given in the FSF texinfo format which can generate a printed manual, info files and HTML.” [3]

2.6.2 FreeTTS

“FreeTTS is a speech synthesis system written entirely in the Java programming language. It is based upon Flite, which is a small run-time speech synthesis engine developed at Carnegie Mellon University. Flite is derived from the Festival Speech Synthesis System from the University of Edinburgh and the FestVox project from Carnegie Mellon University.

This release of FreeTTS includes:

- Core speech synthesis engine
- Support for a number of voices:
    - an 8khz diphone, male, US English voice
    - a 16khz diphone, male US English voice

    - a 16khz limited domain, male US English voice

- Partial support for JSAPI 1.0
- Extensive API documentation
- Several demo applications”[1]

Chapter 3

Methodology

3.1 Approach

To test Lojban's viability in applications in human-computer interaction, this project implemented a first-order reasoning engine driven at the natural language level. The resulting system contains Lojban-tailored speech synthesis to pronounce aloud the dialog carried out with the user and thus emphasize that the interface occurs through spoken, human-oriented language.

3.1.1 Organization

The prototype system is divided into four major layers. The topmost layer contains the GUI and the speech synthesis plugin for user interaction. The knowledge base package supports the interface layer with its plugin architecture and, more importantly, serves as a facade for the remainder of the system. The knowledge base package drives its input and output through the language layer containing the analysis and synthesis packages to interact with the lowest level reasoning layer at the base of the system.

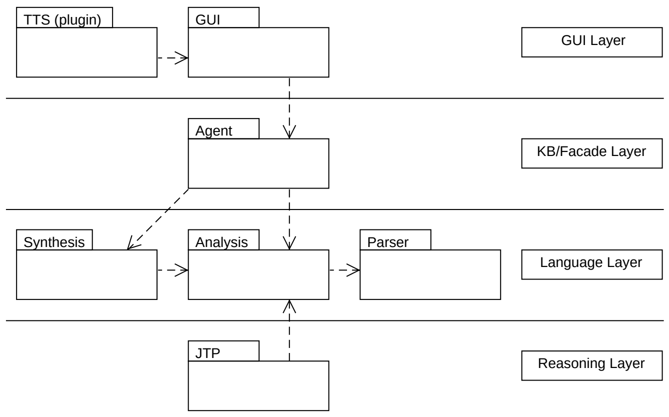


Figure 3.1: Layered Organization

This organization was chosen to allow for flexibility in future work. For specialized or more advanced reasoning, the reasoning layer could be swapped out and replaced with a more powerful or specialized solution. The uppermost interface layer could be augmented by interface plugins, replaced entirely, or serve as an interface for embedding in a larger application.

3.1.2 Development Plan

The Lojban grammar includes far too many features to implement within the scope of this project. However, the team decided that the project could implement the core features for assembling sentences and identifying their subjects and then slowly expand upon those features to support more complex language constructs. This project approached iteratively implementing the language analysis process as follows:

iteration 1 simple sentences with named (*la*) and literal (*zo*) objects

iteration 2 simple sentences with veridical (*lo*) and nonveridical (*le*) objects

iteration 3 complex sentences with connectives

iteration 4 relative clauses (restrictive and nonrestrictive) and possessive forms

iteration 5 query/question *prosumti*

iteration 6 assignable, reflexive, and anaphoric *prosumti*; anaphoric *probridi*

iteration 7 synthesis of results back into Lojban

Many language features were not included. Some were omitted because they do not have value expressible through first order logic, particularly attitudinals and metalinguistic *prosumti*. Others, such as complex relations through the *lujvo* and *tanru* constructs, were omitted because, while important for expressive power, they add little to the computational power of the language. Parallel to the development of increasingly sophisticated grammar, the project advanced the reasoning layer iteratively through its own stages:

iteration 1 save veridical references and answer direct queries with objects for answers

iteration 2 answer indirect queries through logical inference with objects for answers

iteration 3 save nonveridical references and extend query support

iteration 4 describe result uniquely using knowledge about it rather than a unique identifier

iteration 5 describe result efficiently and without repeating parts of the question

iteration 6 answer questions demanding relations or boolean values

iteration 7 answer multi-part questions; answer in complete sentences

Likewise, the speech synthesis package was developed in parallel with the analysis and reasoning engines, ultimately flowing into the construction of the the uppermost layer to provide a user interface. This interface exposed the plugin architecture for experimenting with applications of the system, one of which includes the speech synthesis.

3.2 Textual Analysis

The *Analysis* package is responsible for extracting meaning from a piece of Lojban text. It operates in three phases. First, the text is parsed using a formal grammar to generate a tree of tokens. Next, those tokens are used to identify the meanings of larger syntactic constructs, approximately the Lojban equivalents of clauses. Finally, those constructs are visited by a resolver class (defined as an interface here and implemented by the reasoning package) that they manipulate to represent the information they contain in the reasoner's language.

3.2.1 Parse Phase

In *The Complete Lojban Language*, Cowan provides both YACC and EBNF grammars for defining legal Lojban utterances, however additional post-processing is required for properly handling many constructs, particularly optional terminators [8]. Robin Lee Powell has done additional work to produce a more powerful Parsing Expression Grammar to represent Lojban. Because PEG allows for ordered choices, rather than the unordered choices of YACC and EBNF [10], Powell's grammar can handle directly those cases which required post-processing in Cowan's solutions.

For the first analysis pass, the system utilizes a parser produced by Robert Grimm's Rats! parser generator using Powell's PEG grammar. Using the grammar, the parser verifies the statement is legal Lojban and annotates its syntactic structure. For example, the phrase "*lo cribe poi xekri cu klama*" would be annotated as:

The output of the parser is then used to build an in-memory n-tree of tokens for the next analysis phase.

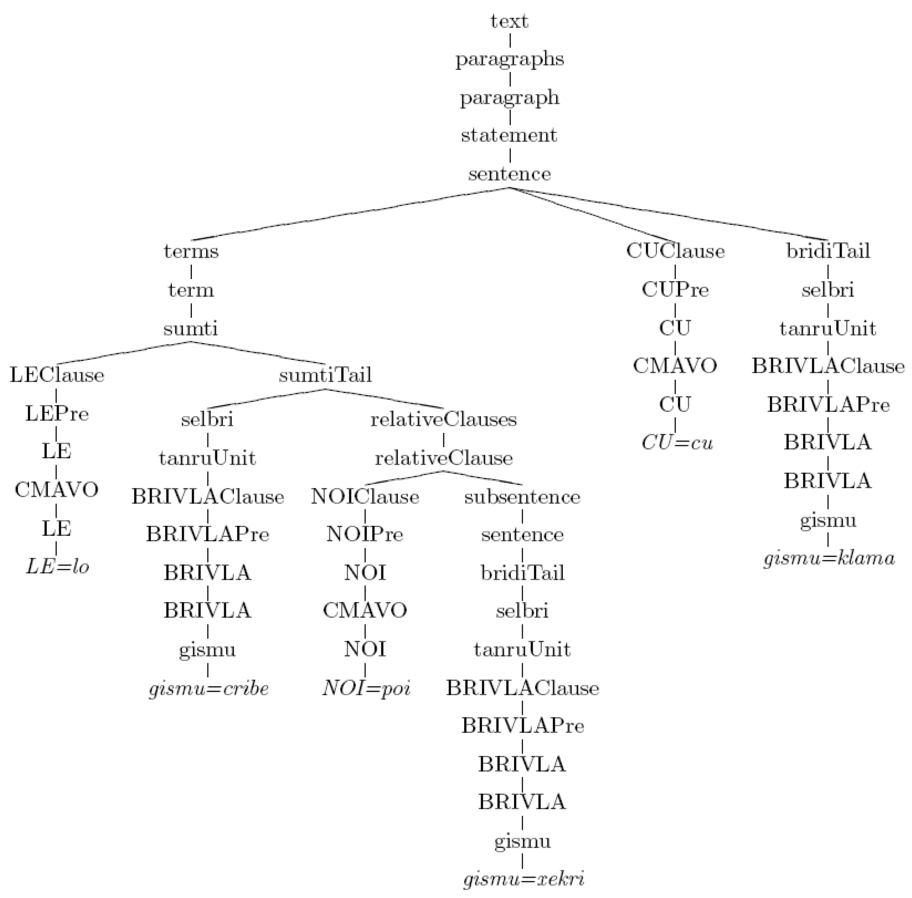


Figure 3.2: Sample Parse Tree

3.2.2 Structure Analysis Phase

The tree of tokens generated by the parser serves as the input for a series of analysis classes that examine branches of the trees according to their larger grammatical functions. Specifically, they are grouped into sentences which assert a single or logically connected group of facts, reference clauses which identify objects for *sumti*, relative clauses which modify references, and relation clauses which determine relations for *selbri*. The organization is still hierarchical: a sentence contains references and relations, references contain relative clauses, and relative clauses in turn contain sentences.

The structure produced by this analysis serves as an intermediate level between the raw grammatical constructs of the text and its logical import. Also, as this structure connects the logical meaning (what was said) to the actual language constructs that conveyed it (how it was said), this point makes available both halves of the information necessary to attempt translation between the Lojban text and other natural languages.

3.2.3 Meaning Resolution Phase

Once the analysis classes are constructed in memory, their overall meaning may be extracted through use of a `Resolver`. The `Resolver` serves as a factory for this operation, producing the necessary building blocks for the structure tree to represent its meaning. When complete, only the logical import of text remains: how the speaker conveyed that meaning with actual words is discarded.

The structure produced above is recursively visited by a `Resolver` to build up their meanings using *`Fact`*s, *`Relation`*s, and *`Subject`*s.

A *`Fact`* either aggregates two sub-*`Fact`*s and a logical connective or aggregates zero to five Subjects with their a *`Relation`*.

A *`Relation`* is a stand-in for the string which identifies a verb (generally a *gismu*).

A *`Subject`* serves as a variable which has possible values as defined by attached *`Fact`*s that contain query *`Subject`*s as produced by relative clauses. It may be either veridical (containing information about a new object) or nonveridical (implicitly making reference to one or more objects which are already known).

The `Resolver` itself follows the visitor and factory patterns. Each *Analysis* class accepts a resolver with its `resolve`() method and then uses the `Resolver` in different ways to construct its meaning.

The `Resolver` must be able to construct a...

- *`Fact`* from its *`Relation`* and *`Subject`*s
- *`Fact`* from two sub-*`Fact`*s and a logical connective
- *`Subject`* from a name
- *`Subject`* from *`Fact`*s with query-related *`Subject`*s (the relative clauses)
- *`Relation`* from a string

The *`Resolver`* also tracks and supplies *`Subject`*s for the query-related *prosumti*, including those used to ask questions (*ma* and *mo*) and those used for more advanced relative clause structures (*ke'a*). These *prosumti* behave in a stack-like manner and directly resolve from strings to Subjects independent of any neighboring text.

Through this process of resolution, the facts reach a form in which the reasoning engine may act upon them.

3.3 Reasoning

The reasoning phase of the system is responsible for taking in the data structures produced by the analysis phase, building a knowledge base, and performing logic against this knowledge base to respond to queries. With this in mind, the team decided to divide the reasoning phase into a number of parts: conversion of analysis data structures into reasoning structures, telling or querying the knowledge base, and finally describing the result, which is then sent off to the synthesis phase.

The following subsections outline how this particular reasoning implementation works. Future implementations need not exactly follow this outline; the highest level requirements of the reasoning phase are simply to take in data structures produced by the analysis phase and return answers to queries to the synthesis package when requested.

3.3.1 Reasoning Structures

The analysis phase provides Java interfaces for all the data structures it uses. The reasoning phase must therefore implement these interfaces, simultaneously providing the analysis phase a way to store its data and the reasoning phase a way to work with it. The data structures are for concepts such as subjects, facts, and relations.

The system is designed so the reasoning phase is modular, and can be substituted for another by implementation various interfaces specified by the analysis phase. In this way, the provided reasoner, Java Theorem Prover (JTP), can be replaced by another without the entire system being affected.

3.3.2 The Knowledge Base

The knowledge base stores the knowledge gathered through the course of a user's interaction with the system, and may be queried against to answer questions. The knowledge base and logic engine used in the project team's implementation is the theorem prover, JTP. It can use First Order Logic (FOL) to infer information from gathered knowledge and therefore answer queries.

3.3.3 Describing the Result

JTP will produce a string of subjects and how they are related when a query is run against it. Subjects are stored with unique, coded identifiers in JTP, not with their actual Lojban names. For example, 'Arthur' would not be stored, but instead, 's1' may be stored, with the relation that 's1' is named 'Arthur'. This method allows 's1' to also be associated with 'king', for example. Therefore, the reasoning phase must decide how to describe 's1' if it is used in the response to a query, and send that up to the synthesis phase so that it can reply to user with a sensical, and useful, Lojban statement.

3.4 Textual Synthesis

When an answer is returned from the reasoning engine, some work must be done to make it useful to the user. First, the description functionality of the reasoning package should be applied to any subjects which the system must refer to in its conversation with the user. Then, the answer Fact, original question Fact, and the generated description Subjects can be passed to the Synthesis package to generate corresponding Lojban text.

3.4.1 Answer Generation

First, Synthesis must turn a fact signifying a question into a fact forming an appropriate answer to that question. The Synthesis class provides the `createAnswerFact()` method for this purpose. It prunes the original question fact to only the subjects which were queried and the relations and connectives which attach them. It then replaces the query subjects in the pruned fact with subjects that contain attached descriptions, as supplied to it in a map.

3.4.2 Fact Synthesis

The answer fact must then be rendered into Lojban text using `Synthesis.synthesizeFact()`. For aesthetics, `synthesizeFact()` attempts to flatten the top-level nested facts connected with logical "and" into linked top-level sentences for the answer. Any remaining composite facts are then recursively turned into text as nested, logically-connected sentences.

At the leaves of the tree, `synthesizeFact()` delegates rendering of the subjects of simple sentences into text to its `synthesizeSubject()` function. It the assembles the resulting text into a simply formulated sentence and returns back to its parent composite fact to be inserted in the proper context.

3.4.3 Subject Synthesis

Subjects must be turned into *sumti* clauses by `synthesizeSubject()` in order to be of use to the user. The description process from the reasoning phase will provide a unique list of facts in which the subject participates with free variables in place of the subjects. However, simply outputting these facts as relative clauses is, though technically correct, difficult to read and quite unnatural. Necessarily, all output *sumti* will be either by name or nonveridical description, as the reasoning engine will only refer to already discussed objects rather than invent new objects on its own.

First, an approach must be chosen to appropriately synthesize the subject. If one of the describing facts returned by the reasoner included a name (for example "*zo .artur. cmeme X*", or "*the string 'artur' is the name of this*"), the name can be used as the base reference and any additional facts appended as restricting clauses. If there is no name but one of the describing facts has the subject in its primary place with no additional subjects, the *selbri* from that describing fact can be used as the base reference and the other facts, once again, appended as restricting clauses. If neither of these cases occurs, then any *selbri* from a fact with the subject in the first place may be used as a base reference, but then all facts (including that which supplied the base reference) must be attached as relative clauses. Finally, if the subject never appears in the primary place, Synthesis may use a generic base reference such as "*le sidbo*" or "*the thing*" and, once again, apply all the descriptions as relative clauses.

Each relative clause must then be synthesized and appended to the base reference. If the subject appears in the first place, the first subject place can be omitted and the *selbri* and remainder of the *sumti* appended. If the subject appears later, then *ke'a* must be used to mark the place where the described subject would fall.

The completed subject string is then returned for inclusion in its appropriate context.

3.5 Speech Synthesis

To emphasize that Lojban is a spoken language meant for human interaction, the project includes a system to pronounce Lojban utterances via speech synthesis. To realize this goal, the project team had to select a speech synthesizer, create a new voice or modifying an existing voice, and plug the new voice and synthesis program in to the system.

3.5.1 Selecting a Speech Synthesizer

When initially investigating possible text to speech solutions a list of requirements was created for the speech synthesis system. Any text to speech solution must (1) be designed to handle languages other than English, (2) have a well-documented interface, (3) be able to drop into the Java interface easily, and (4) be open-source with a license which is compatible with the goal of releasing this project as open-source when completed.

Festival was the first text to speech synthesizer that the team evaluated. Festival is a text to speech framework which is fairly well-known for it's flexibility with languages. The US English voice included with its distribution package is very advanced and produces excellent output. The Festival website touts its ability to handle multiple languages, so the first requirement is satisfied. The second requirement is less clearly met. Festival's User's Manual, *The Festival Speech Synthesis System* ([3]), does an excellent job of explaining parts of a voice that need to be constructed, but the process to actually create the voice in code is not clearly documented. There is no real API documentation, and most of the voice code is written in Scheme. The learning curve for implementing and interfacing a voice in Festival would therefore be very steep. With the third requirement, Festival completely fails because the library functions for the system are only available in Scheme and C++. Using this system limits the portability of the final product because C++ is not cross-platform in the same way Java is, and therefore it would be difficult to link the native C++ library into the Java application. The requirement for a compatible open-source license was met, but the lack of API documentation and incompatibility with Java were sufficient to make this solution undesirable.

The second speech synthesis system considered was FreeTTS. Through research of Festival, the team found several references to this system and decided to investigate it further. FreeTTS is a Java-based system derived from Flite (originating from "Festival Lite"). Although Flite was originally written in C, FreeTTS is written natively in Java and implements portions of the Java Speech API (Java Speech API (JSAPI)), making it compliant with Java standards. FreeTTS is designed so that new voices can be created for any language and it is possible to convert Festival voices to FreeTTS ones, though doing so is "not trivial, and it requires using Festival and FestVox." [1] Since FreeTTS is implemented in Java, plugging in a FreeTTS-based synthesizer would be relatively easy. It would remain cross-platform and not require any kind of cross-language bindings. The license for FreeTTS is also acceptable. Although it is not a standard open-source license, it gives the right to modify, sell, license, and alter the product so long as copyright notices and author information is not removed and changes from the original code are marked. This system presents an acceptable solution for a speech synthesizer for a Lojban voice and was subsequently selected for creating a voice to speak Lojban.

3.5.2 Creating a Voice

`LojbanVoice` is the class that defines the actual FreeTTS Lojban voice. It extends an existing English voice which speaks using pre-recorded English diphones. The original voice was designed to speak English utterances with good pronunciation, pauses, stresses, durations, and other language features as you would expect them in spoken English using a series of processors to modify the output. The Lojban voice simply re-implemented some of these features so that they comply with Lojban grammar and pronunciation. The following sections will describe how this voice differs from the original English version.

Tokenization

We began our voice modifications by looking at the English tokenizer implementation. The English class performs the tokenization task (separating each token in the utterance) and token to word conversion for English, but the English rules were unnecessarily complex for Lojban. Although Lojban words are still separated by spaces, there are no cases where one token might map to more than one word. Therefore, much of the processing done by the English tokenizer is not necessary in Lojban tokenizer.

Phrase Breaks and Pauses

The Lojban tokenizer uses the Rats!-generated parser to mark the end of a sentence. This method takes advantage of the already-existing parser, thereby preventing the need for another algorithm to determine if the next token begins a new phrase. After the string has been marked for phrase ends, the tokenizer can easily find where phrase breaks occur in the string.

Pauses in the Lojban voice are done via `if` statements which check to see if pausing conditions are met. Currently two conditions exist to check if the current word begin with a vowel or end with a consonant. Other rules are ignored because the syllabification and part of speech information which is required to check for the conditions is not in place.

Part of Speech Tagging

Unlike English and many other spoken languages, Lojban word pronunciations do not change based on part of speech. This initially seemed to make part of speech tagging like in English superfluous, but full implementation of the pausing rules requires knowledge of part of speech. No part of speech tagging has been implemented for the Lojban voice, though the Rats!-based parser could be used to determine part of speech in the future.

Lexicon

The Lojban voice and the English class it inherits from differ primarily in their use of the lexicon. The English voice uses an extensive list of pronunciations for known words, which could be specific to the intended part of speech, to generate more accurate pronunciations. For unknown words, it uses a statistical analysis of the word to best guess how it should be pronounced.

In Lojban, each phoneme and diphthong map to specific phones. Using the phone set from the English voice, the team matched each Lojban phoneme and diphthong to the appropriate existing English phones. The real work here was in figuring out what each of the pre-existing phones actually sounded like: marked with cryptic representations such as “ah”, “eh”, “ey”, and “ey”, the sound represented was sometimes difficult to determine. With forty phones to work with, experimentation was needed to obtain a good result. diphthongs were even more difficult because their pronunciation had to be determined based on examples from other languages whose pronunciation is also subjective. For example, the diphthong “ia” sounds like the German Ja, ultimately best formed with “iy” and “aw”. Several iterations were needed to find all the correct phones and get all the pronunciation bugs worked out.

Chapter 4

Implementation

This section describes the actual Java implementation of the project, including details on package and class organization, the responsibilities of both, and other architecture decisions.

4.1 Textual Analysis

The process of Textual Analysis is contained within the *Analysis* package. The `Analysis` facade class is the only public class exposed by the package. A calling class performs the analysis process in programmatic two stages: the first constructs an internal structure describing the meaning of the text, while the second extracts that meaning to another form.

The first stage wraps up the logical phases, described below, of parsing the string into a tree of generic nodes and then building the internal structured analysis tree. It is initiated by providing a string to `Analysis` in its constructor or by delivering it through the `analysis`() method. Subsequent strings in the same conversation should share an `Analysis` class to maintain state. This is vital for determining *prosumti* and *probridi* antecedents.

The second stage is initiated by providing a `Resolver` to facade's `resolve`() method. This stage ultimately extracts the meaning of the string as objects understandable by the reasoner. Alternately, a `Dictionary` class may be passed to the `toTranslation`() method of `Analysis` to instead extract meaning as approximate English or Spanish text.

4.1.1 Parse Phase

Upon receiving a string, the `Analysis` facade delegates parsing to the *Parser* package. There, the Rats! parser generator is invoked upon the input string. The parser returns a resulting string similar to that quoted below, corresponding to the phrase "*lo cribe poi xekri cu klama*" diagrammed in section 4.2.1. That result string is then parsed into an n-tree of `GenericParseNode`s and returned to the calling method in `Analysis`.

> text[text1[paragraphs[paragraph[statement[statement1[statement2[statement3[sentence[terms[terms1[terms2[term[term1[sumti[sumti1[sumti2[sumti3[sumti4[sumti5[sumti6[LEClause[LEPre[LE[CMAVO[LE=**lo**]]]] sumtiTail[sumtiTail1[selbri[selbri1[selbri2[selbri3[selbri4[selbri5[selbri6[tanruUnit[tanruUnit1[tanruUnit2[BRIVLAClause[BRIVLAPre[BRIVLA[BRIVLA[gismu=**cribe**]]]]]]]]]]]]]] relativeClauses[relativeClause[relativeClause1[NOIClause[NOIPre[NOI[CMAVO[NOI=**poi**]]]] subsentence[sentence[bridiTail[bridiTail1[bridiTail2[bridiTail3[selbri[selbri1[selbri2[selbri3[selbri4[selbri5[selbri6[tanruUnit[tanruUnit1[tanruUnit2[BRIVLAClause[BRIVLAPre[BRIVLA[BRIVLA[gismu=**xekri**] CUClause[CUPre[CU[CMAVO[CU=**cu**]]]] bridiTail[bridiTail1[bridiTail2[bridiTail3[selbri[selbri1[selbri2[selbri3[selbri4[selbri5[selbri6[tanruUnit[tanruUnit1[tanruUnit2[BRIVLAClause[BRIVLAPre[BRIVLA[BRIVLA[gismu=**klama**]

4.1.2 Structure Analysis Implementation

The `Analysis` facade then delivers the topmost node in the parse tree to the `TextAnalysis` class. The `TextAnalysis` expands the uppermost levels of the tree corresponding to utterances and paragraphs, to find sentence nodes. It then creates and delegates each found sentence node to an instance of `SentenceAnalysis`. A `SentenceAnalysis` then creates and delegates to a single `RelationAnalysis` for its *selbri* and some number of `ReferenceAnalysis` classes for its *sumti*. Each `ReferenceAnalysis` may in turn create and delegate to `RelativeClauseAnalysis` classes, which in turn can create and delegate to a `SentenceAnalysis` class, recursing as needed by the complexity of the text.

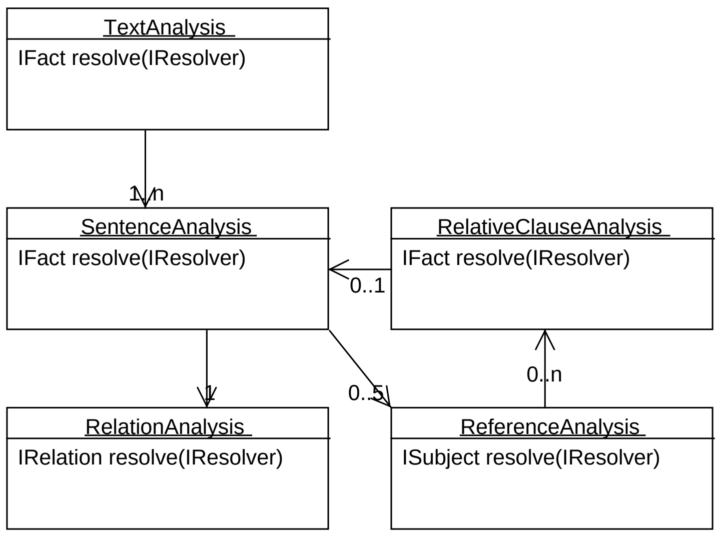


Figure 4.1: Analysis Package, General Organization

For faster implementation, an `AbstractRecursiveAnalysis` class was built as the parent to all of the *Analysis* classes. This abstract parent contains utility functions to handle a node's children and to delegate handling of a given node to a void member function named `processX()` where X is the node's token type. The former is useful for rapidly coding default handling of nodes which are themselves inconsequential but have useful children. The latter, by using reflection to perform the delegation, allows for runtime detection and smart reporting of grammatical constructs which have not yet been handled. Together, these functions made incremental implementation of language features much more manageable. In some final complete implementation, the reflection mechanism would presumably be replaced with a switch upon the node types.

The result of this processing phase is an in-memory structure of analysis classes which links the collections of words to their function in the text. This structure is then visited in the next phase to produce the meaning of the entire text, or alternatively may be recursively examined using a dictionary object to produce an approximate translation of the text into another language, as is done to produce the English and Spanish translations in the demo application.

TextAnalysis

The `TextAnalysis` class is primarily responsible for delegating handling of the sentence nodes it receives to `SentenceAnalysis` classes. `SentenceAnalysis` also handles logical connectives (particularly structure words signifying uncertainty and implying a true/false style query) placed between sentences. When resolved in the next phase, it will connect the resulting sentence meanings together with the appropriate logical functions.

Additionally, `TextAnalysis` records and supplies the implicit and explicit assignments of *prosumti* and *probridi* words (similar to pronouns in English) which occur in the sentences it contains.

SentenceAnalysis

A `SentenceAnalysis` operates in one of two guises. If it contains connected clauses, it will aggregate the `SentenceAnalysis` classes which process those subclauses and supply logical connectives upon resolution. If, instead, it contains only a single clause (*bridi*), it will break the clause down into its verb and objects (*selbri* and *sumti*) and delegate to `RelationAnalysis` and `ReferenceAnalysis`.

`SentenceAnalysis` also takes responsibility for identifying when its components are indirect references (*prosumti* or *probridi*) rather than direct references. In these cases, the `SentenceAnalysis` performs the lookup with `TextAnalysis` and replaces the indirect references with direct references to the antecedents.

RelationAnalysis

As the project aimed only to capture simple root word *selbri* rather than the more complex metaphor-like *tanru*, the implementation of `RelationAnalysis` proves extremely straightforward. `RelationAnalysis` must only save the unique string which identifies its verb and determine if the verb has been negated. Alternatively, in cases where a *probridi* is used in place of an actual relation, `RelationAnalysis` must detect this and provide the information necessary to `SentenceAnalysis` to find its antecedent.

ReferenceAnalysis

`ReferenceAnalysis` must take a named, veridical, or nonveridical descriptor, a *sumti*, and turn it into a logical structure representing the object being identified. Alternatively, if a *prosumti* was used, the `ReferenceAnalysis` must provide information to identify its antecedent.

A *le* or *lo* article present will indicate the presence of a veridical or nonveridical descriptor, for example "*lo cribe poi xekri*". In this case, the base relation, "*cribe*", and any relative clauses, "*poi xekri*", must be delegated to `RelationAnalysis` and `RelativeClauseAnalysis`. During the resolution phase, they will be attached together to specify a subject. Here, the subject would be marked as veridical (because of the *lo*) and specify "*cribe(x) and xekri(x)*", or "*x is a bear and x is black*".

RelativeClauseAnalysis

`RelativeClauseAnalysis` serves as a wrapper around a `SentenceAnalysis`, essentially replacing an argument in the `SentenceAnalysis` with a variable. In the example above, the phrase "*poi xekri*" would have been delegated to a `RelativeClauseAnalysis` object. `RelativeClauseAnalysis` delegates processing of the contained sentence "*xekri*" to `SentenceAnalysis`. When visited in the next phase, it will modify the meaning of the result `SentenceAnalysis` returns from "*something is black*" to "*X is black, where X is the referent of the parent reference*".

`RelativeClauseAnalysis` must also describe the function of its phrase in the text. In the above example, *poi* signifies that the clause is restrictive: it limits the objects to which the delegating reference may apply. If *noi* had been used instead, the clause becomes incidental: rather than restrict what objects the reference might indicate, it provides additional information about the object. For example, "*le cribe poi xekri*" would mean "*the bear which is known to be black*", while "*le cribe noi xekri*" would mean "*the bear which, incidentally, the speaker asserts is black.*" These will produce differently structured results during the resolution phase.

4.1.3 Meaning Resolution Implementation

At this point, `Analysis` contains the tree which organizes the string according to the meaning of its parts and the constructor or the `analyze`() call has returned. However, no information has yet been passed back. To extract meaning from the `Analysis`, a *`Resolver`* must be allowed to visit the nodes of the tree via their `resolve` methods.

`Resolver` and the *`Fact`*s, *`Relation`*s, and *`Subject`*s it provides, as described in section 4.2.3, are defined as interfaces in the *Analysis* package. They must be implemented by any knowledge storage/reasoner system to be used with the *Analysis* classes. The actual implementation of these classes is discussed below with the reasoning engine.

`Resolver` exposes to nodes it visits the methods needed for construction of a tree describing their logical meaning in terms the reasoner can understand. Specifically, it provides:

- `createFact` and `createConnectiveFact` to build and attach sentences;
- `getRelationByName`, `getSubjectByName`, and `getSubjectByRestrictions` to form the component pieces of sentences, *`Relation`*s and *`Subject`*s;
- `getRelativeSubject`, `getQuerySubject`, and `getLiteralSubject` to provide unique placeholder *`Subject`*s for parts of sentences;
- and `pushAssignment`, `popAssignment`, and `getAssignment` to facilitate management of stack-like *prosumti*, such as the relative clause placeholder *ke'a*.

4.2 Reasoning

The analysis and resolution process results in a fact structure suitable for processing during the reasoning phase of the system. *`Fact`*s, *`Subject`*s, and *`Relation`*s must then be converted into the reasoning engine's syntax. The engine then performs first order logic reasoning on its knowledge base, and returns an error if there is an inconsistency, an acceptable result if everything is consistent, or a series of possible answers in the case of a query. The reasoning phase then takes this result, and transforms it into something useful for the user. In the case of an asserted statement, it simply bubbles up the success or failure of the assertion. In the case of a query, it passes on the collection of possible answers to the "describe" algorithm which figures out how best to give the user the answer to the question they posed. Finally, this described answer is passed up to the synthesis phase.

4.2.1 The Data Structures

Analysis and *Resolution* are responsible for the conversion of Lojban into data structures. The end result of this process is a `JTPFact`. There are two classes which inherit from `JTPFact`, `JTPSimpleFact` and `JTPConnectiveFact`. `JTPSimpleFact` expresses one simple Lojban *bridi*, so it is composed of a `JTPRelation` (verb-like words, Lojban *selbri*), up to five `JTPSubject` type objects (x0 through x4, Lojban *sumti*), and a truth property, which indicates whether this fact is known to be true or false. A `JTPConnectiveFact` contains two `JTPFact` objects, a connective (or, and, if and only if, whether or not), and a truth value for each `JTPFact` stating whether that fact is negated in the logical connection. Any Lojban *bridi* can be expressed in this `JTPFact` data structure.

`JTPSubject` is an abstract class with three implementing classes: `JTPConcreteSubject`, `JTPLateSubject`, and `JTPLiteral`. `JTPLateSubject` expresses a nonveridical subject, referring to a subject already known. The describe algorithm returns this type of subject. `JTPConcreteSubject` expresses a veridical subject, one new to the reasoning engine which must be defined. `JTPLiteral` simply indicates a literal, or string. For example, in the statement "*Arthur is-the-name-of a king,*" "*a king*" is veridical and therefore would become a `JTPLateSubject` and "*Arthur*" is a `JTPLiteral`. On the other hand, should the sentences be rephrased to be "*Arthur is-the-name-of the king*", then "*the king*" is nonveridical and would therefore be a `JTPConcreteSubject`. `JTPRelation` expresses a relation. In the previous example, "*is-the-name-of*" is the relation.

4.2.2 The Reasoning Engine

Java Theorem Prover (Java Theorem Prover (JTP)) was chosen to serve as the reasoning engine for the project because it is written in Java which allows for easy integration, it is well documented, it supports positive and negative assertions, and it is open source which allows future programmers to expand its capabilities. For more information about JTP, see section 2.3.2.

JTP's syntax is intentionally similar to that of Prolog because many programmers are accustomed to that language. ("*loves Arthur Guen*") states that there exists a relation "*loves*" where "*Arthur*" is in position one, and "*Guen*" is in position two. To query against that fact, simply put a question mark in the position that one desires to have filled in. For example, ("*loves Arthur ?x*") would return ("*?x = Guen*"). Multiple question mark variables may be used, and the engine will return all possible values for each variable. However, ("*?x Arthur Guen*") is illegal in JTP syntax, as it is not First Order Logic (FOL) (that syntax requires a higher order logic). To get around this limitation, statements and queries were modified to use always use the relation "fact" and then insert the fact. Therefore, ("*loves Arthur Guen*") becomes ("*fact loves Arthur Guen*"), and now ("*fact ?x Arthur Guen*") can be asked. Although this change may have some unanticipated negative effects, the project team never encountered any problems through their testing with low- to mid-complexity queries.

The project team also encountered another issue in the form of the halting problem. FOL theorem proving is not guaranteed to halt, so the project team expected some cases to come up where the engine would continue searching forever. Initial tests resulted in looping for unacceptably simple queries, however rephrasing the statements used to prime the reasoner to avoid recursive loops eliminated most simple problems. Due to the nature of this inherent FOL theorem prover problem, however, sufficiently complex queries may still cause processing to fail to halt.

`JTPFact` implementations include methods allowing the `JTPFact` to convert itself to JTP query language. For each contained subject, there are two major cases: nonveridical (late) and veridical (concrete) subjects. In the first case, the query is formed with the references to the object on the left of an implication and then the fact on the right. A “?qn” is used as a reference to the object. The “?” indicates that this reference is a variable which will be filled by previously (or future) defined subjects. For example, take the sentence “*the table is white.*” “*The table*” is a nonveridical, which becomes a concrete subject. “*Is white*” is the relation. Therefore, the JTP statement is “(=¿ (fact table ?q1) (fact white ?q1))”. Note that, for purposes of readability, the example is in English. In the actual project, all data is handled in Lojban.

For veridical (concrete) subjects the query is formed with the conjunction of the fact involving the subject and the overall fact. Instead of a “?qn” reference, an “sn” reference must be used because, as in the case of veridical subjects, an actually object is being defined. Consider the example, “*a table is white.*” “*A table*” is veridical, so it becomes a concrete subject. Therefore, the JTP statement is “(and (fact table s1) (fact white s1))”.

An example of a fact with both a veridical and nonveridical subject is the English sentence, “*A dog goes to the hydrant.*” In JTP, this sentence is expressed as “(and (=¿ (fact dog ?q2) (fact goes ?q2 s2)) (fact hydrant s2))”.

Note that all ?q numbers and s numbers must be unique outside of their JTP statements. Reuse of “?qn” or “sn” references outside of their original JTP statements can result in JTP not evaluating the knowledge base as the end user would expect. Therefore, a static counter is kept and used to provide unique numbers for identifiers.

4.2.3 The Describe Algorithm

The describe algorithm takes a `JTPLiteral` which contains the s number of a subject to describe, and returns a `JTPConcreteSubject` which is the description of the specified subject. To reach this goal, it gathers all facts involving the subject, tests to determine which facts are unique, sorts the list of unique facts according to their desirability, and finally builds and returns a `JTPConcreteSubject`.

The first step is to gather all the facts that involve the subject. Two JTP queries are run: one for “relates”, and one for “relatesnot”. These queries find relations known to be true or false which contain the subject. “relates” is defined as “(=¿ (fact ?r ?x1 ?x2 ?x3 ?x4 ?x5) (and (relates ?r ?x1 x1) (relates ?r ?x2 x2) (relates ?r ?x3 x3) (relates ?r ?x4 x4) (relates ?r ?x5 x5)))”. Similarly, “relatesnot” is defined as “(=¿ (factnot ?r ?x1 ?x2 ?x3 ?x4 ?x5) (and (relatesnot ?r ?x1 x1) (relatesnot ?r ?x2 x2) (relatesnot ?r ?x3 x3) (relatesnot ?r ?x4 x4) (relatesnot ?r ?x5 x5)))”. The results of both of these queries are turned into `JTPFact` objects and put into an list.

The list must now be searched to remove all facts which are not unique. In order to perform this filtering, each fact is examined individually. The ‘s’ reference to the subject being described is replaced with a query subject, then this fact, now a query, is run against JTP. If the fact is unique, JTP will return exactly one answer, where the query subject has the value of the subject being described. In any other case, the fact is ambiguous, and the describe algorithm cannot provide a valid description, and therefore returns an error.

The list of unique facts is now sorted so the most useful description is returned. The project team came up with a heuristic involving the number of subjects in a fact and their types to gauge how useful each fact is relative to the others. This heuristic was mostly formed by guess and check, and now provides more useful descriptions in every case the team has requested of it.

The highest ranking unique fact is then returned. This fact is taken from the top of the list, and the `JTPConcreteSubject` constructor is called with this fact and the subject being described as its parameters, associating a description with the “sn” literal.

4.3 Textual Synthesis

4.4 Speech Synthesis

The actual `LojbanVoice` class extends `CMUDiphoneVoice` and overrides the `getTokenizer`() method. The `LojbanLexicon` lexicon is specified as the default in `LojbanVoice`'s `VoiceDirectory`, which creates the voice at runtime. Some of the implementation details for `LobanVoice` are discussed in more detail in this section.

4.4.1 Tokenizer

`LojbanTokenizer` implements the *`Tokenizer`* interface provided by FreeTTS. The `setInputText`() method accepts the string to be tokenized. When this method is called, a private string array is filled with all the words from the input string separated at whitespace characters. A position tracking integer is used so that the `getNextToken`() method knows which token to return next.

4.4.2 Pause Generator

In `LojbanTokenizer`, the `setInputText`() method When `setInput`() is called it, it first passes the incoming string to a marking function which uses Rats! to add an asterisk after each sentence. It then splits the string up into an array. The `isBreak`() method checks to see if the next token in the string array is an asterisk and returns true if it is, indicating a new sentence follows. This public method is available for other classes to know when phrase breaks occur; however, the `LojbanTokenizer` does not use it because there is no current need for this knowledge in the tokenizer.

In `getNextToken`(), after the `Token` object has been created, conditions are checked to determine if a preceding or trailing pause is required. The `Token` class contains data members for pre- and post-punctuation, which are set to the 'pau' phone (which is silent) when a pause is required in either position.

4.4.3 Lexicon

The `LobanLexicon` implements the *`Lexicon`* interface. It's primary method, `getPhones`(), takes two arguments: a word and a part of speech. The part of speech is ignored since it does not affect pronunciation. The word, taken as a series of characters representing a phonemes, is turned into a series of phones. For each phoneme, a simple analysis is done to determine what phone it will map to. First, the phoneme is checked for validity. Invalid phonemes produce runtime exceptions for easy debugging, so invalid letters are not allowed. Next, the current and next phoneme are checked to see if they are both vowels. If so, then they are matched against the diphthong list, and invalid diphthongs throw runtime exceptions. Other consonants and vowels are mapped directly to a phone. All the phones for the word are returned in an array of strings.

Inherited Processors

All other *`UtteranceProcessor`*s from `CMUDiphoneVoice` are inherited. The `LojbanVoice` inherits the syllable segmenter from `CMUDiphoneVoice` which does a good job identifying the highly regular Lojban syllables. The intonation, duration, pitch, and a few other features are also unchanged from English. The overall effect on the synthesized speech seems positive. The sound output is quite good and the intonation and syllable stresses all seem appropriate.

Registering the Voice

FreeTTS requires a *`VoiceDirectory`* which describes information about the voice to the `VoiceManager`. The `VoiceManager` reads the list of available *`VoiceDirectorie`*s from a voices.txt file, which is expected to be in the same directory as the FreeTTS source. Creating a *`VoiceDirectory`* makes the voice available to the system. Once this is completed and the voice is fully implemented, the user only needs to initialize the `VoiceManager` and ask it to provide an instance of the desired *`Voice`*.

Chapter 5

Results and Conclusions

5.1 Language

The final application implements only a subset of the complete Lojban language, including the fundamentals and those aspects most powerful for addressing an engine for solving problems of First Order Logic (FOL). The specific features are identified in the table below.

- *bridi*
  - simple *bridi*
  - *sumti* rearrangement by *fa* series
  - *bridi* linked by *.ijeks*
  - compound *bridi* with *geks*
- *selbri*
  - *gismu* as *selbri*
  - *cei* assignment clauses

- negation through *na*
- true/false question through *xu*

- *sumti*
  - *lo* and *le* for veridical and nonveridical description
  - *la* for *cmene* description
  - *zo*, *zoi* and *lo'u* literal quotes
  - *li* number clauses captured but unprocessed
- *sumti* relative clauses
  - *goi* assignment clauses
  - *noi* incidental clauses
  - *poi* restrictive causes
  - *pe* and *po* for possessive clauses
  - *zi'e* for compound relative clauses
  - *ke'a* for complex relative clauses
- *probridi*
  - *go'i*
  - *broda* series
  - *sumti* overriding
- *prosumti*
  - *ko'a* and *fo'a* assignable series
  - *vo'a* reflexive series

- *ri* anaphoric references
- *ma* and *mo* question *prosumti*
- personal *prosumti* (*mi*, *do*, *ko*)

Because the grammar is rigorously defined, the tree-like approach used for adding further features to the language analysis phase proved remarkably easy. These features were added in an incremental approach as the reasoning engine grew to support them. The code base is thus hopefully conducive to future expansion.

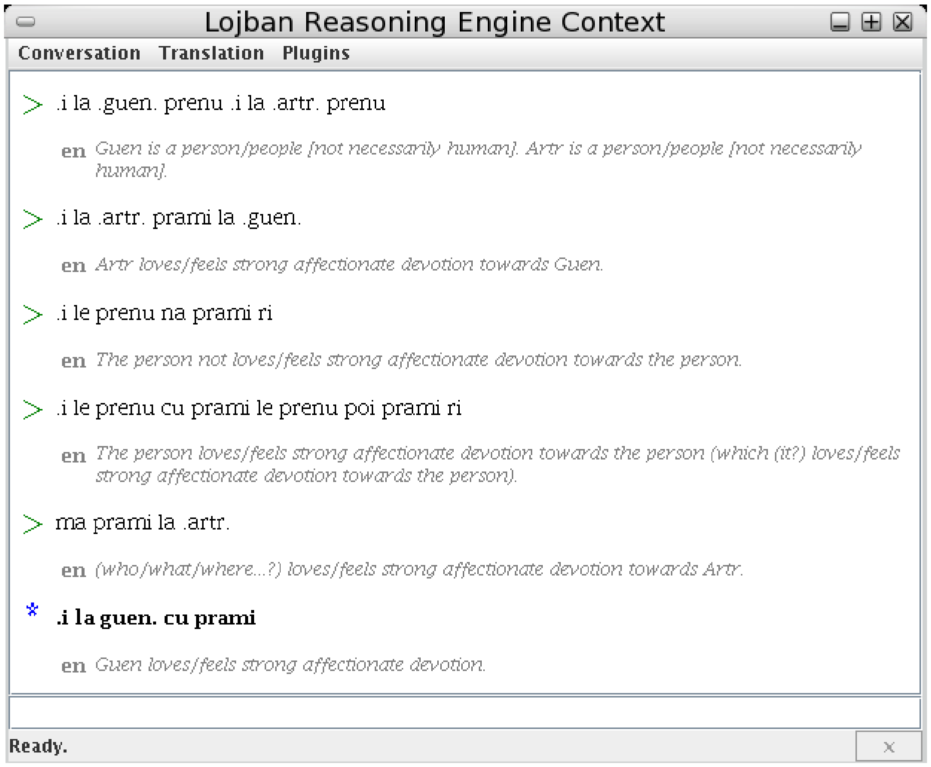
Some notably important language features were not implemented because they are not constrained to first-order logic. Any metalinguistic features that would allow a sentence to become an object for another sentence were omitted. Also, numeric features including mathematical expressions and numeric quantifiers were not implemented.

5.2 Logic and Reasoning

The logic and reasoning section of the project team's implementation of a Lojban Natural Language Interface successfully met the team's goals. The project is able to accept Lojban statements and assert them to be true or false to its knowledge base. Questions may be posed to the knowledge base, in Lojban, and a result is formulated and returned also in Lojban.

There are a few features that would have been desirable but which were impossible or infeasible to implement. First, as outlined in the halting problem, the system cannot guarantee a reply to all queries. As testing the validity of a FOL sentence is not a decidable problem, there are Lojban sentences for which, when translated to queries, the theorem prover will not return. As this behavior is fundamentally unavoidable, the team added an "X" button to the user interface to abort a query by prematurely halting the logic engine if the user chooses not to wait.

Additionally, temporal change is not handled. The user may not state contrary facts, for



> .i la .guen. prenu .i la .artr. prenu

en *Guen is a person/people [not necessarily human]. Artr is a person/people [not necessarily human].*

> .i la .artr. prami la .guen.

en *Artr loves/feels strong affectionate devotion towards Guen.*

> .i le prenu na prami ri

en *The person not loves/feels strong affectionate devotion towards the person.*

> .i le prenu cu prami le prenu poi prami ri

en *The person loves/feels strong affectionate devotion towards the person (which (it?) loves/feels strong affectionate devotion towards the person).*

> ma prami la .artr.

en *(who/what/where...?) loves/feels strong affectionate devotion towards Artr.*

* **.i la guen. cu prami**

en *Guen loves/feels strong affectionate devotion.*

Ready.

Figure 5.1: The Final User Interface

example, and expect the later one to be true, effectively overriding the initial assertion. This limitation is a feature of Java Theorem Prover (JTP). Other theorem provers do exist that can work with temporal logics and they may be used in place of JTP in future versions of this project.

Finally, the team did not implement commands. The user can not tell the computer to do something by finding what it needs to do to make the desired change and using an external interface to actually do it. This ability would be extremely useful. For example, one could tell the computer to lower the temperature, then the computer could know, through a previously expressed assertion, that an open window lowers the temperature, so the system would then open a window. The use of temporal logics, as mentioned above, would facilitate such an extension.

5.3 User Interface

The Graphical User Interface (GUI) that the team designed is clean, simple, and easy to use. The program consists of a line for the user to type Lojban in, a box above that for the program to respond, and a series of drop down menus in the customary menu bar location. Also, to the right of the input line, there is an "X" button, that, when pushed, will kill the reasoner if the user feels that it has been running for too long. To use the program, the user enters a Lojban statement on the input line. He will see the soon see the statement echoed back in the response box, followed by any translations he has enabled in the drop down menus. Also, if the user has enabled speech synthesis, he will hear his statement spoken aloud. Next, if the statement was question (versus an assertion) the program will determine the answers. If the user selected the option to allow multiple answers in the drop down menus, the program will display all answers, sorted by predicted order of usefulness (determined by heuristics) in the response box. Otherwise, it will display only the answer that the heuristics determined was most useful in the result box. In both cases, all answers will be spoken if that option is enabled, and all selected translations will be displayed if any have been chosen following that answer. At this point, the process starts again.

5.4 Text to Speech

5.4.1 Phone Mapping

The method used to map phonemes to phones in the new Lojban voice is much simpler than in the original English voice. In the English voice, a large pronunciation lexicon is used to quickly look up a word's pronunciation for several thousand known words. For the rest, an algorithm was applied to "best guess" the pronunciation. Since Lojban pronunciation rules are much simpler than English, this large lexicon is unnecessary and all words can be easily mapped from their spelling.

One issue that the team did encounter when creating the phone map system was the difficulty in pronouncing the Lojban letter ‘x’. To determine what the letter sounds like we looked for examples. *Lojban for Beginners* ([12]) says ‘x’ sounds like “*German Ba**ch**, Spanish **J**ose or Arabic **Kh**aled*”. The reason this was difficult for us to reproduce became apparent when we reached page 31 in [8]: “the sound of Lojban ‘x’ doesn’t appear in most English dialects at all.” After reading this we decided to settle for a similar English phone, ‘k’. Although this works for an English-speaker’s pronunciation of “*Bach*” or “*Khaled*”, it is completely wrong for “*Jose*”. This solution is good enough for this project, but a full implementation would need this phone to be available in the voice in order to be considered complete and correct.

5.4.2 Pronunciation

Lojban uses a subset of the the English phone set (with one exception) included with `CMUDiphoneVoice` to synthesize speech. As all but one Lojban phones exist in normal English, this method proves acceptable. The actual pronunciation in the Lojban voice is handled by mapping individual letters and diphthongs directly to phones. The mapping was done manually by trial-and-error, eventually resulting in accurate synthesis. Although the International Phonetic Alphabet (IPA) glyphs are available in [8], the phones provided in the English voice did not have their matching symbols available. As a result, the matching was done based on demonstrated pronunciation from a English words found in [12].

5.4.3 Pausing

Pausing in the Lojban voice is performed in the `Tokenizer` class. Although it could be performed by a separate *`UtteranceProcessor`*, our choice for where to implement it was for simplicity: since we only implement a subset of the pausing rules (see section 2.1.2 for the full rules), the pausing decision could be implemented in only two if statements. The resulting pauses are completely accurate in placement, but full pause implementation would be better suited for an `UtteranceProcessor`.

5.4.4 Other Potential `UtteranceProcessor`s

There are several other pronunciation features that might have been implemented for the Lojban voice that were omitted for a variety of reasons. Two of the more noticeable ones are discussed in this section.

Syllables and Stresses

Because Lojban is so highly structured it is easy to separate individual syllables and predict stresses. In Lojban, stresses occur on the penultimate syllable (the next-to-last), not counting syllables which contain 'y', 'l', 'm', 'n', or 'r'. [8] Since the rules are much simpler than English they would be simple to implement in an `UtteranceProcessor` interface. This feature was omitted from the Lojban voice because the inherited processor from the English voice did a sufficient job of adding stress to words. Reimplementing this processor could make the stress placement more accurate and, depending on the English implementation, increase performance.

Part of Speech

There are three primary parts of speech in Lojban, *cmavo*, *brivla*, and *cmene*. It should be easy to perform pattern matching on words to determine their part of speech, but this feature was not implemented because it would not have assisted in the speech synthesis as it did with the English voice. However, full implementation of pausing rules does require part of speech information in order to make decisions, so implementing this feature is necessary for that goal.

Chapter 6

Future Work

In this section, areas of potential future research will be described, as well as possible applications of the concepts and discoveries explored during this project.

6.1 Language

Both the analysis and synthesis stages could be refined for more powerful and more natural communication with the user.

6.1.1 Additional Analysis Features

Many language features were omitted from this sample implementation that could prove interesting for future development. The powerful structures for mathematical notation could be explored with a mathematically-equipped reasoner. The *probridi* and nesting constructs which allow for metalinguistic expressions might be implemented for experimentation with higher-level logics. The implications of using *tanru* constructs for a large range of vocabulary in sentence construction also warrants further exploration, though it does not yield any additional power from the point of view of logical expression.

6.1.2 Enhanced Synthesis

Future systems will want to extend the synthesis phase to generate more fluid prose. An understanding of when terminators may be elided and when they are required would lead to less verbose output.

Additionally, this system has the limitation that any object to which it refers during synthesis must be describable with a single positive or negative fact. A more powerful system would be able to identify objects which are not atomically identified by a fact but which are uniquely indicated by the intersections of some facts.

6.2 Reasoning

The reasoning engine used in this project could be replaced or enhanced to make the Lojban natural language system far more useful and powerful.

6.2.1 Tense

The present logic engine, Java Theorem Prover (JTP), cannot be used in a system which understands the concept of tenses. JTP is a theorem prover, so it inherently cannot deal with facts changing, which is something that must be dealt with in a time-aware system. Therefore, the entire class of theorem provers is excluded from the list of potential logic engines for such a system. As far as the project group could tell, no engine exists today that is as powerful as JTP that also understand the concept of time, so a future project would have to create such an engine.

Tense awareness would be beneficial. Changing conditions could be understood by the system. For example, a change in temperature could be expressed to the system, and then the user could query that information, asking for its value at any time in the past. With a sufficiently sophisticated engine, the user could even ask what the average rate of change is over specific time intervals.

6.2.2 Commands

The ability to tell the system to do something, and then have it figure out the various way to make that something happen, and finally have the system perform whatever actions it needs to do, would clearly have many practical applications. For example, one could tell the computer that opening a window causes the temperature to go down. Then, one could tell the computer to make the temperature go down. Logically, the computer could reason that opening the window would have the desired effect, and it would therefore take that action. JTP cannot handle commands without modification, as it cannot bind facts to actions. For example, the step of opening a window cannot be bound to code that will cause some robotic system to open a window. However, as JTP is open source, the project team is confident that future project teams could perform the necessary modifications.

6.2.3 Epistemology

Epistemology is the study of knowing how knowledge is derived, and reasoning about what others know. There are a number of logic problems that can only be solved by reasoning about what other actors know and believe. In everyday life, people use infer about what others know to predict their actions, and derive other such useful knowledge. For example, if A knows that B will not do his homework if tomorrow is a snow day, and A believes that tomorrow will be a snow day, then B can be reasonably assure that A thinks that he will not be doing his homework.

The major problem with epistemology is that there are a number of theories. Some theories yield conclusions which conflict with other theories' conclusions under specific circumstances. Therefore, some set of non-conflicting theories must be accepted, or a number of them may be used and heuristics applied to determine which is most useful in a particular situation.

6.2.4 Mathematics

JTP already provides mechanisms to do a great deal of mathematics. However, the linguistic features were not implemented, so this feature was not examined. The benefits of being able to do math in a future implementation are many and are quite obvious, so they shall not be listed here. The project team believes that most Lojban mathematics structures can be done with JTP, or with minor modifications to it.

6.3 Text to Speech

There are several remaining issues with the speech synthesizer system than need to be resolved.

6.3.1 Part of Speech Tagging

Several of the pausing rules depend on the part of speech of the current word and preceding and following words. The current `LojbanVoice` implements three of the seven pausing rules, but recognizing part of speech would allow for six of the rules to be followed. The last remaining rule depends on recognizing Lojban-ized words from other languages. It could be assumed that this is the case if it does not match any other part of speech but we could also simply require that the user provides the preceding and following periods to indicate the pauses.

6.3.2 Syllables and Stress

`LojbanVoice` currently does not follow any of the Lojban stress rules. In order to do this, part of speech tagging would need to be implemented and the words would need to be broken up into syllables. Proper syllable breakups could be done based on the language rules (the rules can be found in section X) and then the stressed syllable could be marked properly.

6.3.3 Additional UtteranceProcessors

The addition of the part of speech tagger and the syllabifier would necessitate four additional *`UtteranceProcessor`*s: one for tagging part of speech, one for adding pre- and post-pauses, one for separating syllables, and one for determining the stressed syllable. These four processors would enable the voice to speak utterances which more strictly complied with the requirements of the Lojban language.

Appendix A

Pronouncing Lojban

This appendix contains several tables describing the letters and diphthongs in Lojban and their corresponding pronunciation. The information is taken from Chapter 3 of [8]. Each table shows the ASCII representation of the IPA symbol and a description of the phone's features.

Letter	**IPA**	**Description**
a	[a], [A]	an open vowel
e	[E], [e]	a front mid vowel
i	[i]	a front close vowel
o	[o], [O]	a back mid vowel
u	[u]	a back close vowel
y	[@]	a central mid vowel

Table A.1: List of Lojban vowels and their associated IPA phones.

Letter	IPA	Description
ai	[aj]	an open vowel with palatal off-glide
ei	[Ej]	a front mid vowel with palatal off-glide
oi	[oj]	a back mid vowel with palatal off-glide
au	[aw]	an open vowel with labial off-glide
ia	[ja]	an open vowel with palatal on-glide
ie	[jE]	a front mid vowel with palatal on-glide
ii	[ji]	a front close vowel with palatal on-glide
io	[jo]	a back mid vowel with palatal on-glide
iu	[ju]	a back close vowel with palatal on-glide
ua	[wa]	an open vowel with labial on-glide
ue	[wE]	a front mid vowel with labial on-glide
ui	[wi]	a front close vowel with labial on-glide
uo	[wo]	a back mid vowel with labial on-glide
uu	[wu]	a back close vowel with labial on-glide
iy	[j@]	a central mid vowel with palatal on-glide
uy	[w@]	a central mid vowel with labial on-glide

Table A.2: List of valid Lojban diphthongs and their associated IPA phones.

Letter	IPA	Description
’	[h]	a unvoiced glottal spirant
,	—	the syllable separator
.	[?]	a glottal stop or a pause
b	[b]	a voiced bilabial stop
c	[S], [s.]	an unvoiced coronal sibilant
d	[d]	a voiced dental/alveolar stop
f	[f], [P]	an unvoiced labial fricative
g	[g]	a voiced velar stop
j	[Z],[z]	a voiced coronal sibilant
k	[k]	an unvoiced velar stop
l	[l], [l’]	a voiced lateral approximant (may be syllabic)
m	[m], [m.]	a voiced bilabial nasal (may be syllabic)
n	[n], [n-], [N]	a voiced dental or velar nasal (may be syllabic)
p	[p]	an unvoiced bilabial stop
r	[r], [*], [R], [r-], [*-], [R-]	a rhotic sound
s	[s]	an unvoiced alveolar sibilant
t	[t]	an unvoiced dental/alveolar stop
v	[v], [B]	a voiced labial fricative
x	[x]	an unvoiced velar fricative
z	[z]	a voiced alveolar sibilant

Table A.3: Lojban alphabet characters and their associated IPA phones.

Glossary

Backus-Naur form (BNF)

A formal syntax for representing grammars 2.4

bridi

A Lojban sentence; the application of a selbri relation to its sumti arguments 2.1, 4.2, 5.1

brivla

A class of constructs usable as a selbri, includes gismu and tanru; similar to verbs 5.4

cmavo

A Lojban structure word 2.1, 5.4

cmene

Lojban equivalent of a proper noun 2.1, 5.1, 5.4

coarticulation

The assimilation of the place of articulation of one speech sound to that of an adjacent speech sound. 2.5

concatenate

To connect two or more strings together into one string. 2.5

diphone

A diphone is half of one phone followed by half of the next phone. 2.5, 3.5

diphthong

Two vowels whose juxtaposition alters the phones they produce. 2.5, 3.5, 4.4, 5.4

First Order Logic (FOL)

first-order logic permits the formulation of quantified statements 2.2, 2.3, 3.3, 4.2, 5.1, 5.2

FreeTTS

FreeTTS is a speech synthesis system written entirely in the Java programming language. 2.6, 3.5, 4.4

gismu

A Lojban root word, five characters long with a meaning defined in relation to 0-5 parameters 2.1, 5.1

Graphical User Interface (GUI)

A method of interacting with a computer by manipulated graphics 5.3

halting problem

In computability theory the halting problem is a decision problem which can be informally stated as follows: Given a description of a program and its initial input, determine whether the program, when executed on this input, ever halts (completes). The alternative is that it runs forever without halting. Alan Turing proved in 1936 that a general algorithm to solve the halting problem for all possible inputs cannot exist. We say that the halting problem is undecidable over Turing machines. 4.2, 5.2

International Phonetic Alphabet (IPA)

A system of phonetic notation devised by linguists to accurately and uniquely represent each of the wide variety of sounds (phones or phonemes) used in spoken human language. It is intended as a notational standard for the phonemic and phonetic representation of all spoken languages. 2.1, 5.4

Java Speech API (JSAPI)

An API specification for a speech synthesis system in Java. The API is an unimplemented portion of the Java library which can be implemented by a 3rd party package. 3.5

Java Theorem Prover (JTP)

JTP is an object-oriented modular reasoning system developed by the Knowledge Systems Laboratory of Computer Science Department in Stanford University. 2.3, 3.3, 4.2, 5.2, 6.2

lexicon

In a text to speech system, the processor which is responsible for determining the pronunciation of a word in phones. 3.5, 4.4, 5.4

Natural Language Processing (NLP)

Natural language processing is a sub-field of artificial intelligence and linguistics. It studies the problems inherent in the processing and manipulation of natural language, and, natural language understanding devoted to making computers ’understand’ statements written in human languages. 2.0, 2.1, 2.4

nonveridical

sumti clause referring to some already existing objects; logically translates into a universal quantification; similar to English the 4.2

phone

A phone is the actual sound that is produced to audibly represent a phoneme. 2.5, 3.5, 4.4, 5.4

phoneme

A sub-word unit of language that is intended to be communicated. 2.5, 3.5, 4.4, 5.4

phonetic language

A language whose phonemes convert directly to specific phones. 2.5

predicate logic

a system of symbolic logic that represents individuals and predicates and quantification over individuals (as well as the relations between propositions) 2.1

probridi

A word that stands in for a bridi; like a pronoun for an entire sentence 3.1, 5.1

propositional logic

Propositional calculus or sentential calculus is a formal deduction system whose atomic formulas are propositional variables. (Compare this to the predicate calculus which is quantificational and whose atomic formulas are propositional functions, and modal logic which may be non-truth-functional.) 2.2, 2.3

prosumti

A word that stands in for a sumti; like an English pronoun 2.1, 3.1, 5.1

recursively enumerable

a countable set S is called recursively enumerable if there is an algorithm that, when given an input typically an integer or a tuple of integers or a sequence of characters eventually halts if it is a member of S. Otherwise, there is no guarantee that the algorithm will halt. 2.2

restrictive

relative clause which limits the objects to which the statement being made applies 3.1

Sapir-Whorf hypothesis

posits a relationship between the grammatical structure of the language a person speaks and the resulting modes of understanding available to the speaker; suggests that the form of language affects how people think 2.1

selbri

A relation expressed between sumti by a bridi; similar to a verb in an English sentence 2.1, 4.2, 5.1

speech synthesis

Speech synthesis is the actual process of generating speech, usually from text. 2.5, 2.6, 3.5

sumti

One of the arguments to a selbri to build a bridi; similar to an English subject or object 2.1, 4.2, 5.1

token

A token is a unique item in an utterance, usually separated by a space. 2.5, 3.5, 4.4

tokenizer

A tokenizer is a processor which it responsible for separating the utterance into tokens and converting them into pronounceable words. 2.5, 3.5, 4.4

utterance

In speech synthesis, an entire portion of text which is intended for synthesis. 2.1, 2.4, 2.5, 3.5

veridical

sumti clause implying the existence of some objects; logically translates to the existential quantifier; similar to English a 4.2

Bibliography

[1] General information about freetts. `http://freetts.sourceforge.net/docs/index.php`, September 2005.

[2] Johnathan Allen, M. Sharon Hunnucutt, and Dennis Klatt. *From Text to Speech: The MITalk System*. Press Syndicate of the University of Cambridge, The Pitt Building, Trumpington Street, Cambridge CB2 1RP, 1987.

[3] Alan W Black, Paul Taylor, and Richard Caley. The festival speech synthesis system, system documentation. Available online, retrieved 6 September 2005, June 1999.

[4] Geoff Bristow. *Electronic Speech Synthesis: Techniques, Technologies and Applications*. McGraw-Hill Book Company, New York, 1984.

[5] James Cooke Brown. Loglan. *Scientific American*, June 1960.

[6] W.F. Clocksin and C.S. Mellish. *Programming in Prolog*. Springer, New York, 1994.

[7] John Cowan and Nick Nicholas. *What Is Lojban?: .i la lojban. mo*. Logical Language Group, 2904 Beau Lane, Fairfax VA 22031, 2003.

[8] John Woldemar Cowan. *The COMPLETE Lojban Language*. The Logical Language Group, 2904 Beau Lane, Fairfax VA 22031, 1997. `http://xiron.pc.helsinki.fi/lojban`.

[9] Melvin Fitting. *First-Order Logic and Automated Theorem Proving*. Springer-Verlag, 175 Fifth Avenue, New York, NY, 1990.

[10] Robert Grimm. Practical packrat parsing. Technical Report TR2004-854, New York University, Dept. of Computer Science, New York, 2004.

[11] Mathematics and Computer Science Division of Argonne National Laboratory. Otter: An automated deduction system. `http://www-unix.mcs.anl.gov/AR/otter/`, 2005.

[12] Nick Nucholas and Robin Turner. Lojban for beginners – velcli befi la lojban. bei loi co'a cilre. `http://ptolemy.tlg.uci.edu/~opoudjis/lojbanbrochure/lessons/`, September 2005.

[13] Klaus K. Obermeier. *Natural Language Processing Technologies in Artificial Intelligence*. Halsted Press, 605 Third Avenue, New York, NY, 1989.

[14] Stanford University. Jtp: An object-oriented modular reasoning system. `http://www.ksl.stanford.edu/software/JTP/`, 2005.