

2010

Analysis and Design of Speech-Recognition Grammars

Yue Shi

University of Windsor

Follow this and additional works at: <http://scholar.uwindsor.ca/etd>

Recommended Citation

Shi, Yue, "Analysis and Design of Speech-Recognition Grammars" (2010). *Electronic Theses and Dissertations*. Paper 417.

This online database contains the full-text of PhD dissertations and Masters' theses of University of Windsor students from 1954 forward. These documents are made available for personal study and research purposes only, in accordance with the Canadian Copyright Act and the Creative Commons license—CC BY-NC-ND (Attribution, Non-Commercial, No Derivative Works). Under this license, works must always be attributed to the copyright holder (original author), cannot be used for any commercial purposes, and may not be altered. Any other use would require the permission of the copyright holder. Students may inquire about withdrawing their dissertation and/or thesis from this database. For additional inquiries, please contact the repository administrator via email (scholarship@uwindsor.ca) or by telephone at 519-253-3000ext. 3208.

Analysis and Design of Speech- Recognition Grammars

by

Shi, Yue

A Dissertation
Submitted to the Faculty of Graduate Studies
Through the School of Computer Science
in Partial Fulfillment of the Requirements for
the Degree of Doctor of Philosophy at the
University of Windsor

Windsor, Ontario, Canada

2010

© 2010 Shi, Yue

Analysis and Design of Speech-Recognition Grammars

by
Shi, Yue

APPROVED BY:

Dr. Diana Inkpen, External Examiner
School of Information Technology and Engineering, University of Ottawa

Dr. Tanja Collet-Najem
Department of Languages, Literatures and Cultures (French)

Dr. Joan Morrissey
School of Computer Science

Dr. Jianguo Lu
School of Computer Science

Dr. Richard A. Frost, Advisor
School of Computer Science

Dr. Abdul A. Hussein, Chair of Defense
Department of Mathematics and Statistics

Sep 15, 2010

Sep 15, 2010

Author's Declaration of Originality

I hereby certify that I am the sole author of this thesis and that no part of this thesis has been published or submitted for publication.

I certify that, to the best of my knowledge, my thesis does not infringe upon anyone's copyright nor violate any proprietary rights and that any ideas, techniques, quotations, or any other material from the work of other people included in my thesis, published or otherwise, are fully acknowledged in accordance with the standard referencing practices. Furthermore, to the extent that I have included copyrighted material that surpasses the bounds of fair dealing within the meaning of the Canada Copyright Act, I certify that I have obtained a written permission from the copyright owner(s) to include such material(s) in my thesis and have included copies of such copyright clearances to my appendix.

I declare that this is a true copy of my thesis, including any final revisions, as approved by my thesis committee and the Graduate Studies office, and that this thesis has not been submitted for a higher degree to any other University of Institution.

Abstract

Currently, most commercial speech-enabled products are constructed using grammar-based technology. Grammar design is a critical issue for good recognition accuracy. Two methods are commonly used for creating grammars: 1) to generate them automatically from a large corpus of input data which is very costly to acquire, or 2) to construct them using an iterative process involving manual design, followed by testing with end-user speech input. This is a time-consuming and very expensive process requiring expert knowledge of language design, as well as the application area. Another hurdle to the creation and use of speech-enabled applications is that expertise is also required to integrate the speech capability with the application code and to deploy the application for wide-scale use.

An alternative approach, which we propose, is 1) to construct them using the iterative process described above, but to replace end-user testing by analysis of the recognition grammars using a set of grammar metrics which have been shown to be good indicators of recognition accuracy, 2) to improve recognition accuracy in the design process by encoding semantic constraints in the syntax rules of the grammar, 3) to augment the above process by generating recognition grammars automatically from specifications of the application, and 4) to use tools for creating speech-enabled applications together with an architecture for their deployment which enables expert users, as well as users who do not have expertise in language processing, to easily build speech applications and add them to the web.

Dedication

To my parents for their endless and unconditional love

and

To my sister and brothers for their strong support and encouragement

and

To my husband for his enduring understanding, support, and love

and

To my daughter for her sweet love and

being the source of my joy.

Acknowledgements

First and foremost, I would like to thank Dr. Richard A. Frost, my advisor, for his enduring help, guidance, and ideas in my Master's and Doctoral studies. Throughout my graduate study, Dr. Frost has encouraged me to exceed my own expectations. Dr. Frost has provided extensive proofreading of my research papers, surveys, and thesis. Without his consideration, understanding, and encouragement, I could not have completed this work.

I would like to thank Dr. Diana Inkpen for her time and consideration to be my external examiner, and for the extensive constructive comments which she made which have helped me to improve the presentation of my results and conclusions. Also, I would like to thank Dr. Tanja Collet-Najem, Dr. Joan Morrissey, and Dr. Jianguo Lu, for serving on my dissertation committee, and for providing valuable guidance and advice. I'd like to express my sincere thanks to Dr. Collet-Najem, for spending so much time on my work with valuable feedback. Special thanks are sent to Dr. Morrissey who has been supportive of me since my Master's study. Her consideration and encouragement have helped me through the long journey in my graduate studies. Also, I'd like to present many thanks to Dr. Jianguo Lu for spending a great deal of time and effort on my thesis and providing valuable comments. In addition, I'd like to thank Dr. Hussein for serving as chair on my dissertation committee.

I am very grateful to the faculty, staff, and colleagues in the School of Computer Science at University of Windsor for their help and support in my graduate studies. Thanks to my colleagues Nabil Abdullah and Hafiz Rahmatullah, who have worked with me in the same laboratory, and other friends for giving me their time and support.

Last but not the least, I would like to say that this research would have been impossible without the unconditional love and support from my whole family, my parents, sister, brothers, my husband, and my daughter. To support my study,

my husband has given up much that is extremely important for him. My daughter's sweet kiss and hug dissolve all my tiredness and stress. It is my family's unconditional love that has supported me throughout the hardships during the long and difficult journey of my graduate study. I am completely indebted to them and nothing I say or do can ever repay that debt.

TABLE OF CONTENTS

Author's Declaration of Originality	iii
Abstract	iv
Dedication.....	v
Acknowledgements	vi
List of Tables	xi
List of Figures	xii
List of Appendices	xiv
1. INTRODUCTION	1
1.1 The Problem	3
1.2 An Alternative Less-Expensive Approach	4
1.2.1 Average Branching Factor (ABF).....	4
1.2.2 Semantic Constraint Grammar (SCG)	5
1.2.3 Automatic generation of speech-recognition grammars	5
1.2.4 A Public-Domain SpeechWeb	6
1.3 The Thesis Statement.....	7
1.3.1 Importance of the thesis	8
1.3.2 Proof of the thesis statement.....	9
1.4 Contributions of This Thesis Work	10
1.5 The Structure of This Thesis Report	11
2. AN ALGORITHM FOR COMPUTING THE AVERAGE BRANCHING FACTOR (PERPLEXITY) OF SPEECH-RECOGNITION GRAMMARS	14
2.1 The Need for Grammar Metrics.....	15
2.2 Existing Grammar Metrics.....	16
2.3 Analysis of Existing Grammar Metrics.....	18
2.4 Introduction to the Average Branching Factor (ABF).....	20
2.5 Constraints on Speech-Recognition Grammars	21
2.6 Preliminary Discussion of the ABF Algorithm	22
2.7 The New ABF Algorithm.....	27
2.8 More Examples	31
2.8.1 Example 1.....	32
2.8.2 Example 2.....	34
2.8.3 Example 3.....	38
2.9 Implementation of the ABF Algorithm.....	40
2.10 Summary.....	44

3. PROOFS OF THE ABF ALGORITHM	46
3.1 Introduction to Miranda	46
3.2 Miranda Code for the ABF Algorithm	48
3.3 Proof of Termination.....	58
3.4 Proof of Correctness	64
3.5 Proof of Complexity	110
4. SEMANTIC CONSTRAINT GRAMMARS (SCG).....	116
4.1 Context-Free Grammars (CFG)	117
4.2 Semantic Constraint Grammars (SCG).....	118
4.3 An Example of a CFG and a Related SCG	120
4.3.1 The example CFG	121
4.3.2 The example SCG	124
4.4 Analyze the CFG and SCG Using Grammar Metrics	130
4.5 Results from an Experiment Investigating Speech- Recognition Accuracy	132
4.6 Comparison of ABFs and Speech Recognition Accuracy Results	133
4.7 Summary.....	136
5. AUTOMATIC GENERATION OF SPEECH-RECOGNITION GRAMMARS FROM RELATIONAL DATABASE SCHEMAS	137
5.1 Related Work and Its Shortcomings.....	137
5.1.1 Related work.....	137
5.1.2 Shortcomings of related work	138
5.2 The New Approach	141
5.3 Semantic Specification.....	142
5.4 CFG Grammar-Generation System.....	151
5.4.1 The CFG Grammar Template.....	152
5.4.2 The CFG Grammar-Generation Engine.....	154
5.5 The SCG Grammar-Generation System	163
5.5.1 The SCG Grammar Template.....	164
5.5.2 SCG Grammar-Generation Engine.....	168
5.6 Analysis of the Automatically-Generated CFG and SCG Using the Grammar Metrics	175
5.7 Comparison to Related Work	177
5.8 Summary.....	178
6. A PUBLIC-DOMAIN SPEECHWEB.....	179
6.1 Introduction	179
6.2 SpeechWebs.....	179
6.3 The LRRP SpeechWeb Architecture.....	179
6.4 A Public-Domain SpeechWeb	181
6.5 The Example of a Speech Application	184
6.6 Summary.....	189

7. CONCLUSION.....	190
7.1 Proof of the Thesis	190
7.2 Future Work	190
BIBLIOGRAPHY	193
APPENDIX A: A SURVEY- DESIGN OF RECOGNITION GRAMMAR FOR VXML- LIKE APPLICATIONS.....	215
APPENDIX B: A SURVEY – AUTOMATIC GENERATION OF SPEECH- RECOGNITION GRAMMARS	274
APPENDIX C: GRAMMAR – READ A BOOK.....	318
APPENDIX D: GRAMMAR – READ A BOOK (5-word sequence).....	321
APPENDIX E: GRAMMAR – WORD SEQUENCE (SOLAR SYSETM)	323
APPENDIX F: EXAMPLE DATABASE CONNECTIONS	325
APPENDIX G: AUTOMATICALLY GENERATED CFG GRAMMAR FOR THE SOLAR SYSTEM.....	327
APPENDIX H: AUTOMATICALLY GENERATED SCG GRAMMAR FOR THE SOLAR SYSTEM.....	329
APPENDIX I: THE XML FILE FOR SPEECH APPLICATION Read-A-Book .	333
APPENDIX J: AN EXCERPT OF THE INTERPRETER FOR SPEECH APPLICATION Read-A-Book	351
APPENDIX K: THE CGI FILE FOR SPEECH APPLICATION Read-A-Book..	353
APPENDIX L: SAMPLE SCREENSHOTS FOR SPEECH APPLICATION Read-A- Book	354
VITA AUCTORIS	361

List of Tables

1. Table 2.6 (1): summary of JSGF features	22
2. Table 2.6 (2): values for decision points in the derivation tree approach	26
3. Table 2.7.6: information from the annotated grammar	31
4. Table 2.8.1: information from Example grammar 1	34
5. Table 2.8.2: information from Example grammar 2	37
6. Table 2.8.3: information from Example grammar 3	40
7. Table 2.9: results of applying the ABF algorithm	43
8. Table 4.4: grammar metrics of sample CFG and SCG	130
9. Excerpt of Table 2.9: experimental results with ABF algorithm	134
10. Table 7.3.2 (1): the “Correct” feature using the semantics set	135
11. Table 7.3.2 (2): the “Incorrect” feature using the semantics set	135
12. Table 5.3: comparison among the Semantic Specification, ER Modeling, and UML Representation	150
13. Table 5.4.2 (1): database information	154
14. Table 5.4.2 (2): table names and aliases	155
15. Table 5.5.2: example syntactic and semantic constraints	171
16. Table 5.6: comparison among manually-scripted and automatically-generated CFG and SCG grammars	175

List of Figures

1. Figure 2.6.1: a sample CFG grammar	23
2. Figure 2.6.2: sample grammar with director sets and branching factors	23
3. Figure 2.6.3: derivation tree for the example grammar in Figure 2.6.1....	24
4. Figure 2.7.1: annotated example grammar with branching factors	28
5. Figure 2.7.2: annotated grammar with subsizes	28
6. Figure 2.7.3: annotated grammar with the number of occurrences	29
7. Figure 2.7.4: annotated grammar with bracketed left-hand subscripts ...	30
8. Figure 2.7.5: example grammar with decision-points	30
9. Figure 2.8.1: example grammar 1	32
10. Figure 2.8.1: example grammar 1 (Step 1)	32
11. Figure 2.8.1: example grammar 1 (Step 2)	32
12. Figure 2.8.1: example grammar 1 (Step 3)	33
13. Figure 2.8.1: example grammar 1 (Step 4)	33
14. Figure 2.8.1: example grammar 1 (Step 5)	33
15. Figure 2.8.2: example grammar 2	34
16. Figure 2.8.2: example grammar 2 (Step 1)	35
17. Figure 2.8.2: example grammar 2 (Step 2)	35
18. Figure 2.8.2: example grammar 2 (Step 3)	36
19. Figure 2.8.2: example grammar 2 (Step 4)	36
20. Figure 2.8.2: example grammar 2 (Step 5)	37
21. Figure 2.8.3: example grammar 3	38
22. Figure 2.8.3: example grammar 3 (Step 1)	38
23. Figure 2.8.3: example grammar 3 (Step 2)	38
24. Figure 2.8.3: example grammar 3 (Step 3)	39
25. Figure 2.8.3: example grammar 3 (Step 4)	39
26. Figure 2.8.3: example grammar 3 (Step 5)	40
27. Figure 3.2: Miranda source code for the ABF algorithm	52
28. Figure 4.1.1: a sample CFG	117

29. Figure 4.3.1: example CFG with the domain of the Solar system	122
30. Figure 4.3.2: a sample SCG with the domain of the Solar system	125
31. Figure 5.3: example Database schema for the Solar system	144
32. Figure 5.3 (1): properties of entity types	144
33. Figure 5.3 (2): possible activities/ relationships among tables	144
34. Figure 5.3 (3): example Semantic Specification	147
35. Figure 5.4.1: CFG grammar template	153
36. Figure 5.4.2: screenshot - example command to generate a CFG grammar	163
37. Figure 5.5.1: SCG grammar template	166
38. Figure 5.5.2: screenshot - example command to generate a SCG grammar	174
39. Figure 6.4: LRRP SpeechWeb architecture	182
40. Figure 6.5: the screenshot of Read-A-Book application	188

List of Appendices

1. Appendix A: A Survey – Design of Recognition Grammar for VXML-like Applications	215
2. Appendix B: A Survey – Automatic Generation of Speech-Recognition Grammars	274
3. Appendix C: Grammar – Read A Book	318
4. Appendix D: Grammar – Read A Book (5-word Sequence)	321
5. Appendix E: Grammar – Word Sequence (Solar System)	323
6. Appendix F: Example Database Connections	325
7. Appendix G: Automatically Generated CFG Grammar for the Solar System	327
8. Appendix H: Automatically Generated SCG Grammar for the Solar System	329
9. Appendix I: URLs for Example XML Files	333
10. Appendix J: An Excerpt of the Interpreter for Speech Application Read-A-Book	351
11. Appendix K: THE CGI FILE FOR SPEECH APPLICATION Read-A-Book	353
12. Appendix L: Sample Screenshots for Speech Application Read-A-Book	354

1. INTRODUCTION

Currently, most commercial speech-enabled products are constructed using grammar-based technology. The speech recognition engine processes the input and phonetically matches it with sentences that are generated by a top-down expansion of the recognition grammar. The process stops when a match is made which meets a certain confidence level.

One method for creating speech-recognition grammars is to generate them automatically from a large corpus of input data which is very costly to acquire. Another method is to manually design the grammar from a specification of the application and then to test and modify the grammar by experimenting with end-user speech input. The high cost of both of these methods is one of the bottlenecks slowing the production of speech applications (Meng and Siu, 1999, 2002) and (Wang and Acero, 2001, 2003a, 2006). In this thesis, we describe an alternative approach which is based on grammar metrics. Rather than develop a grammar and then improve it through experiment, the proposed approach is to design a grammar, analyze it using grammar metrics which have been claimed by other researchers to be good indicators of recognition accuracy, modify the grammar, re-analyze using the metrics, and iteratively improve the grammar with respect to the metrics. We also facilitate the process of grammar design by showing how recognition accuracy can be improved by coding semantic constraints in the syntax rules. We further augment the process of grammar design by showing how some grammars can be generated automatically from specifications of applications.

One of the major contributions of this dissertation is the presentation of the first algorithm to compute an important grammar metric, the Average Branching Factor (perplexity) of the grammar. The ABF has been claimed by others to be a good indicator of speech-recognition accuracy. We formally prove termination, correctness, and polynomial complexity of our algorithm.

We then further investigate a concept that was the focus of the author's Master's work. The concept is that of Semantic Constraint Grammars (SCGs), which include semantic constraints encoded in the syntax rules. These grammars are still context-free grammars and we use the name "Semantic Constraint Grammar (SCG)" only to indicate that the grammar contains what are usually regarded as semantic constraints in its context-free syntax rule. SCGs are not a new class of grammar. We compare SCGs with the Context-Free Grammars (CFGs) from which they were derived, with respect to a number of grammar metrics, including ABF, number of rules, number of symbols, number of terminals, number of non-terminals, and size of the language. We compare the analysis with respect to metrics to the experimental results of voice recognition accuracy which were obtained as part of a Master's thesis which was completed by the candidate before commencing this doctoral work. The results support the claims that 1) grammar metrics are good indicators of speech-recognition accuracy, and 2) that encoding semantic constraints in the syntax of recognition grammars can improve speech recognition accuracy.

In order to further reduce the cost of creating speech-recognition grammars, we investigate the possibility of generating grammars automatically from application-specific data. We begin by showing that little work has been carried out on this approach. We then show how speech-recognition grammars for a simple spoken database query processor can be generated automatically from a relational-database schema. We generate straightforward recognition grammars and also Semantic Constraint Grammars (SCGs) from the database schemas. We analyze the two types of grammar with respect to their ABFs, and show that the SCGs have lower ABFs, and are likely, therefore to have better recognition accuracy.

In addition to improving the ease with which speech-recognition grammars can be developed, we are also interested in facilitating the deployment of speech-enabled applications. In addition to the work on speech-recognition grammars, the candidate has also contributed to the development of the architecture and sample applications for a Public-Domain SpeechWeb, which is an augmentation

of the conventional web with hyperlinked speech applications that are designed for natural-language speech interaction. As part of this work, the candidate has developed a SpeechWeb application using readily available software technology and commonly-used communication protocols. The application allows users who have access to the Internet to contact a remote application `Read-A-Book` and interact with the book “Sleeping Beauty” through speech input and output. We include a description of this simple application to illustrate the ease with which hyperlinked speech applications can be created and deployed on the Web.

1.1 The Problem

Despite huge improvements in speech-recognition technology, very few speech applications are available to the public. We have observed the following two reasons for this problem:

(1) the high cost of grammar creation.

In general, there are two methods in grammar creation. One approach is a statistical approach, which constructs recognition grammars by analyzing a large corpus of data, which is costly to acquire. The second approach to construct speech-recognition grammars using an iterative process, involving manual design followed by testing with end-user speech input. This is a time-consuming and very expensive process requiring expert knowledge of language design as well as the application area.

(2) the difficulty in integrating speech-recognition component with application code.

It requires much expertise to integrate the speech capability with the application code and to deploy the application for wide scale use.

1.2 An Alternative Less-Expensive Approach

To tackle the above problem, we propose an alternative approach which should be less expensive as it does not require end-user testing. The approach is based on grammar metrics and which we present from the following four perspectives: Average Branching Factor (ABF), Semantic Constraint Grammars (SCG), automatic generation of speech-recognition grammars, and a Public-Domain SpeechWeb.

1.2.1 Average Branching Factor (ABF)

Instead of the iterative process of grammar design followed by testing with end-user speech input, we use metrics to assess the quality of recognition grammars as they are developed either by hand or automatically. In particular, the Average Branching Factor (ABF) is claimed by other researchers to be a good indicator of recognition accuracy. We will define the ABF later in this sub-section.

Speech-recognition accuracy is a significant issue that researchers have been working on for many decades. Grammar features have been studied from a variety of perspectives in order to improve the performance of speech applications.

The accuracy of speech recognition is dependent on the Average Branching Factor (ABF) of the recognition grammar. Grammars with lower ABFs are likely to have better recognition accuracy than those with higher ABFs (Hauptmann et. al., 1988), (Young et. al., 1989), (Young, 1990), (Waibel and Lee, 1990), (Edelkamp and Korf, 1998), and (Morimoto and Takahashi, 2008, 2009). Much work has been carried out to reduce the ABF. However, there would appear to be no published algorithm that computes the ABFs directly from recognition grammars.

In grammar-directed speech recognition, the *branching factor* of a single decision point is the number of possible words to be considered as candidates at that point. During the recognition process, if at any point, it needs to examine the next

symbol on the input to make a choice (even if the choice is a single branch), this point is a *decision point*. The *Average Branching Factor (ABF)* is the average of the branching factors of all decision points in all of the utterances in the language defined by the recognition grammar. The ABF is also called the *perplexity* of the language.

The Average Branching Factor (ABF) is one grammar metric. Other grammar metrics include size (number of sentences) of language, number of rules, number of symbols, number of terminals, and number of non-terminals.

1.2.2 Semantic Constraint Grammar (SCG)

A Semantic Constraint Grammar (SCG) is a Context-Free Grammar (CFG) that encodes semantic constraints in the syntax rules of the grammar to reduce the language size and the ABF of the CFG grammar.

In the candidate's Master's work (Shi, 2003b), we carried out experiments to investigate the recognition accuracy of SCGs and CFGs with an iterative testing process with end-user speech input and test case design. In this doctoral dissertation, we compare SCGs with the CFGs with respect to a set of grammar metrics, including ABF, number of rules, number of symbols, number of terminals, number of non-terminals, and size of the language. We compare and analyze the results from the Master's experiment (Shi, 2003b) and the grammar metrics in this dissertation work. The results support the claims that 1) grammar metrics are good indicators of speech-recognition accuracy, and 2) that encoding semantic constraints in the syntax of recognition grammars can improve speech recognition accuracy.

1.2.3 Automatic generation of speech-recognition grammars

Although technology for grammar-based speech applications is readily available, it is not yet being extensively used to create speech applications. One problem is

that there appears to be a lack of theory and tools to facilitate the construction of speech-recognition grammars.

Currently, most grammars for speech-enabled applications are written manually, which is costly, laborious, and error-prone. Writing a domain-specific grammar has been a barrier to the rapid development of spoken-language systems (Meng and Siu, 2002), (Wang and Acero, 2003a), and (Wang and Acero, 2006).

Spoken language often contains repetitions, corrections, interruptions, or unfinished utterances. These phenomena are often referred as disfluencies (Jørgensen, 2007). Due to the disfluencies and non-grammatical utterances of spoken language, a handcrafted grammar cannot guarantee a good coverage of the input language when deployed in real applications (Meng and Siu, 2002). Bangalore and Johnston (2003) point out that the heavy cost of authoring and maintaining grammars and the lack of coverage in the rule sets, are the main reasons for the bottleneck in the development of conversational systems. Wang and Acero (2001, 2005, 2006) conclude that writing domain-specific grammars is a major obstacle in making conversational systems mainstream.

In this dissertation, we show that for some simple applications it is possible to generate grammars automatically from application specifications. We illustrate this by generating speech-recognition CFG and SCG grammars automatically from relational database schemas. This approach further reduces the cost, time, effort, and the requirement of linguistic knowledge in grammar authoring.

1.2.4 A Public-Domain SpeechWeb

The growth of the Internet since the early 1990's has changed people's lives by providing access to huge amounts of information on the web. It also expands commercial opportunities for business and convenience for customers, by allowing business transactions to be conducted anytime around the clock and anywhere around the world. However, the conventional web is mainly based on text and graphics, which excludes people with visual disabilities and limits the

use of the web where it is not convenient to access it by hand. Therefore, we need a supplement to the traditional web. One approach is to augment the web with hyperlinked speech applications, collectively referred as a SpeechWeb (Frost and Chitte 1999) and Frost (2004).

By taking the advantage of the rapidly developing wireless industry, a SpeechWeb that can be accessed via cell phones, will undoubtedly bring profit to business. Hartzell (2003) estimated that speech-enabled services would generate more than \$4.6 billion in revenue for North American wireless carriers and \$25 billion worldwide by the end of the middle to end of the first decade of the 21st century.

In addition, the hyperlinking of applications solves the problem of low accuracy of large recognition grammars by enabling large applications to be partitioned into smaller hyperlinked components that use smaller grammars with better recognition accuracy.

1.3 The Thesis Statement

The thesis is that natural-language speech-recognition grammars are amenable to methodical analysis and design techniques. In particular:

- (1) Various grammar metrics, including the Average Branching Factor (ABF) can be computed automatically and efficiently.
- (2) Semantic constraints can be encoded in syntax rules in order to decrease language size and ABF.
- (3) Recognition grammars can be created automatically from relational database schemas and application specifications.
- (4) Readily-available speech-recognition technology and commonly-used communication protocols can be used by non-expert as well as expert users to create and deploy speech applications.

1.3.1 Importance of the thesis

This thesis is an attempt to tackle the problem stated in sub-section 1.1. Through extensive surveys, which are presented in the Appendices, we have shown that this problem has not yet been solved.

Our solution to the problem is important because of our “constructive” proofs which involved the creation of algorithms and software, which contribute to the viability of the alternative less-expensive method for creating speech applications.

In particular:

(1) The importance of the ABF algorithm

The ABF is an important grammar metric that determines the recognition accuracy of speech-recognition grammars. Our algorithm for computing the Average Branching Factor (ABF) directly from a speech-recognition grammar makes it possible to use the ABF to preliminarily examine and assess recognition grammars avoiding costly and time-consuming experimentation involving iterative user speech input testing. With the assistance of the ABF, time, cost, and effort are reduced in grammar design and development.

(2) The importance of Semantic Constraint Grammars (SCGs)

SCGs encode semantic constraints in the syntax rules to naturally decrease the size of language and ABF therefore improve recognition accuracy.

(3) The importance of the approach for automatic grammar generation

The approach of generating speech-recognition grammars automatically from relational database schemas illustrates that this approach could facilitate the development of speech-enabled applications and services. It has the potential to significantly reduce the time, cost, and difficulty in speech-recognition grammar authoring. Using the proposed approach, a developer with little

linguistic knowledge and grammar scripting experience can create a high-quality speech-recognition grammar. More importantly, the method that we have developed demonstrates that recognition grammars with good recognition accuracy can be generated automatically.

(4) The importance of a Public-Domain SpeechWeb

The Public-Domain SpeechWeb architecture (Frost, 2005) makes it possible for expert users and users who do not have expertise in language processing to easily develop and deploy speech applications in the public-domain SpeechWeb. In addition, the SpeechWeb provides a solution to improve recognition accuracy for large grammars by dividing the application into small hyperlinked components which have smaller grammars and consequently better recognition accuracy.

1.3.2 Proof of the thesis statement

The thesis has been proven by “constructive proofs”. Algorithms and software have been built and analyzed in order to prove each part of the thesis by construction of an example. Such proofs are informal and are really “proof of concept”. However, formal mathematic proofs have been used to analyze properties of the algorithm developed.

To prove the thesis statement, we consider a set of speech-recognition grammar metrics, including the number of symbols, the number of terminals, the number of non-terminals, the number of rules, the size of the language, and the Average Branching Factor (ABF). We also review metrics proposed by others as follows. Details of existing metrics are in sub-section 2.2.

- (1) McCabe’s Complexity (McCabe, 1976) metric measures the number of linearly independent paths through a flow graph.
- (2) Fenton’s Impurity (Fenton and Pfleeger, 1996) describes successors’ dependency between non-terminals.

- (3) Power and Malloy (2000, 2004) discuss the following grammar metrics: the average size (the number of terminals or non-terminals) of the right-hand-side of syntax rules, levels, and depth (the number of non-terminals in the largest grammatical level).
- (4) Grammar Confusability Metrics (GCM) (Cai and Hamaker, 2008) describe a likelihood that a phrase will be confused by the speech recognizer with another phrase currently allowed by an active grammar rule.

We then describe an efficient algorithm for determining the Average Branching Factor (ABF) automatically from a speech-recognition grammar. We formally prove termination, correctness, and polynomial time complexity. We have implemented the algorithm and applied it to several example grammars.

We have developed a method to improve grammars with respect to the metrics by integrating semantics with syntax. We use a set of grammar metrics to measure the properties of the initial grammars, and the revised “semantic” grammars (SCGs).

We have also developed a method of generating speech-recognition grammars (CFGs or SCGs) automatically from relational database schemas. The generated grammars are analyzed and compared using the set of grammar metrics.

Finally, we have created a small SpeechWeb application to illustrate the ease with which grammar-based speech applications can be developed and deployed on the web.

1.4 Contributions of This Thesis Work

The contributions of this thesis work are summarized as follows:

- (1) We have proposed the first algorithm to correctly compute the Average Branching Factor (Perplexity) directly from a speech-recognition grammar. This algorithm provides a method for more-easily and less-expensively calculating a grammar metric which is useful when developing speech-

recognition grammars. The algorithm has been formally proven with respect to termination, correctness, and polynomial time complexity.

- (2) We have further investigated the notion of Semantic Constraint Grammars (SCGs), and compared SCGs and CFGs with respect to a set of grammar metrics.
- (3) We have proposed a novel approach to generate speech-recognition grammars automatically from relational database schemas and application specification. The approach has been implemented and the generated grammars are analyzed using a set of grammar metrics. This approach demonstrates the viability of automatic grammar authoring and facilitates the development of conversational systems.
- (4) The example SpeechWeb application illustrates the ease of creating and deploying grammar-based speech applications in a Public-Domain SpeechWeb using readily available technology and protocols.

1.5 The Structure of This Thesis Report

The remainder of this thesis report is structured as follows:

Section 2 begins by discussing the need for recognition grammar metrics and the definition of the Average Branching Factor (ABF). We analyze existing grammar metrics and compare them with the ABF metric. Before presenting the new algorithm for computing the ABF, we discuss a naïve approach which is incorrect. Then, we illustrate a correct but impractical approach which has exponential complexity with respect to the length of the utterances. Next, we present the new polynomial time ABF algorithm. We include three examples of applying the ABF algorithm and discuss the implementation of the algorithm. The results of applying the ABF algorithm on several example grammars are included and discussed in this section too.

Section 3 contains proofs of the ABF algorithm with respect to termination, correctness, and polynomial time complexity. To better present the proofs, we

include the Miranda source code for the ABF algorithm implementation. An introduction to Miranda is included in section 3.

In section 4, we discuss and compare Context-Free Grammars (CFGs) and Semantic Constraint Grammars (SCGs). Examples of these two grammars are included and analyzed using a set of grammar metrics. In addition, we refer to the experimentation for investigating speech recognition accuracy which was carried out in the candidate's Master's work. The experimental results from the Master's work are compared with a new analysis of the grammars using the ABF and other metrics.

Section 5 presents a novel approach for generating speech-recognition grammars automatically from relational database schemas and application specifications. We further discuss, analyze, and compare the generated CFGs/SCGs using a set of grammar metrics.

We discuss the Public-Domain SpeechWeb and the LRRP (Local speech Recognition and Remote Processing) SpeechWeb architecture in section 6. We illustrate the ease of creation, deployment, and access to a hyperlinked speech application using an example speech application.

Section 7 summarizes the work done, concludes the thesis report, and discusses future work.

Two surveys are appended, Appendix A: "A Survey – Design of Recognition Grammar for VXML-like Applications", and Appendix B: "A Survey – Automatic Generation of Speech-Recognition Grammars". Appendix C and D are two example grammars with the same vocabulary, *Read-A-Book* and a word sequence grammar. Appendix E is a word-sequence grammar for a small solar system which has the same domain with the example grammars in section 4. Appendix F includes Java code for example database connection. Two automatically generated grammars are in Appendix G and Appendix H respectively. Appendix I lists the URLs for the XML files of the example speech

applications of the Public-Domain SpeechWeb. Appendix J includes parts of the interpreter for the example speech application `Read-A-Book`. Appendix L includes some sample screenshots for the example speech application `Read-A-Book`.

2. AN ALGORITHM FOR COMPUTING THE AVERAGE BRANCHING FACTOR (PERPLEXITY) OF SPEECH- RECOGNITION GRAMMARS

Although speech-recognition technology has achieved significant progress over decades, it is not yet perfect. Speech recognition is not an exact, analytical science, but a probabilistic art and incorporates elements of sophisticated guessing (Abbott, 2001). There are still many limitations in voice applications.

Stochastic (statistical) techniques and grammar-based techniques are two of the main methods used in natural language speech processing. Stochastic models were the most popular up to the late 90's, whereas grammar-based technology has been more widely used in commercial products since 2001 (Barnard et al., 1999), (Knight et al., 2001) and (Caskey et al., 2003).

In grammar-based speech applications, recognition grammars are a key component that directly affects the performance of speech applications. The design of speech-recognition grammars determines speech-recognition accuracy, robustness, efficiency, and maintenance complexity of speech applications. A well-defined grammar also provides the user with great flexibility and comfort in voice services. Good grammars are essential for the usability of a speech application. However, writing grammars is a daunting and expensive task, which forms a major bottleneck in the development of spoken language systems (Siu and Meng, 1999).

In the survey at Appendix A, we have reviewed the design of recognition grammar for VXML-like applications. VoiceXML is an XML-based markup language for building distributed voice applications, much as HTML is a markup language for creating distributed visual applications. A grammar is a set of rules that define the possible words, phrases, or utterances which are accepted by the

2. An Algorithm for Computing the Average Branching Factor (Perplexity) of Speech-Recognition Grammars

speech recognition engine. A grammar is a fundamental building block of speech technology (Mané and Levin, 2005).

From first-hand experience in writing grammars for real-world voice applications, some guidelines in VoiceXML application grammar design have been overviewed and summarized in Appendix A. The topics included grammar design, dialog design, prompt design, sub-grammar design, sub-dialog design, grammar weights and probabilities, error handling, and testing. Also, we have reviewed the design for Voice User Interface (VUI), which is the key to the success of a VoiceXML application, and the tools and environments for grammar design.

In summary, we have found over 20 articles on grammar design and 4 of which are refereed scientific papers. More than 10 articles are related to Voice User Interface (VUI) design and 2 of which are refereed scientific papers. 4 non-refereed articles talk about voice-application testing. 15 voice application development environments are available, 4 of which are freely downloadable.

2.1 The Need for Grammar Metrics

In speech application development, carrying out experiments is one of the major methods to test speech-recognition accuracy and performance. This process may involve the following phases:

- (1) Design and develop a speech-recognition grammar.
- (2) Link the speech-recognition grammar to a speech application.
- (3) Design a set of test cases.
- (4) Identify a number of subjects (testers) to participate the testing.
- (5) Testers go through the set of test cases and record their results.
- (6) Analyze the results and summarize the performance of the speech-application.

For each change of a speech-recognition grammar, the testing process has to be repeated. Testing and tuning are an iterative process for analyzing and

2. An Algorithm for Computing the Average Branching Factor (Perplexity) of Speech-Recognition Grammars

optimizing system performance. This process is a complex task that can take a long time (sometimes, several months) and involves an interdisciplinary team of professionals, including developers, testers, linguists, and psychologists (Biber and Kozminski, 2005) and (Eisenzopf, 2006).

In software engineering, software metrics are often used to measure and manage the complexity of software and estimate the difficulty of maintenance in order to determine the cost of change, and as an indicator for automatic detection of a transformation that can improve the quality of a system (Power and Malloy, 2004). The use of software metrics is essential to good software engineering (Fenton and Pfleeger 1996) and (Power and Malloy, 2004).

Similarly, there is a need for a set of grammar metrics to analyze and estimate the performance of speech-recognition grammars so that the time, cost, and difficulty in grammar design may be reduced.

2.2 Existing Grammar Metrics

Features of speech-recognition grammars have been studied and analyzed from a variety of perspectives over many years. Researchers have developed a variety of grammar metrics.

Power and Malloy (2000) developed a technique to map six established metrics in software engineering to grammars, and extend their work in (Power and Malloy, 2004). The six metrics are as follows:

- (1) The number of terminals and non-terminals.
- (2) The McCabe Complexity measures the linearly independent paths through a flow graph (McCabe, 1976). It is typically interpreted as a measure of the number of decisions in the flow graph. In a CFG, decisions are made at non-terminals while the recognition process choosing next alternatives. The McCabe complexity for a CFG is the total number of alternatives in the grammar.

2. An Algorithm for Computing the Average Branching Factor (Perplexity) of Speech-Recognition Grammars

- (3) The average RHS size is the average of the number of symbols, including terminals and non-terminals, on the right-hand side of CFG production rules. It is the quotient of the total of terminals and non-terminals on the right-hand-side of the rules divided by the number of non-terminals.
- (4) The Fenton's Impurity (Fenton and Pfleeger, 1996) concerns the dependencies between procedures in a program in software engineering. The dependencies between procedures in the program are edges of a call graph, which is represented as a directed graph. In a CFG, a non-terminal is regarded as a procedure of a program and the successor relations between non-terminals are edges in the call graph. The Fenton's Impurity defines the dependencies of non-terminals of a CFG.
- (5) Grammatical Levels (Power and Malloy, 2000 and 2004). If non-terminal A derives some sequence of symbols β , and β contains some non-terminal B , we say that B is an immediate successor of A , and write $A \triangleright B$. If β derives some sequence of symbols γ , and γ contains some non-terminal C , we say that C is a successor of A , and write $A \triangleright^* C$. If $A \triangleright^* C$ and $C \triangleright^* A$, we say that A is equivalent to C and denote as $A \equiv C$. An equivalence relation on a set partitions the set into a collection of equivalence classes. All the elements in a given equivalence class are considered equivalent. For grammar non-terminals, these equivalence classes are called *grammatical levels*.
- (6) Depth. Based on the definition of grammatical level, *Depth* is defined as the number of non-terminals in the largest grammatical level (Power and Malloy, 2000 and 2004).

Cai and Hamaker (2008) proposed a Grammar Confusability Metric (GCM) to describe a likelihood that a phrase will be confused by the speech recognizer with another phrase currently allowed by an active grammar rule. The GCM identifies pairs of phrases in the grammar with different semantic meanings which are difficult for the speech recognizer to distinguish reliably. For example, the user says, "repeat this voicemail". The speech recognizer may misrecognize it as

2. An Algorithm for Computing the Average Branching Factor (Perplexity) of Speech-Recognition Grammars

“delete this voicemail” because they are acoustically alike. The GCM is used to flag the existence of two phrases in the grammar that have different semantic meanings but with similar pronunciation.

A Probabilistic Context-Free grammar (PCFG) is a Context-Free Grammar (CFG) (the formal definition of a CFG is given in sub-section 4.1) with a probability distribution defined over all productions that share the left-hand side (Rosenfeld, 2000b), (Moore, 1999), and (Weber and Görz, 1999). The entropy of a probabilistic CFG is computed in (Kuich, 1970) and further studied in (Soule, 1974) and (Justensen and Larsen, 1975). Using CFGs to categorize the ways in which nodes branch to yield daughter nodes, Miller and O’Sullivan (1992) examine the entropies of the branching processes associated with trees that branch according to a finite number of rules. Miller and O’Sullivan use the theory of multi-type Galton Watson processes (Harris, 1963), these processes are characterized according to their branching rates as sub-critical, critical, and super-critical with branching rates $\rho < 1$, $\rho = 1$, and $\rho > 1$ respectively. The *branching rate* is the rate of growth of the logarithm of the total number of derivations from the grammar (the total number of trees possible starting from the start node) (Miller and O’Sullivan, 1992). To characterize the rate of growth of the number of derivations in the language, Miller and O’Sullivan (1992) observe that if there are K non-terminals at the lowest level of a tree, then the number of trees that can be grown from this level equals the product of the number of trees which can be grown from each of the K non-terminals.

2.3 Analysis of Existing Grammar Metrics

Power and Malloy (2000 and 2004) state that the use of grammar metrics can facilitate the maintenance of grammar-based software applications. They apply established metrics in software engineering to Context-Free Grammars (CFGs). In the mapping process from software metrics to grammar metrics, procedures are considered as non-terminals and procedure bodies are the right-hand sides of production rules (Power and Malloy, 2000). The grammar metrics in (Power

2. An Algorithm for Computing the Average Branching Factor (Perplexity) of Speech-Recognition Grammars

and Malloy, 2000 and 2004) provide a measurement of Context-Free Grammars (CFGs) and facilitate the estimation of the difficulty of design, implementation, testing, and maintenance for large grammars from the point of view of software development.

The GCM (Grammar Confusability Metric) focuses on phrases with similar acoustic features in the grammar (Cai and Hamaker, 2008). GCM is used to flag the processing of two phrases that are acoustically alike but have different semantic meanings.

With the assistance of the concepts of “branching rate” and “the number of trees”, Miller and O’Sullivan (1992) investigate the growth of the derivations of grammars. “The number of trees” in (Miller and O’Sullivan, 1992) has the similar meaning to “branching factor” for a single decision point in our work. However, their approach differs from our ABF work in the following ways:

- (1) Miller and O’Sullivan specify the properties of generation level, ancestors, and the offspring of each node. We take into account the branching factors of all decision points of all the utterances in the language defined by the recognition grammar.
- (2) We obtain the value of the Average Branching Factor (ABF) of the recognition grammar. Miller and O’Sullivan are more interested in the growth with derivation depth at some specific level.
- (3) Our ABF algorithm is efficient. Miller and O’Sullivan’s approach is not practical for non-trivial grammars as it has exponential time complexity with respect to the depth of the derivation tree.

Note that all of the above grammar metrics are concerned with different properties of grammars. However, none of them gives a good indication of overall recognition accuracy. We will discuss what the ABF is and how it is related to recognition accuracy in the following sub-sections.

2.4 Introduction to the Average Branching Factor (ABF)

In grammar-directed speech recognition, the *branching factor* of a single decision point is the number of possible words to be considered as candidates at that point. During the recognition process, if at any point, it needs to examine the next symbol on the input to make a choice (even if the choice is a single branch), this point is a *decision point*. The *Average Branching Factor (ABF)* is the average of the branching factors of all decision points in all of the utterances in the language defined by the recognition grammar. The ABF is also called the *perplexity* of the language.

The Average Branching Factor (ABF) is important in predicting speech-recognition performance (National Research Council (U.S.), 1984). An increase of the ABF is likely to result in decreased performance. A smaller ABF indicates higher constraints and better recognition performance because the system has fewer choices to make (National Research Council (U.S.), 1984), (Hauptmann et. al., 1988), (Young et. al., 1989), and (Waibel and Lee, 1990).

Hauptmann et al. (1988) use various types of dialog-level knowledge to reduce the branching factor in a speech-recognition task to improve speech-recognition accuracy. Hauptmann et al. (1988), Young et al. (1989), and Waibel and Lee (1990) state that the ABF is a standard measure for determining the complexity (not computational complexity) of languages and a meaningful measure for speech-recognition systems. Young (1990) claims that a decrease in the Average Branching Factor b results in a decrease in the search space size s , for $s = b^d$, where d is the depth of the search space. The ABF is also important for determining the complexity of search algorithms (Edelkamp and Korf, 1998).

Experiments in (Morimoto and Takahashi, 2008, 2009) show that the ABF (Perplexity) directly affects speech-recognition accuracy, where the ABF decreases, the recognition accuracy is likely to improve.

2.5 Constraints on Speech-Recognition Grammars

The algorithm for computing the Average Branching Factor in this thesis requires that the grammars are proper, 1-lookahead, and non-recursive:

- (1) A context-free grammar is said to be *proper* if starting from the distinguished non-terminal, the only non-terminals produced are those whose further rewriting can eventually result in a string of terminals (Jelinek and Lafferty, 1991).
- (2) A *1-lookahead grammar* is one in which the director sets for each alternative in a production rule are disjoint. The *director set* of a rule is the set of terminals which start expressions that can be obtained by expansion of the rule. The cardinality of this set is the branching factor of that symbol. Consequently, a 1-lookahead grammar is deterministic and one in which the decision of which alternative to expand in a rule can be determined by looking at the next word on the input and matching it against a terminal in at-most one of the director sets of the alternatives.
- (3) A *non-recursive grammar* is one in which no non-terminal is defined in terms of itself, either directly in one production rule, or through mutual recursion involving more than one production rule. For example a rule of the form $\langle A \rangle = a \langle A \rangle b \mid c$; is not allowed.

The first requirement implies that all non-terminals must be properly defined, i.e., must appear on the left hand side with definition on the right hand side of a rule. This constraint is necessary for a grammar to be applied correctly in a speech application. The second constraint is not difficult to overcome as all context-free grammars can be converted to 1-lookahead grammars by a process of factoring (see for example Aho, Sethi and Ullman 1986). The last constraint is also not too significant in speech recognition owing to the fact that in the majority of applications there will be a limit on the length of the input utterances and on the depth of recursion. In many cases, it will be possible to easily rewrite the grammar so that iteration of syntactic constraints, which is implicit in recursive

2. An Algorithm for Computing the Average Branching Factor (Perplexity) of Speech-Recognition Grammars

grammars, is made explicit. For example, the recursive production rule $\langle A \rangle = a \langle A \rangle b \mid c$; with a limit of recursive depth 2 can be rewritten to $\langle A \rangle = c \mid a c b \mid a a c b b$;

2.6 Preliminary Discussion of the ABF Algorithm

Before presenting the algorithm for computing the Average Branching Factor from speech grammars, we first present a naïve and incorrect approach. Speech recognition grammars are context free grammars. Java Speech Grammar Format (JSGF) is the most common notation used to specify recognition grammars..JSGF is a platform-independent, vendor-independent textual representation of grammars for use in speech recognition (Sun, 2000). A summary of JSGF features is shown below in Table 2.6 (1). The formal definition of a CFG is given in sub-section 4.1. Figure 2.6.1 is a sample Context-Free Grammar (CFG), written in Java Speech Grammar Format (JSGF).

Table 2.6 (1): summary of JSGF features

Feature	Purpose
Word or “word”	Words (terminals, tokens) need not be quoted
$\langle \text{rule} \rangle$	Rule names (non-terminals) are enclosed in $\langle \rangle$
$[x]$	Optionally x
(\dots)	Grouping
$x y z \dots$	A sequence of x then y then z then ...
$x \mid y \mid z \dots$	A set of alternatives of x or y or z or ...
$\langle \text{rule} \rangle = x$; Public $\langle \text{rule} \rangle = x$;	A private and a public rule definition

2. An Algorithm for Computing the Average Branching Factor (Perplexity) of Speech-Recognition Grammars

$$\begin{aligned}
 \langle p \rangle &= \begin{array}{l} \langle q \rangle \langle q \rangle \langle b \rangle \\ | \quad \langle r \rangle \langle b \rangle ; \end{array} \\
 \langle b \rangle &= \begin{array}{l} 0 \\ | \quad 1 ; \end{array} \\
 \langle r \rangle &= \begin{array}{l} a \\ | \quad b \\ | \quad c ; \end{array} \\
 \langle q \rangle &= x \quad \langle r \rangle ;
 \end{aligned}$$

Figure 2.6.1: a sample CFG grammar

Example expressions defined by this grammar (Figure 2.6.1) are: $xaxa0$, $xaxa1$, $xaxb0$, $xaxb1$, $a0$, $a1$, $b0$, and $b1$.

First, we determine the director sets for each production rule in the grammar. Then we label each symbol with the director sets (in curly brackets) and the branching factors (with superscripts), as in Figure 2.6.2:

$$\begin{aligned}
 \langle p \rangle \{x, a, b, c\}^4 &= \begin{array}{l} \langle q \rangle^1 \quad \langle q \rangle^1 \quad \langle b \rangle^2 \\ | \quad \langle r \rangle^3 \quad \langle b \rangle^2 ; \end{array} \\
 \langle b \rangle \{0, 1\}^2 &= \begin{array}{l} 0 \\ | \quad 1 ; \end{array} \\
 \langle r \rangle \{a, b, c\}^3 &= \begin{array}{l} a \\ | \quad b \\ | \quad c ; \end{array} \\
 \langle q \rangle \{x\}^1 &= x \quad \langle r \rangle^3 ;
 \end{aligned}$$

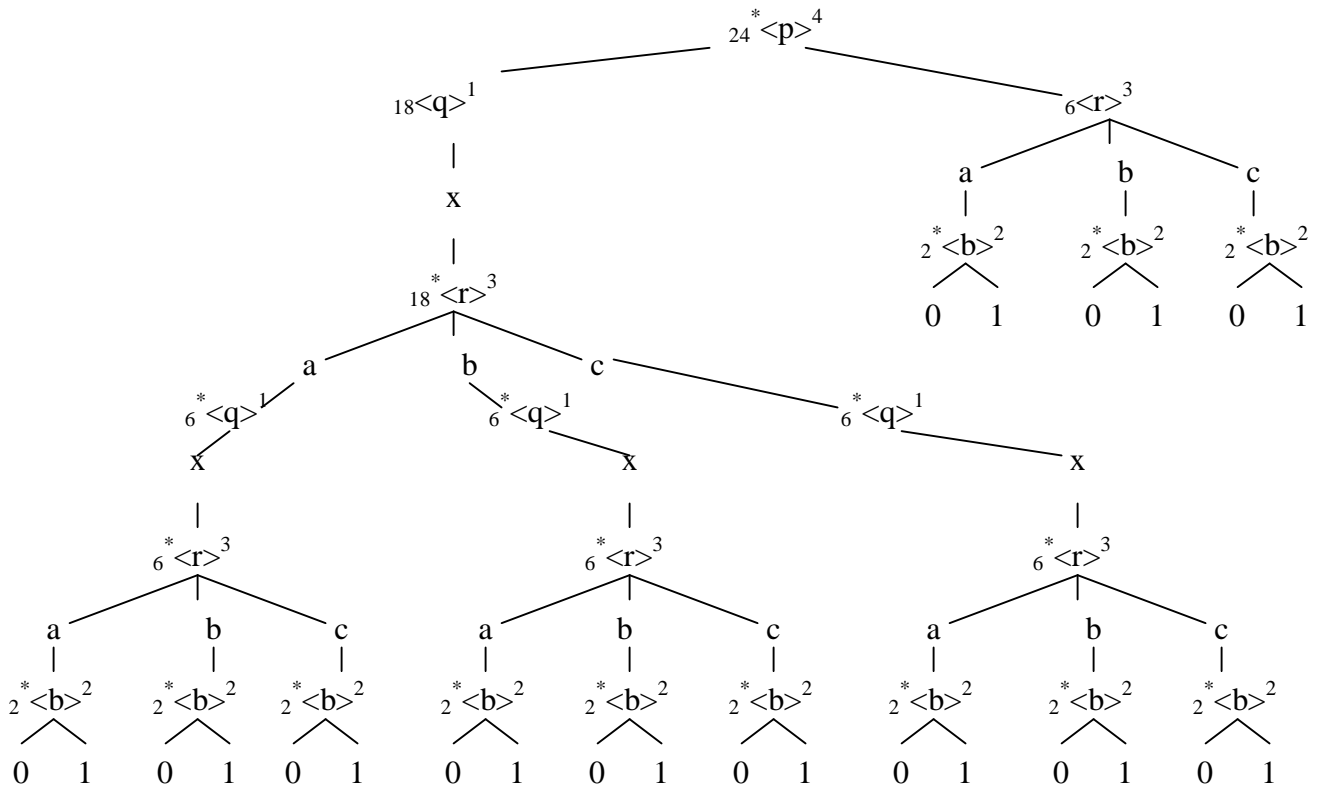
Figure 2.6.2: sample grammar with director sets and branching factors

To compute the ABF, one might be tempted to add the branching factors for all non-terminals on the right hand sides of the grammar rules and the start symbol, then divide the sum by the total number of non-terminals, as follows: $(4+1+1+2+3+2+3) / 7 \approx 2.3$. We will see this is incorrect.

2. An Algorithm for Computing the Average Branching Factor (Perplexity) of Speech-Recognition Grammars

There are a few reasons why the naïve approach gave an incorrect result. One is that it considers the number of times a symbol appears in the grammar rather than the number of times the symbol is expanded in deriving all sentences of the language. The second problem is that the naïve calculation counts all non-terminals in the grammar as decision points, which is not correct. For example, in the starting rule, while $\langle p \rangle$ denotes a decision point with 4 possible next terminals, the first $\langle q \rangle$ and the alternative $\langle r \rangle$ do not denote decision points as there are no more terminals to consider after the decision at $\langle p \rangle$ has been made.

Now, we consider another approach which is not practical, but gives us some insights. In this impractical approach, we compute the ABF from the tree that represents all derivations from the grammar (Figure 2.6.1). Note that we continue to use superscripts to denote the size of the director sets, i.e., branching factor. We also introduce subscripts to denote the number of utterances which contain that node. The derivation tree for the example grammar is shown in Figure 2.6.3.



2. An Algorithm for Computing the Average Branching Factor (Perplexity) of Speech-Recognition Grammars

Figure 2.6.3: derivation tree for the example grammar in Figure 2.6.1

The algorithm for calculating the ABF from the derivation tree is as follows:

Step 1: Expand the grammar to obtain the complete derivation tree. The start symbol is the root node in the tree. The set of alternatives of each symbol becomes a set of branches below that symbol. A sequence of symbols becomes a path in the tree.

Step 2: Add superscripts for branching factors for each node in the tree by examining the director sets. For example, the director set for node $\langle b \rangle$ is $\{0, 1\}$, so node $\langle b \rangle$ has a branching factor of 2.

Step 3: Add left-hand subscripts to denote the number of sentences which contain that node i.e. the size of the sub-language below the node. The left-hand subscripts are obtained in a bottom-up manner, starting from the bottom of the tree and working up.

Step 4: Identify decision nodes and mark them with *. For example, at the node $\langle b \rangle$, we need to make choice of terminal 0 or 1, so node $\langle b \rangle$ is a decision point. Note that the root $\langle p \rangle$ is a decision point, but the nodes $\langle q \rangle$ and $\langle r \rangle$ just below $\langle p \rangle$ are not decision points. The reason is that when we have made the decision at point $\langle p \rangle$, selecting a terminal from $\{x, a, b, c\}$, the decision has already been made for $\langle q \rangle$ and $\langle r \rangle$.

Step 5: Tabulate the values for decision points (Table 2.6(2)) with: branching factor, number of times occurring in the whole language, and the total number of branches involving the nodes. The total number of branches for a node is the product of the branching factor and the number of occurrences of the node in the language.

2. An Algorithm for Computing the Average Branching Factor (Perplexity) of Speech-Recognition Grammars

Table 2.6 (2): values for decision points in the derivation tree approach

Decision point	Branching factor	Number of occurrences	Total number of branches
<p>	4	24	96
<r>	3	18	54
	2	2	4
	2	2	4
	2	2	4
<q>	1	6	6
<q>	1	6	6
<q>	1	6	6
<r>	3	6	18
<r>	3	6	18
<r>	3	6	18
	2	2	4
	2	2	4
	2	2	4
	2	2	4
	2	2	4
	2	2	4
	2	2	4
	2	2	4
	2	2	4
Total		102	270

Step 6: Calculate the ABF as dividing the total number of branches of all decision points by the sum of the total number of occurrences of those decision points. The result for the above example is:

$$ABF = 270/102 = 2.65.$$

2. An Algorithm for Computing the Average Branching Factor (Perplexity) of Speech-Recognition Grammars

This method of calculating the ABF is straightforward, but is not practical for non-trivial grammars as it has exponential complexity with respect to the length of the utterances. However, it provides insight and justification for the new efficient algorithm given next.

2.7 The New ABF Algorithm

We note that the derivation-tree method involves “sweeps” through the tree from bottom to top and then top to bottom in which nodes are annotated with values representing properties used in calculating the ABF. We also note that the symbols from the grammar are repeated in the derivation tree and that we must find some way of combining the annotated values when labeling the grammar. Consideration of these factors leads to the following algorithm for computing the ABF directly from a grammar. We use the grammar from Figure 2.6.1 as example.

Step 1: Label each symbol of the grammar with right-hand superscripts denoting branching factors.

- 1) Each empty alternative has a branching factor of 0.
- 2) All terminals have a branching factor of 1.
- 3) A Left-Hand-Side (LHS) symbol of a rule has a branching factor obtained by summing the branching factors of the alternatives on the Right-Hand Side (RHS) of the rule.
- 4) The branching factor for a RHS non-terminal is the same as it appears on the LHS of its defining rule. For example:

2. An Algorithm for Computing the Average Branching Factor (Perplexity) of Speech-Recognition Grammars

$$\begin{aligned}
 \langle p \rangle^4 &= \begin{array}{l} \langle q \rangle^1 \quad \langle q \rangle^1 \quad \langle b \rangle^2 \\ | \quad \langle r \rangle^3 \quad \langle b \rangle^2 \quad ; \end{array} \\
 \langle b \rangle^2 &= \begin{array}{l} 0^1 \\ | \quad 1^1 \quad ; \end{array} \\
 \langle r \rangle^3 &= \begin{array}{l} a^1 \\ | \quad b^1 \\ | \quad c^1 \quad ; \end{array} \\
 \langle q \rangle^1 &= x^1 \quad \langle r \rangle^3 ;
 \end{aligned}$$

Figure 2.7.1: annotated example grammar with branching factors

Step 2: Label each symbol with a right-hand subscript denoting the size of the sub-language (subsize) obtained by expanding that symbol. The subsize of the start symbol gives the size of language defined by the grammar (Shi, 2003b).

- 1) Each empty alternative has a subsize of 0.
- 2) Each terminal has a subsize of 1.
- 3) The subsize of a LHS symbol is the sum of the subsizes of all its alternative sequences on the right hand side of the rule.
- 4) The subsize of a symbol in a sequence is the product of the subsizes of each symbol in the sequence.
- 5) The subsize of a RHS non-terminal is the same as it appears on the LHS of its definition rule. For example:

$$\begin{aligned}
 \langle p \rangle_{24}^4 &= \begin{array}{l} \langle q \rangle_3^1 \quad \langle q \rangle_3^1 \quad \langle b \rangle_2^2 \\ | \quad \langle r \rangle_3^3 \quad \langle b \rangle_2^2 \quad ; \end{array} \\
 \langle b \rangle_2^2 &= \begin{array}{l} 0_1^1 \\ | \quad 1_1^1 \quad ; \end{array} \\
 \langle r \rangle_3^3 &= \begin{array}{l} a_1^1 \\ | \quad b_1^1 \\ | \quad c_1^1 \quad ; \end{array} \\
 \langle q \rangle_3^1 &= x_1^1 \quad \langle r \rangle_3^3 ;
 \end{aligned}$$

Figure 2.7.2: annotated grammar with subsizes

2. An Algorithm for Computing the Average Branching Factor (Perplexity) of Speech-Recognition Grammars

Step 3: Label each symbol on the right hand side of each rule with a left-hand subscript which denotes the number of occurrences of that symbol in all derivations of expressions that are obtained by one expansion of the rule.

- 1) The number of occurrences of an empty alternative is 0.
- 2) All of the symbols of a sequence have the same number of occurrences, which is the product of the subsizes of each symbol in the sequence. Note that, if a sequence has only one symbol, the number of occurrences for this sequence is the subsize of that symbol. For example:

$$\begin{aligned}
 \langle p \rangle_{24}^4 &= \begin{array}{l} 18 \langle q \rangle_{3}^1 \quad 18 \langle q \rangle_{3}^1 \quad 18 \langle b \rangle_{2}^2 \\ | \\ 6 \langle r \rangle_{3}^3 \quad 6 \langle b \rangle_{2}^2 \end{array} ; \\
 \langle b \rangle_{2}^2 &= \begin{array}{l} 1 0_{1}^1 \\ | \\ 1 1_{1}^1 \end{array} ; \\
 \langle r \rangle_{3}^3 &= \begin{array}{l} 1 a_{1}^1 \\ | \\ 1 b_{1}^1 \\ | \\ 1 c_{1}^1 \end{array} ; \\
 \langle q \rangle_{3}^1 &= 3 x_{1}^1 \quad 3 \langle r \rangle_{3}^3 ;
 \end{aligned}$$

Figure 2.7.3: annotated grammar with the number of occurrences

Step 4: Starting with the start symbol of the grammar, label all symbols on the left hand side of the rules with a left-hand subscript in brackets denoting the total number of times the symbol occurs in derivations of all expressions in the language. Concurrently, label all symbols on the right hand side of rules with a left-hand subscript in brackets, and preceded with “*”, which denotes a “multiplication factor”.

- 1) The “multiplication factor” indicates the number of times the rule is used in different parts of the whole derivation tree. This multiplication factor is obtained by dividing the number of occurrences (left-lower subscript) of the symbol on the left hand side of the rule by its sub-language size of that symbol (lower right-hand subscript).

2. An Algorithm for Computing the Average Branching Factor (Perplexity) of Speech-Recognition Grammars

- 2) To obtain the value of the LHS bracketed subscripts for symbols on the left hand side of rules, we add together the values of left-hand subscripts of all instances of that symbol occurring on the RHS of all rules, each multiplied by the multiplication factor given in the associated bracketed subscript. For example:

$$\begin{aligned}
 (24) \quad <p>_{24}^4 &= \begin{array}{l} 18(*1) \quad <q>_{33}^1 \quad 18(*1) <q>_{33}^1 \quad 18(*1) _{22}^2 \\ | \quad 6(*1) \quad <r>_{33}^3 \quad 6(*1) \quad _{22}^2 \quad ; \end{array} \\
 (24) \quad _{22}^2 &= \begin{array}{l} 1(*12) \quad 0_{11}^1 \\ | \quad 1(*12) \quad 1_{11}^1 \quad ; \end{array} \\
 (42) \quad <r>_{33}^3 &= \begin{array}{l} 1(*14) \quad a_{11}^1 \\ | \quad 1(*14) \quad b_{11}^1 \\ | \quad 1(*14) \quad c_{11}^1 \quad ; \end{array} \\
 (36) \quad <q>_{33}^1 &= \quad 3(*12) \quad x_{11}^1 \quad 3(*12) <r>_{33}^3 \quad ;
 \end{aligned}$$

Figure 2.7.4: annotated grammar with bracketed left-hand subscripts

Step 5: Label the start symbol of the grammar, and all symbols on the RHS of all rules, except the leftmost alternative symbols, with an “*” superscript to indicate that they are decision points. Note that if an alternative consists of a single symbol, that symbol is not a decision point. For example:

$$\begin{aligned}
 (24) \quad * <p>_{24}^4 &= \begin{array}{l} 18(*1) <q>_{33}^1 \quad 18(*1) \quad * <q>_{33}^1 \quad 18(*1) \quad * _{22}^2 \\ | \quad 6(*1) \quad <r>_{33}^3 \quad 6(*1) \quad * _{22}^2 \quad ; \end{array} \\
 (24) \quad _{22}^2 &= \begin{array}{l} 1(*12) \quad 0_{11}^1 \\ | \quad 1(*12) \quad 1_{11}^1 \quad ; \end{array} \\
 (42) \quad <r>_{33}^3 &= \begin{array}{l} 1(*14) \quad a_{11}^1 \\ | \quad 1(*14) \quad b_{11}^1 \\ | \quad 1(*14) \quad c_{11}^1 \quad ; \end{array} \\
 (36) \quad <q>_{33}^1 &= \quad 3(*12) x_{11}^1 \quad 3(*12) \quad * <r>_{33}^3 \quad ;
 \end{aligned}$$

Figure 2.7.5: example grammar with decision-points

Step 6: Tabulate values for the decision points. The branching factors are the right-hand superscripts. The total number of times the decision point occurs is the

2. An Algorithm for Computing the Average Branching Factor (Perplexity) of Speech-Recognition Grammars

left-hand subscript multiplied by the multiplication factor given in brackets. The total number of branches for a decision point is the product of the branching factor and the number of the node's occurrences in the language. From Figure 2.7.5, we obtain Table 2.7.6.

Table 2.7.6: information from the annotated grammar

Decision Point	Branching Factor	Number of Occurrences	Total number of Branches
<p>	4	24	96
<q>	1	18	18
	2	18	36
	2	6	12
<r>	3	36	108
Total		102	270

Step 7: Calculate the ABF by summing the total number of branches and dividing by the total number of the occurrences in the language of decision points. From Table 2.7.6, the ABF is calculated as follows:

$$ABF = 270 / 102 = 2.65$$

2.8 More Examples

We include three more example grammars in this section. These example grammars are CFGs written in the Java Speech Grammar Format (JSGF). They are representatives because they cover the basic features of CFGs, such as rules, alternatives, symbol sequences, terminals, and non-terminals. Also, they are simple so that we can hand-trace them for the ABFs and present calculation details. We show each step for the ABF calculation below. We will further test these example grammars with the ABF implementation in sub-section 2.9 (grammars 1 – 3).

2. An Algorithm for Computing the Average Branching Factor (Perplexity) of Speech-Recognition Grammars

2.8.1 Example 1

$$\begin{aligned}
 \langle p \rangle &= \langle q \rangle \langle b \rangle \\
 &\quad | \quad \langle b \rangle ; \\
 \langle q \rangle &= a \langle s \rangle b \\
 &\quad | \quad x \langle s \rangle y ; \\
 \langle b \rangle &= 0 \quad | \quad 1 ; \\
 \langle s \rangle &= 2 \quad | \quad 3 ;
 \end{aligned}$$

Figure 2.8.1: example grammar 1

(1) calculate the Branching Factor (right-hand superscript):

$$\begin{aligned}
 \langle p \rangle^4 &= \langle q \rangle^2 \langle b \rangle^2 \\
 &\quad | \quad \langle b \rangle^2 ; \\
 \langle q \rangle^2 &= a^1 \langle s \rangle^2 b^1 \\
 &\quad | \quad x^1 \langle s \rangle^2 y^1 ; \\
 \langle b \rangle^2 &= 0^1 \quad | \quad 1^1 ; \\
 \langle s \rangle^2 &= 2^1 \quad | \quad 3^1 ;
 \end{aligned}$$

Figure 2.8.1: example grammar 1 (Step 1)

(2) calculate the subsize (right-hand subscript):

$$\begin{aligned}
 \langle p \rangle_{10}^4 &= \langle q \rangle_{4}^2 \langle b \rangle_{2}^2 \\
 &\quad | \quad \langle b \rangle_{2}^2 ; \\
 \langle q \rangle_{4}^2 &= a_{1}^1 \langle s \rangle_{2}^2 b_{1}^1 \\
 &\quad | \quad x_{1}^1 \langle s \rangle_{2}^2 y_{1}^1 ; \\
 \langle b \rangle_{2}^2 &= 0_{1}^1 \quad | \quad 1_{1}^1 ; \\
 \langle s \rangle_{2}^2 &= 2_{1}^1 \quad | \quad 3_{1}^1 ;
 \end{aligned}$$

Figure 2.8.1: example grammar 1 (Step 2)

2. An Algorithm for Computing the Average Branching Factor (Perplexity) of Speech-Recognition Grammars

(3) calculate the number of occurrences for RHS symbols (left-hand subscript):

$$\begin{aligned}
 \langle p \rangle_{10}^4 &= \begin{array}{c} 8 \langle q \rangle_4^2 \quad 8 \langle b \rangle_2^2 \\ | \quad 2 \langle b \rangle_2^2 ; \end{array} \\
 \langle q \rangle_4^2 &= \begin{array}{c} 2 a_{11}^1 \quad 2 \langle s \rangle_2^2 \quad 2 b_{11}^1 \\ | \quad 2 x_{11}^1 \quad 2 \langle s \rangle_2^2 \quad 2 y_{11}^1 ; \end{array} \\
 \langle b \rangle_2^2 &= \begin{array}{c} 1 0_{11}^1 \quad | \quad 1 1_{11}^1 ; \end{array} \\
 \langle s \rangle_2^2 &= \begin{array}{c} 1 2_{11}^1 \quad | \quad 1 3_{11}^1 ; \end{array}
 \end{aligned}$$

Figure 2.8.1: example grammar 1(Step 3)

(4) calculate the number of occurrences for LHS symbols (left-hand subscript in brackets) and multiplication factors for RHS symbols (left-hand subscript preceded with an “*”):

$$\begin{aligned}
 (10) \langle p \rangle_{10}^4 &= \begin{array}{c} 8(*) \langle q \rangle_4^2 \quad 8(*) \langle b \rangle_2^2 \\ | \quad 2(*) \langle b \rangle_2^2 ; \end{array} \\
 (8) \langle q \rangle_4^2 &= \begin{array}{c} 2(*) a_{11}^1 \quad 2(*) \langle s \rangle_2^2 \quad 2(*) b_{11}^1 \\ | \quad 2(*) x_{11}^1 \quad 2(*) \langle s \rangle_2^2 \quad 2(*) y_{11}^1 ; \end{array} \\
 (14) \langle b \rangle_2^2 &= \begin{array}{c} 1(*) 0_{11}^1 \quad | \quad 1(*) 1_{11}^1 ; \end{array} \\
 (8) \langle s \rangle_2^2 &= \begin{array}{c} 1(*) 2_{11}^1 \quad | \quad 1(*) 3_{11}^1 ; \end{array}
 \end{aligned}$$

Figure 2.8.1: example grammar 1(Step 4)

(5) label the decision points (left-hand “*” superscript):

$$\begin{aligned}
 (10) * \langle p \rangle_{10}^4 &= \begin{array}{c} 8(*) \langle q \rangle_4^2 \quad 8(*) * \langle b \rangle_2^2 \\ | \quad 2(*) \langle b \rangle_2^2 ; \end{array} \\
 (8) \langle q \rangle_4^2 &= \begin{array}{c} 2(*) a_{11}^1 \quad 2(*) * \langle s \rangle_2^2 \quad 2(*) * b_{11}^1 \\ | \quad 2(*) x_{11}^1 \quad 2(*) * \langle s \rangle_2^2 \quad 2(*) * y_{11}^1 ; \end{array} \\
 (14) \langle b \rangle_2^2 &= \begin{array}{c} 1(*) 0_{11}^1 \quad | \quad 1(*) 1_{11}^1 ; \end{array} \\
 (8) \langle s \rangle_2^2 &= \begin{array}{c} 1(*) 2_{11}^1 \quad | \quad 1(*) 3_{11}^1 ; \end{array}
 \end{aligned}$$

2. An Algorithm for Computing the Average Branching Factor (Perplexity) of Speech-Recognition Grammars

Figure 2.8.1: example grammar 1(Step 5)

(6) tabulate values for the decision points:

Table 2.8.1: information from Example grammar 1

Decision Point	Branching Factor	Number of Occurrences	Total number of Branches
<p>	4	10	40
	2	8	16
<s>	2	4	8
B	1	4	4
<s>	2	4	8
Y	1	4	4
Total		34	80

(7) Step 7, calculate the ABF:

$$ABF = 80 / 34 = 2.35$$

2.8.2 Example 2

<p> = <q> <s> <q>
 | ;

<s> =
 | c;

<q> = a
 | b ;

 = 0
 | 1;

Figure 2.8.2: example grammar 2

2. An Algorithm for Computing the Average Branching Factor (Perplexity) of Speech-Recognition Grammars

1) calculate the Branching Factor (right-hand superscript):

$$\begin{array}{l} \langle p \rangle^4 = \langle q \rangle^2 \langle s \rangle^3 \langle q \rangle^2 \\ | \quad \langle b \rangle^2; \end{array}$$

$$\begin{array}{l} \langle s \rangle^3 = \langle b \rangle^2 \\ | \quad c^1; \end{array}$$

$$\begin{array}{l} \langle q \rangle^2 = a^1 \langle b \rangle^2 \\ | \quad b^1 \langle b \rangle^2; \end{array}$$

$$\begin{array}{l} \langle b \rangle^2 = 0^1 \\ | \quad 1^1; \end{array}$$

Figure 2.8.2: example grammar 2 (Step 1)

2) calculate the subsize (right-hand subscript):

$$\begin{array}{l} \langle p \rangle_{50}^4 = \langle q \rangle_{40}^2 \langle s \rangle_{30}^3 \langle q \rangle_{40}^2 \\ | \quad \langle b \rangle_{20}^2; \end{array}$$

$$\begin{array}{l} \langle s \rangle_{30}^3 = \langle b \rangle_{20}^2 \\ | \quad c_{10}^1; \end{array}$$

$$\begin{array}{l} \langle q \rangle_{40}^2 = a_{10}^1 \langle b \rangle_{20}^2 \\ | \quad b_{10}^1 \langle b \rangle_{20}^2; \end{array}$$

$$\begin{array}{l} \langle b \rangle_{20}^2 = 0_{10}^1 \\ | \quad 1_{10}^1; \end{array}$$

Figure 2.8.2: example grammar 2 (Step 2)

2. An Algorithm for Computing the Average Branching Factor (Perplexity) of Speech-Recognition Grammars

- 3) calculate the number of occurrences for RHS symbols (left-hand subscript):

$$\begin{array}{l} \langle p \rangle_{50}^4 = 48 \langle q \rangle_4^2 \quad 48 \langle s \rangle_3^3 \quad 48 \langle q \rangle_4^2 \\ | \\ 2 \langle b \rangle_2^2 i \end{array}$$

$$\begin{array}{l} \langle s \rangle_3^3 = 2 \langle b \rangle_2^2 \\ | \\ 1 c_1^1 i \end{array}$$

$$\begin{array}{l} \langle q \rangle_4^2 = 2 a_1^1 \quad 2 \langle b \rangle_2^2 \\ | \\ 2 b_1^1 \quad 2 \langle b \rangle_2^2 i \end{array}$$

$$\begin{array}{l} \langle b \rangle_2^2 = 1 0_1^1 \\ | \\ 1 1_1^1 i \end{array}$$

Figure 2.8.2: example grammar 2 (Step 3)

- 4) calculate the number of occurrences for LHS symbols (left-hand subscript in brackets) and multiplication factors for RHS symbols (left-hand subscripts preceded with an “ * ”):

$$\begin{array}{l} (50) \langle p \rangle_{50}^4 = 48(*1) \langle q \rangle_4^2 \quad 48(*1) \langle s \rangle_3^3 \quad 48(*1) \langle q \rangle_4^2 \\ | \\ 2(*1) \langle b \rangle_2^2 i \end{array}$$

$$\begin{array}{l} (48) \langle s \rangle_3^3 = 2(*16) \langle b \rangle_2^2 \\ | \\ 1(*16) c_1^1 i \end{array}$$

$$\begin{array}{l} (96) \langle q \rangle_4^2 = 2(*24) a_1^1 \quad 2(*24) b_1^1 \\ | \\ 2(*24) b_1^1 \quad 2(*24) \langle b \rangle_2^2 i \end{array}$$

$$\begin{array}{l} (130) \langle b \rangle_2^2 = 1(*65) 0_1^1 \\ | \\ 1(*65) 1_1^1 i \end{array}$$

Figure 2.8.2: example grammar 2 (Step 4)

2. An Algorithm for Computing the Average Branching Factor (Perplexity) of Speech-Recognition Grammars

5) label the decision points (left-hand “*” superscript):

$$\begin{aligned}
 (50) \quad {}^* <p>^4_{50} &= \begin{array}{l} 48(^*1) \quad <q>^2_4 \quad 48(^*1) \quad {}^* <s>^3_3 \quad 48(^*1) \quad {}^* <q>^2_4 \\ | \\ 2(^*1) \quad ^2_2; \end{array} \\
 (48) \quad <s>^3_3 &= \begin{array}{l} 2(^*16) \quad ^2_2 \\ | \\ 1(^*16) \quad c^1_1; \end{array} \\
 (96) \quad <q>^2_4 &= \begin{array}{l} 2(^*24) \quad a^1_1 \quad 2(^*24) \quad {}^* ^2_2 \\ | \\ 2(^*24) \quad b^1_1 \quad 2(^*24) \quad {}^* ^2_2; \end{array} \\
 (130) \quad ^2_2 &= \begin{array}{l} 1(^*65) \quad 0^1_1 \\ | \\ 1(^*65) \quad 1^1_1; \end{array}
 \end{aligned}$$

Figure 2.8.2: example grammar 2 (Step 5)

6) tabulate values for the decision points:

Table 2.8.2: information from Example grammar 2

Decision Point	Branching Factor	Number of Occurrences	Total number of Branches
<p>	4	50	200
<s>	3	48	144
<q>	2	48	96
	2	48	96
	2	48	96
Total		242	632

7) calculate the ABF:

$$ABF = 632 / 242 = 2.61$$

2. An Algorithm for Computing the Average Branching Factor (Perplexity) of Speech-Recognition Grammars

2.8.3 Example 3

$$\begin{aligned}
 \langle bs \rangle &= \langle b \rangle \quad \langle b1 \rangle ; \\
 \langle b1 \rangle &= \text{empty} \\
 &\quad | \quad \langle b \rangle \quad \langle b2 \rangle ; \\
 \langle b2 \rangle &= \text{empty} \\
 &\quad | \quad \langle b \rangle ; \\
 \langle b \rangle &= 0 \\
 &\quad | \quad 1 ;
 \end{aligned}$$

Figure 2.8.3: example grammar 3

1) calculate the Branching Factor (right-hand superscript):

$$\begin{aligned}
 \langle bs \rangle^2 &= \langle b \rangle^2 \quad \langle b1 \rangle^2 ; \\
 \langle b1 \rangle^2 &= \text{empty}^0 \\
 &\quad | \quad \langle b \rangle^2 \quad \langle b2 \rangle^2 ; \\
 \langle b2 \rangle^2 &= \text{empty}^0 \\
 &\quad | \quad \langle b \rangle^2 ; \\
 \langle b \rangle^2 &= 0^1 \\
 &\quad | \quad 1^1 ;
 \end{aligned}$$

Figure 2.8.3: example grammar 3 (Step 1)

2) calculate the subsize (right-hand subscript):

$$\begin{aligned}
 \langle bs \rangle^2_2 &= \langle b \rangle^2_2 \quad \langle b1 \rangle^2_2 ; \\
 \langle b1 \rangle^2_4 &= \text{empty}^0_0 \\
 &\quad | \quad \langle b \rangle^2_2 \quad \langle b2 \rangle^2_2 ; \\
 \langle b2 \rangle^2_2 &= \text{empty}^0_0 \\
 &\quad | \quad \langle b \rangle^2_2 ; \\
 \langle b \rangle^2_2 &= 0^1_1 \\
 &\quad | \quad 1^1_1 ;
 \end{aligned}$$

2. An Algorithm for Computing the Average Branching Factor (Perplexity) of Speech-Recognition Grammars

Figure 2.8.3: example grammar 3 (Step 2)

- 3) calculate the number of occurrences for RHS symbols (left-hand subscript):

$$\langle bs \rangle_2^2 = {}_4\langle b \rangle_2^2 \quad {}_4\langle b1 \rangle_2^2 ;$$

$$\begin{array}{l} \langle b1 \rangle_4^2 = {}_0empty_0^0 \\ | \quad {}_4\langle b \rangle_2^2 \quad {}_4\langle b2 \rangle_2^2 ; \end{array}$$

$$\begin{array}{l} \langle b2 \rangle_2^2 = {}_0empty_0^0 \\ | \quad {}_2\langle b \rangle_2^2 ; \end{array}$$

$$\langle b \rangle_2^2 = {}_10_1^1 \quad | \quad {}_11_1^1 ;$$

Figure 2.8.3: example grammar 3 (Step 3)

- 4) calculate the number of occurrences for LHS symbols (left-hand subscript in brackets) and multiplication factors for RHS symbols (left-hand subscript preceded with an “*”):

$$({}_4)\langle bs \rangle_2^2 = {}_4(*_1)\langle b \rangle_2^2 \quad {}_4(*_1)\langle b1 \rangle_2^2 ;$$

$$\begin{array}{l} ({}_4)\langle b1 \rangle_4^2 = {}_0empty_0^0 \\ | \quad {}_4(*_1)\langle b \rangle_2^2 \quad {}_4(*_1)\langle b2 \rangle_2^2 ; \end{array}$$

$$\begin{array}{l} ({}_4)\langle b2 \rangle_2^2 = {}_0empty_0^0 \\ | \quad {}_2(*_2)\langle b \rangle_2^2 ; \end{array}$$

$$({}_{12})\langle b \rangle_2^2 = {}_1(*_6)0_1^1 \quad | \quad {}_1(*_6)1_1^1 ;$$

Figure 2.8.3: example grammar 3 (Step 4)

2. An Algorithm for Computing the Average Branching Factor (Perplexity) of Speech-Recognition Grammars

5) label the decision points (left-hand “*”superscript):

$$\begin{aligned}
 {}_{(4)} * <bs>^2_2 &= {}_{4(*1)} ^2_2 \quad {}_{4(*1)} * <b1>^2_2 ; \\
 {}_{(4)} <b1>^2_4 &= {}_0 \text{empty}^0_0 \\
 &\quad | \quad {}_{4(*1)} ^2_2 \quad {}_{4(*1)} * <b2>^2_2 ; \\
 {}_{(4)} <b2>^2_2 &= {}_0 \text{empty}^0_0 \\
 &\quad | \quad {}_{2(*2)} ^2_2 ; \\
 {}_{(12)} ^2_2 &= {}_{1(*6)} 0^1_1 \quad | \quad {}_{1(*6)} 1^1_1 ;
 \end{aligned}$$

Figure 2.8.3: example grammar 3 (Step 5)

6) tabulate values for the decision points:

Table 2.8.3: information from Example grammar 3

Decision Point	Branching Factor	Number of Occurrences	Total number of Branches
<bs>	2	4	8
<b1>	2	4	8
<b2>	2	4	8
Total		12	24

7) calculate the ABF:

$$ABF = 24 / 12 = 2$$

2.9 Implementation of the ABF Algorithm

The implementation of the ABF algorithm includes three phases: 1) preprocessing, 2) algorithm application, and 3) post-processing. In the preprocessing phase, the program reads in the grammar file, tokenizes the grammar symbols, and generates the required lists for the next phase. The lists generated in phase one include `grammar`, `isTerminal`, `isRHS`, `isAlternative`, `ruleNo`, `sequence`, and `isDecPoint`. Details about the lists are in sub-section 3.2.

2. An Algorithm for Computing the Average Branching Factor (Perplexity) of Speech-Recognition Grammars

With the preliminary information obtained from the preprocessing, the second phase of the algorithm is able to further obtain the required information for calculating the ABF, such as the branching factors and subsizes for each symbol of the grammar, the number of occurrences for the RHS and LHS symbols, and the multiplication factors for the RHS symbols of the grammar. Phase two implements the first four steps of the algorithm ABF.

With the information obtained from phase one and phase two, phase three calculates and outputs the ABF by summing the total number of branches for decision points and dividing by the total number of occurrences for decision points in the language, which are the last three steps of the algorithm ABF. Meanwhile, the program outputs the ABF and other related grammar metrics, such as the number of rules of the grammar, the number of symbols in the grammar, the number of terminals of the grammar, the number of non-terminals of the grammar, the number of decision points of the grammar, and the size of the language defined by the grammar.

We test the ABF algorithm with three groups of nine grammars. The grammars are as follows:

- (1) Group one is a set of simple CFG grammars (Grammar 0-3) for testing. Grammar 0, 1, 2, and 3 are the example grammars in Figure 2.6.1, Figure 2.8.1, Figure 2.8.2, and Figure 2.8.3, respectively. These grammars are fed to the ABF algorithm application and are hand-traced to calculate the ABFs in order to informally illustrate how the algorithm computes the correct answers. Note that this does not prove correctness of the algorithm nor does it show the absence of errors in the implementation of the algorithm.
- (2) Group two includes a set of small practical grammars, i.e., the *Read-A-Book* grammar (Grammar 4, shown in Appendix C) for the speech

2. An Algorithm for Computing the Average Branching Factor (Perplexity) of Speech-Recognition Grammars

application `Read-A-Book` and its variants (Grammar 5, shown in Appendix D). These two grammars share the same vocabulary.

- (3) Group three includes two more-complicated practical grammars and one word-sequence grammar all with the same domain. Grammar 6 is a Semantic Constraint Grammar (SCG) (in Figure 4.3.2) and Grammar 7 is a Context Free Grammar (CFG) (in Figure 4.3.1). Grammar 8 (in Appendix E) is a word sequence grammar, which accepts word sequences from one word to ten words from the vocabulary. The SCG is the most restricted grammar which directly encodes semantic constraints in the syntax. The word sequence grammar is the most relaxed grammar.

The results of applying the ABF application on the above grammars are shown in Table 2.9.

Note that “# of terminal” in the following and all other tables in the dissertation means “number of instances of a terminal symbol in the grammar” and is a measure of the size of the grammar and not the vocabulary of the language.

2. An Algorithm for Computing the Average Branching Factor (Perplexity) of Speech-Recognition Grammars

Table 2.9: results of applying the ABF algorithm

No.	Grammar	# of Rules	# of Symbols	# of Non- Terminals	# of Terminals	# of Decision Points	Language Size	ABF
0	Simple Grammar-Figure 2.2.1	4	16	10	6	5	24	2.65
1	Simple Grammar-Figure 2.8.1	4	17	9	8	6	10	2.35
2	Simple Grammar-Figure 2.8.2	4	16	11	5	5	50	2.61
3	Simple grammar-Figure 2.8.3	4	13	9	4	3	4	2
4	Read a Book-Appendix C	16	338	37	301	13	$5.38 \cdot 10^8$	44.51
5	Word sequence-Appendix D	2	245	7	238	5	$7.6 \cdot 10^{11}$	238
6	SCG - Figure 4.3.2	41	262	133	129	53	$1.51 \cdot 10^9$	33.99
7	CFG - Figure 4.3.1	17	160	50	110	19	$1.73 \cdot 10^{11}$	52.42
8	Word sequence-Appendix E	12	184	77	107	46	$9.14 \cdot 10^{19}$	188.99

2. An Algorithm for Computing the Average Branching Factor (Perplexity) of Speech-Recognition Grammars

The results are summarized as follows:

1. The results from Group one (Grammars 0 – 3) shows that the ABF implementation program obtains the same results as those from hand-tracing.
2. As expected, with the same domain, the more restricted grammar (Grammar 6, SCG) defines a language with a smaller ABF. The most relaxed grammars (word sequence, e.g., Grammar 5 and Grammar 8) have the largest ABF with the same vocabulary.

In reality, it is costly and time-consuming to design an experiment and arrange subjects to test the performance of a grammar in a speech application. Therefore, before undertaking an experiment with test cases and subjects, we can primarily evaluate the grammar with the ABF metric and other metrics generated by the ABF application. With the same domain, the grammar with a smaller ABF is more likely to be useful as it is likely to have better recognition accuracy.

2.10 Summary

We have presented, what would appear to be, the first algorithm to compute the Average Branching Factor (ABF) of a language from the grammar that defines that language. The motivation for this work was to provide a method for more-easily calculating properties which are useful when designing speech-recognition grammars.

We began by discussing the need for grammar metrics, then reviewed and analyzed existing grammar metrics. We also referred to research by others who claim that the Average Branching Factor (ABF) is a good indicator of speech-recognition accuracy.

We then described our new algorithm for calculating the ABF. We began by describing a naïve but incorrect algorithm and analyzed the reason why it was incorrect. Then, we illustrated an intuitive method by using a derivation tree to

2. An Algorithm for Computing the Average Branching Factor (Perplexity) of Speech-Recognition Grammars

obtain the Average Branching Factor (ABF). This method gives the correct result, but it is impractical for the exponential complexity with respect to the length of the sentences. Next, we introduced the seven-step ABF algorithm and the implementation for this algorithm. The proofs of termination, correctness, and complexity of the algorithm are presented in section 3.

It has been claimed by other researchers that the ABF is a good indicator for speech-recognition accuracy. We believe that the novel algorithm for computing the ABF may be useful for effective and high-quality grammar design and analysis.

3. PROOFS OF THE ABF ALGORITHM

To facilitate the proofs of termination, correctness, and complexity, we have coded the ABF algorithm in Miranda, a declarative non-strict purely functional programming language. A brief introduction to Miranda derived from (Turner, 1986) is given in sub-section 3.1. The Miranda source code is given in sub-section 3.2.

3.1 Introduction to Miranda

Miranda is a non-strict purely functional programming language. A Miranda program is a collection of equations defining various functions and data structures. The order of the equations is not significant. For example,

```
sq x = x*x
```

is a function to calculate the square of the parameter x .

An equation can have several alternative right hand sides distinguished by “guards” on the right following a comma. For example, the function to return the bigger of two numbers can be written as follows:

```
max a b = a, if a >= b
        = b, if a < b
```

The last guard can be written as `otherwise`, instead of using the `if` condition.

Miranda’s evaluation mechanism is “lazy”, in the sense that no sub-expression is evaluated until its value is required.

The most commonly used data structure in Miranda is the *list*, written with square brackets and commas. For example,

```
week_days = ["Mon", "Tue", "Wed", "Thur", "Fri"]
```

3. Proofs of the ABF Algorithm

The elements of a list must be all of the same type. A sequence of elements of mixed types can be represented as a `tuple`, written using parentheses instead of square brackets, e.g.

```
student = ("tom", "computer science", 93)
```

Miranda is strongly typed. This means that, any inconsistency in the type structure of an expression or a sub-expression will result in a compile-time error message. There are three primitive types in Miranda, namely `num`, `bool`, and `char`. The type `num` consists of integer and floating point numbers. The type `bool` has two values, `True` and `False`. The type `char` comprises the ASCII character set.

List comprehensions give a concise syntax for a general class of iterations over lists. The general form of a list comprehension is as follows:

```
[body | qualifiers]
```

Note that two or more qualifiers are separated by semicolons.

An example list comprehension is as follows:

```
[n*n | n <- [1..100] ]
```

which is a list containing (in order) the squares of all the numbers from 1 to 100.

The following are some operators that are used in Miranda programming:

(1) `++` appends two lists. E.g.,

```
[1,2] ++ [3,4] = [1,2,3,4]
```

(2) `:` prefixes an element to the front of a list. E.g.,

```
1:[2,3,4] = [1,2,3,4]
```

(3) # gets the length of a list. E.g.,

$$\#[1, 2, 3, 4] = 4$$

(4) ! does subscripting. Index starts from 0. E.g.,

$$[1, 2, 3, 4] !1 = 2$$

(5) .. , a shorthand notation for lists whose elements form an arithmetic series.

E.g. $[1 \dots 100]$ is the list of 100 elements from 1 to 100.

(6) + - * / , plus, minus, times, division.

(7) > >= = ~= <= < , comparison operators.

(8) & , logical and .

(9) \ / , logical or .

(10) ~ , logical negation.

(11) | | , denotes comments.

3.2 Miranda Code for the ABF Algorithm

We have implemented the ABF algorithm using several Miranda functions. The intermediate results are represented in lists. For example, the branching factors for each symbol of the grammar are obtained in step one of the ABF algorithm (section 2.7). In the algorithm implementation, a list of branching factors for the symbols of the grammar is generated by the component function `bf`. This list of branching factors is used for later calculation of the Average Branching Factor (ABF).

The lists needed for the ABF algorithm implementation are as follow:

(1) List `grammar`. The input grammar is expressed as a list of tuples

$[(x, k)]$, where x is a symbol of the grammar, k is the index of the symbol in the grammar list starting from 0, $0 \leq k \leq (\#grammar - 1)$. For the example grammar in Figure 2.6.1, the `grammar` list is as follows:

```
grammar = [("<p>", 0), ("<q>", 1), ("<q>", 2), ("<b>", 3),  
           ("<r>", 4), ("<b>", 5), ("<b>", 6), ("0", 7),  
           ("1", 8), ("<r>", 9), ("a", 10), ("b", 11),  
           ("c", 12), ("<q>", 13), ("x", 14), ("<r>", 15)]
```

(2) `isTerminal` is a list of Boolean values with the same length as list `grammar`. Its value reflects whether the corresponding symbol in the grammar is a terminal or a non-terminal. If the symbol in the grammar is a terminal, the corresponding value in list `isTerminal` is `True`; otherwise, it is `False`. For the example grammar in Figure 2.6.1, the list `isTerminal` is as follows:

```
isTerminal = [False, False, False, False, False, False,  
              False, True, True, False, True, True, True,  
              False, True, False]
```

(3) `isRHS` is a list of Boolean values with the same length as list `grammar`. Its value reflects whether the corresponding symbol in the grammar is a Right-Hand-Side symbol or not. If the symbol in the grammar is on the right hand side of the rule, the corresponding value in `isRHS` is `True`; otherwise, it is `False`. For the example grammar in Figure 2.6.1, the list `isRHS` is as follows:

```
isRHS = [False, True, True, True, True, True, False, True,  
          True, False, True, True, True, False, True, True]
```


- (4) `isAlternative` is a list of Boolean values with the same length as list `grammar`. Its value reflects whether the corresponding symbol in the grammar is a left-most alternative in the rule. If the symbol in the grammar is the left-most alternative, the corresponding value of `isAlternative` is `True`; otherwise, it is `False`. For the example grammar in Figure 2.6.1, the list `isAlternative` is as follows:

```
isAlternative = [False, True, False, False, True, False,
                False, True, True, False, True, True, True,
                False, True, False]
```

- (5) `ruleNo` is a list of numbers with the same length as list `grammar`. It records the rule number of the corresponding symbol in the grammar. Rule numbers start from 1. For the example grammar in Figure 2.6.1, the list `ruleNo` is as follows:

```
ruleNo = [1, 1, 1, 1, 1, 1, 2, 2, 2, 3, 3, 3, 3, 4, 4, 4]
```

- (6) List `sequence` denotes sequence-related information of the symbols in the grammar. Each symbol of the grammar is associated with a number denoting its sequence-related information. A left-hand-side non-terminal is associated with the number showing the number of sequences on the right hand side of the rule. A right-hand-side symbol is associated with the number showing on which sequence it is. The list `sequence` for the example grammar in Figure 2.6.1 is as follows:

```
sequence = [2, 1, 1, 1, 2, 2, 2, 1, 2, 3, 1, 2, 3, 1, 1, 1]
```

- (7) `isDecPoint` is a list of Boolean values with the same length as list `grammar`. Its value reflects whether the corresponding symbol in the grammar is a decision point or not. If the symbol in the grammar is a decision point, the corresponding value of `isDecPoint` is `True`;

3. Proofs of the ABF Algorithm

otherwise, it is `False`. For the example grammar in Figure 2.6.1, the list `isDecPoint` is as follows:

```
isDecPoint = [True,False,True,True,False,True,False,
              False,False,False,False,False,False,
              False,False,True]
```

These lists can be obtained in a pre-process (phase one), which reads in the grammar file, tokenizes the grammar, analyzes each symbol, and composes the above lists according to their definitions. Then the second and third phases implement the ABF algorithm. The Miranda code for the ABF algorithm is given below in Figure 3.2:

3. Proofs of the ABF Algorithm

```
| | -----
| | -- Step 1: calculate the Branching Factor
| | -- INPUT:  a list of the symbols of the grammar,
| | --          grammar[[char], num)].
| | -- OUTPUT: a list of branching factor for each symbol of
| | --          the grammar, bfList[num].
| | -- Type of the program is:
| | --          bf :: =>  [[char], num)] -> [num]
| | -----

1. bf [] = [0]
   || an empty alternative has a branching factor of 0

2. bf [(x,k)] = [1], if (isTerminal !k) & (isRHS !k)
   || RHS terminals have a branching factor of 1

3. bf [(x,k)] = bf [(y, j) | (y,j) <- grammar;
                           y = x;
                           ~(isTerminal ! j);
                           ~(isRHS ! j)],
   if (isRHS ! k) & ~(isTerminal ! k)
   || a RHS non-terminal has the same branching factor
   || as it appears on the LHS of its definition rule.

4.  bf [(x,k)] = [sumList (bf [ (z, h)
                           |(z, h) <- grammar;
                           (isRHS ! h);
                           (ruleNo ! h) = (ruleNo ! k);
                           (isAlternative ! h))] ],
   if ~(isRHS ! k) & ~(isTerminal ! k)
   || A LHS non-terminal obtains its branching factor by
   || summing the branching factors of the left-most
   || alternatives on the Right-Hand Side of the rule.

5. bf ((a,b): as) = bf ([(a,b)]) ++ bf as
   || calculate the branching factors for each symbol of
   || the list

6. sumList = foldr (+) 0
   || calculate the sum of a list

7. bfList = bf grammar
   || calculate the branching factors for all the symbols
   || of the grammar
```

3. Proofs of the ABF Algorithm

```
| | -----
| | -- Step 2: calculate the Subsize
| | -- INPUT:  a list of the symbols of the grammar,
| | --          grammar[[char], num)].
| | -- OUTPUT: a list of subsize for each symbol of the
| | --          grammar, subsizeList[num].
| | -- Type of the program is:
| | --          subsize :: =>  [[char], num)] -> [num]
| | -----

8. subsize [] = [0]
   | | an empty alternative has a subsize of 0

9. subsize [(x,k)] = [1], if (isTerminal !k) & (isRHS !k)
   | | RHS terminals have a subsize of 1

10. subsize [(x,k)] = subsize [(y, j) | (y,j) <- grammar;
   | |                               y = x;
   | |                               ~(isTerminal ! j);
   | |                               ~(isRHS ! j)],
   | |                               if (isRHS ! k) & ~(isTerminal ! k)
   | | | the subsize of a RHS non-terminal is the same as it
   | | | appears on the LHS of its definition rule.

11. subsize [(x,k)] = [sumList [(productList (subsize [(z,h)
   | | (z,h) <- grammar;
   | | (isRHS !h);
   | | (ruleNo !h)=ruleNo !k;
   | | (sequence !h = s))] )
   | | | s <- [1 .. (sequence !k)] ] ],
   | | | if ~(isRHS ! k) & ~(isTerminal ! k)
   | | | The subsize of a LHS symbol is the sum of the
   | | | subsizes of all its alternative sequences on the
   | | | right hand side of the rule.

12. subsize ((a,b): as) = subsize [(a,b)] ++ subsize as
   | | calculate the subsizes for each symbol of the list

13. productList = foldr (*) 1
   | | calculate the product of a list

14. subsizeList = subsize grammar
   | | calculate the subsizes for all the symbols
   | | of the grammar
```

3. Proofs of the ABF Algorithm

```
| | -----
| | -- Step 3: calculate the number of occurrences for each
| | --           symbol on RHS of the rules
| | -- INPUT:  a list of the symbols of the grammar,
| | --           grammar[([char], num)].
| | -- OUTPUT: a list of the number of occurrences for each
| | --           symbol of the grammar, occur_rhsList[num].
| | -- Type of the program is:
| | --           occur_rhs :: =>  [([char], num)] -> [num]
| | -----

15.  occur_rhs [] = [0]
    | | The number of occurrences of an empty alternative
    | | is 0.

16.  occur_rhs [(x,k)] = [productList [subsizeList !h
    | h <- [0 .. (#subsizeList -1)];
    | (isRHS !h) ;
    | sequence !h = sequence !k;
    | ruleNo !h = ruleNo !k ]],
    | if (isRHS !k)
    | = [0], otherwise

    | | All of the symbols of a RHS sequence have the same
    | | number of occurrences, which is the product of
    | | the subsizes of each symbol in the sequence.

17.  occur_rhs ((a,b): as)
    = occur_rhs [(a,b)] ++ occur_rhs as
    | | calculate the number of occurrences for each
    | | symbol on RHS of the rules

18.  occur_rhsList = occur_rhs grammar
    | | calculate the number of occurrences for each
    | | symbol on RHS of the rules of the grammar
```

3. Proofs of the ABF Algorithm

```
| | -----
| | -- step 4: calculate the number of occurrences for each
| | --           symbol on the LHS of the rules,
| | --           and calculate the multiplication factor for the
| | --           symbols on the RHS of the rules.
| | -- INPUT: a list of the symbols of the grammar,
| | --           grammar[[char], num]].
| | -- OUTPUT: a list of number of occurrences for each symbol
| | --           of the grammar, occur_lhsList[num],
| | --           and a list of multiplication factor for each
| | --           symbol of the grammar, factorList[num].
| | -- Type of the program is:
| | --           occur_lhs :: => [[char], num] -> [num]
| | --           factor    :: => [[char], num] -> [num]
| | -----

19.   occur_lhs [] = [0]
      || For an empty list, this value is 0

20.   occur_lhs [(x,k)]
      = [subsizeList !k],
        if ~(isRHS !k) & ~(isTerminal !k) & ruleNo !k = 1
      || the number of occurrences for start symbol is
      || its subsize

21.   occur_lhs [(x,k)] = [sumList [(occur_rhsList !h) *
                                   ((factor [(y, h)]) !0)
                               | (y,h) <- grammar;
                               y = x;
                               isRHS !h ] ],
        if ~(isRHS !k) & ~(isTerminal !k)
          & ruleNo !k > 1
      || the number of occurrences for a LHS symbol is
      || obtained by adding together the number of
      || occurrences of that symbol occurring on the RHS of
      || all rules, each multiplied by the multiplication
      || factor.

22.   occur_lhs [(x,k)] = [0],           if (isRHS !k)
      || for a RHS symbol, this value is 0

23.   occur_lhs ((a,b): as) = occur_lhs [(a,b)] ++
                               occur_lhs as
      || calculate the number of occurrences for each
      || symbol
```

3. Proofs of the ABF Algorithm

```
||-----
24.   factor [] = [0]
      || For an empty list, this value is 0

25.   factor[(y,h)] =
      [(occur_lhs [(y,h)])!0 / (subsizeList !h)],
      if ~(isRHS !h) & ~(isTerminal !h)
      || the multiplication factor for a LHS symbol is
      || obtained by dividing the number of occurrences of
      || this symbol on the LHS of the rule by its subsize
      || of that symbol.

26.   factor[(y,h)] = factor [(z,j) | (z,j) <- grammar;
      ruleNo !j = ruleNo !h;
      ~(isRHS !j);
      ~(isTerminal !j) ],
      if (isRHS !h)
      || The multiplication factor for a RHS symbol is the
      || same as that for the LHS non-terminal of the rule.

27.   factor ((a,b): as) = factor [(a,b)] ++ factor as
      || calculate the multiplication factor for each
      || symbol

28.   occur_lhsList = occur_lhs grammar
      || calculate the number of occurrences for each
      || symbol of the grammar

29.   factorList = factor grammar
      || calculate the multiplication factor for each
      || symbol of the grammar

||-----
|| -- STEP 5. recognize the decision points of the grammar
|| -- Recognize the start symbol of the grammar, and all
|| -- symbols on the RHS alternatives of all rules, except
|| -- the leftmost symbols as decision points.
|| -- this is done in pre-process.
|| -- The example list is as follows:
|| --
|| -- isDecPoint = [True, False, True, True, False, True,
|| -- False, False, False, False, False, False, False,
|| -- False, True]
||-----
```

3. Proofs of the ABF Algorithm

```

| | -----
| | -- STEP 6. calculate the number of occurrences and the
| | --           total number of branches
| | -----
30.   num_occurList = [(occur_rhsList !i) * (factorList ! i)
                      +(occur_lhsList !i) * (factorList ! i)
                      | i <- [0 .. (#grammar - 1)]]

| | calculate the total number of occurrences a symbol
| | occurs for all symbols of the grammar.

| | The total number of times a symbol occurs is the
| | number of occurrences for RHS/ LHS symbol multiplied
| | by the multiplication factor.

| | Note that, for a RHS symbol, occur_lhsList!i = 0,
| |               for a LHS symbol, occur_rhsList!i = 0.
| | so the above formula can calculate the number of
| | occurrences for each symbol of the grammar.
| | this list has the same length as the list grammar.

31.   num_occurList_Dec = [num_occurList ! i
                          | i <- [0 .. (#grammar - 1)];
                          (isDecPoint ! i) ]
| | a list of the number of occurrences, for only
| | decision points.

32.   total_branchList_Dec = [num_occurList!i * bfList!i
                             | i <- [0 .. (#grammar - 1)];
                             (isDecPoint ! i)]

| | The total number of branches for a decision point is
| | the product of the branching factor and the number
| | of the node's occurrences in the language.

| | -----
| | -- STEP 7. calculate ABF
| | -----

33.   abf = (sumList total_branchList_Dec) /
            (sumList num_occurList_Dec)
| | calculate the ABF by summing the total number of
| | branches and dividing by the total number of
| | decision points' occurrences in the language.
```

Figure 3.2: Miranda source code for the ABF algorithm

3.3 Proof of Termination

In a pure functional programming language such as Miranda, the only form of iteration is through recursion. Therefore, to prove termination it is only necessary to prove that all recursive descents are well founded. The standard method for doing this is called a “size-change” termination proof. The idea is to find a measure of the recursive function argument size that decreases (or increases) on each recursive call such that, after a finite number of recursive calls, it reaches a terminating value (i.e., the base case of the definition of the recursive function). The following proof of termination of the BNF algorithm is a collection of size-change termination proofs for all recursive calls in the BNF algorithm.

(It should be noted that in a “lazy” functional programming language such as Miranda, arguments to functions are not evaluated unless required, and are then only evaluated to the extent required. For example `[1..]!3` returns a value of 4, even though the argument to `!` is the infinite list `[1..]`. The last evaluation process only evaluates the first four values of the list (note that the index starts at 0). This means that in a lazy language the number of terminating programs is larger than the set of programs whose termination can be proven using size-change proofs. Sereni (2006) has developed a termination proof technique for programs whose termination depends on lazy evaluation. However, the BNF algorithm does not rely on lazy evaluation and we show below that its termination can be proven using size-change proof and does not need the more complex proof technique developed by Sereni.)

The algorithm ABF consists of seven steps that are executed in sequence. So, if each step of the algorithm (i.e., component function in the program, Figure 3.2) terminates, the algorithm ABF terminates.

(1) Step 1 (component function bf)

There are three recursive calls in step 1, i.e., lines 3, 4, and 5. Line 7 runs the program to obtain the list of branching factors (i.e., bfList) for all the symbols of the grammar.

- 1) Lines 1, 2, 3, and 4 calculate the branching factor for one symbol of the grammar. Lines 1 and 2 are base cases for the component function bf , which return constant values and terminate.
- 2) Line 3 tests to see if the current symbol is a RHS non-terminal, the component function bf will return the branching factor of the same symbol appearing on the LHS of a grammar rule, which evokes line 4.
- 3) Line 4 calculates the branching factor for a LHS non-terminal by summing all the branching factors for the RHS leftmost-alternatives of the current rule. If there are some RHS non-terminals involving in the sum, it will recursively call line 3.
- 4) Line 5 deals with the list (i.e., grammar) by processing elements (symbols) one by one. In each round of recursion, one symbol is manipulated, and the length of the list decreases by 1 until it reaches 0. So, if the process for one element (symbol) terminates, the component function bf (step 1) will terminate when all of the symbols of the grammar have been processed.

Recursion occurs in lines 3, 4, and 5. Line 5 depends on the termination of lines 3 and 4. Lines 3 and 4 are mutually recursive calls. Since the grammar is non-recursive (sub-section 2.5), each recursive call in line 4 will call for a different symbol using an index which increases in a well-founded sequence (until it reaches a point where there are no more symbols). Since there are a finite number of symbols (terminals/ non-terminals) in the grammar, the algorithm will finally finish the traversal of non-terminal symbols and reach terminal symbols (line 2) and terminate. Therefore, step 1 will terminate.

(2) Step 2 (component function `subsizeList`)

Step 2 includes three recursive calls, lines 10, 11, and 12. Line 14 runs the program to obtain the list of subsize (i.e., `subsizeList`) for all the symbols of the grammar.

- 1) Lines 8, 9, 10, and 11 calculate the subsize for one symbol of the grammar. Lines 8 and 9 are base cases for the component function `subsize`, which return constant values and terminate.
- 2) Line 10 tests to see if the current symbol is a RHS non-terminal, the component function `subsize` will return the subsize of the same symbol appearing on the LHS of the grammar, which evokes line 11.
- 3) Line 11 calculates the subsize for a LHS non-terminal by summing the subsizes for all the sequences of the current rule. If there are some RHS non-terminals involving in the sum, it will recursively call line 10.
- 4) Line 12 deals with the list (i.e., `grammar`) by processing elements (symbols) one by one. In each round of recursion, one symbol is manipulated, and the length of the list decreases by 1 until reaches 0. Therefore, if the process for one element (symbol) terminates, component function `subsize` (step 2) will terminate when all the symbols of the grammar have been processed.

Recursion occurs in lines 10, 11, and 12. Line 12 depends on the termination of lines 10 and 11. Lines 10 and 11 are mutually recursive calls. Since the grammar is non-recursive (section 2.5), each recursive call at line 11 will call a different symbol with an index which increases in a well-founded sequence (until it reaches a point where there are no more symbols). Since there are a finite number of symbols (terminals/ non-terminals) in the grammar, the algorithm will finally finish the traverse of non-terminal symbols and come to terminal symbols (line 9) and terminate. Therefore, step 2 will terminate.

(3) Step 3 (component function `occur_rhs`)

Recursion in step 3 occurs at line 17, which deals with the list `grammar` by processing elements (symbols) one by one. In each round of the recursion, one symbol is manipulated and the length of the list decreases by 1 until reaches 0. Therefore, if the process for one element (symbols) terminates, component function `occur_rhs` (step 3) will terminate when all the symbols of the grammar have been processed.

Lines 15 and 16 calculate the number of occurrences for one symbol of the grammar. Line 18 runs the program to obtain the list of the number of occurrences of the RHS symbols for the symbols of the grammar (`occur_rhsList`).

- 1) Line 15 is the base case of the component function `occur_rhs`, which returns a constant value and terminates.
- 2) Line 16 searches the `grammar` list for the symbols in the same sequence as the current RHS symbol and calculates the product of the subsizes of the symbols in the same sequence. The lengths of the `grammar` list and the sequences are finite. Therefore, the search will terminate while finishing every symbol in `grammar` list and line 16 will terminate. Note that line 16 skips the cases for LHS symbols.

Therefore, the component function `occur_rhs` (step 3) will terminate.

(4) Step 4 (component functions `occur_lhs` and `factor`)

Step 4 includes two component functions, i.e., `occur_lhs` and `factor`. Function `occur_lhs` calculates the number of occurrences for each symbol on the left hand side of the rules. Function `factor` calculates the multiplication factors for the symbols on the right hand side of rules. These two component

functions may mutually recursive call each other. Recursion occurs at lines 21, 23, 25, 26, and 27. Lines 28 and 29 run the programs to obtain the list of the number of occurrences of the LHS symbols (`occur_lhsList`) and the multiplication factors for the symbols of the grammar (`factorList`). They will terminate if the component functions terminate, which we now prove.

Lines 23 and 27 deal with the list `grammar` by processing elements (symbols) one by one. In each round of recursion, one symbol is manipulated and the length of the list decreases by 1 until reaches 0. So, if the process for one element (symbol) terminates, the algorithm terminates.

Lines 19, 20, 21, and 22 calculate the number of occurrences for one symbol of the grammar. Lines 24, 25, and 26 calculate the multiplication factor for one symbol of the grammar.

- 1) Lines 19 and 20 are base cases for the component function `occur_lhsList`, which return constant values and terminate. Note that, line 20 returns the subsize of the LHS symbol, which is already available in step 2.
- 2) Line 21 tests if the current symbol is a LHS non-terminal and not of the first rule, the algorithm will search the `grammar` list for all occurrences of this non-terminal appearing on the right hand side of the grammar. Then, calculate the product of its number of RHS symbol occurrences and the multiplication factor. The sum of the product is returned as the number of the LHS symbol occurrences. Note that in line 21, the numbers of RHS symbol occurrences are already available in step 3. So if the operation of calculating the multiplication factor terminates, the algorithm of calculating the number of LHS symbol occurrences will terminate.
- 3) Lines 24, 25, 26, and 27 calculate the multiplication factors.
- 4) Line 24 is a base case for the component function `factor`, which returns a constant value and terminates.

- 5) Line 25 tests to see if the current symbol is a LHS non-terminal, its multiplication factor is the number of its LHS symbol occurrences divided by its subsize (already available in step 2). Note that, for the first rule, the subsize and the number of the LHS symbol occurrences are available and the same. So the multiplication factor for LHS symbol of the first rule is available, i.e., 1. Meanwhile the multiplication factors for the RHS symbols for the first rule are also available by line 26, which are the same as that of the LHS symbol, i.e., 1.
- 6) Recursion in function `factor` occurs in lines 25 and 26. Since the grammar is non-recursive (section 2.5), each recursion will call for a different symbol with an index which increases in a well-founded sequence (until it reaches a point where there are no more symbols). Since there are a finite number of symbols of the grammar, the algorithm will finally come to the symbols in the first rule and terminate (line 20).

Therefore, the component functions for computing the number of occurrences for LHS symbols (`occur_lhs`), and the multiplication factors for RHS symbols (`factor`) will terminate. Step 4 will terminate.

(5) Step 5

Step 5 goes through all the symbols of the grammar and labels the decision points. There are a finite number of symbols in the grammar. Therefore, step 5 will terminate with the last symbol in the grammar.

(6) Step 6

Step 6 tabulates the values for the decision points. It will terminate with a finite number of decision points.

(7) Step 7

Step 7 performs a simple arithmetic calculation for the Average Branching Factor with the above information and will terminate.

In summary, each step of the ABF algorithm will terminate. Therefore, the algorithm of calculating the Average Branching Factor will terminate.

3.4 Proof of Correctness

We use Structural Induction to prove the correctness of each step of the ABF algorithm. Information regarding the grammar is represented in lists (sub-section 3.2). The length of each list is the number of symbols in the grammar. With reference to the Miranda code in section 3.2, we present the proofs for each step as follows.

(1) Proof of correctness for step 1 (component function `bf`, lines 1 – 7):

Base Case: there is one rule in the grammar with one non-terminal on the Left Hand Side and one terminal on the Right Hand Side of the rule, e.g.,

$$\langle g \rangle = "a"$$

In this case, the list `grammar` is represented as follows:

$$\text{grammar} = [(\langle g \rangle, 0), ("a", 1)]$$

3. Proofs of the ABF Algorithm

Therefore:

```
bf grammar
= bf[("<g>", 0)]++ bf [("a", 1)]           (line 5)

= [sumList(bf [("a", 1)] ) ]++bf[("a", 1)](line 4)

= [sumList [1]] ++ [1]                     (line 2)

= [1] ++ [1]                             (line 6)

= [1,1]                                   (definition of "++")
```

Therefore, for the base case, the algorithm correctly calculates the branching factors for each symbol of the grammar.

Inductive step:

Hypothesis: for a grammar with n symbols, (i.e., $\#grammar = n$), the algorithm correctly calculates the branching factors for each symbol of the grammar, i.e., $bfList$, is obtained successfully,

$$bfList = bf \text{ grammar}$$

Show:

1. Adding one more symbol, x , the algorithm correctly calculates the branching factors for each symbol of the grammar, i.e., the new list of branching factors for all symbols of the grammar will be obtained successfully,

$$bfList_{new} = bf (grammar ++ [(x, n)])$$

Note that, the index for a list starts from 0. For an n -item list, the index for the last symbol is $(n-1)$. Therefore, the index for the newly-added symbol x is n .

There are two cases in the inductive step:

Case 1: the newly-added symbol is a terminal, " x ", on the Right-Hand-Side of

the rule. Since the index of the list starts with 0, the index of the newly-added terminal, "x", will be n .

There are two cases while adding a new terminal, "x":

- 1) The newly-added terminal "x" is added to an existing symbol sequences on the right hand side of the rule. Note that, this newly-added symbol will not affect the branching factors of other existing symbols.

Therefore:

```
bfList_new
= bf (grammar ++ [("x", n)])

= bf grammar ++ bf [("x", n)]
                                (line 5 & definition of "++" and ":")

= bfList ++ bf [("x", n)]      (hypothesis)

= bfList ++ [1]                (line 2)
```

Since `bfList` is a list including the correct branching factors for the n symbols of the grammar (by the hypothesis), list `bfList++[1]` includes correct branching factors for the n symbols and the newly-added terminal "x", whose branching factor is 1.

- 2) The newly-added terminal "x" is a new alternative of the right hand side of the rule.

In this case, the newly-added symbol will affect the branching factor of the LHS symbol (e.g., `lhs_symb`) in the current rule. The branching factors for other symbols of the grammar will not change. If the branching factors for the affected LHS symbol (`lhs_symb`) and the newly-added symbol are calculated correctly, the algorithm correctly calculates the branching factors for all the symbols of the grammar.

3. Proofs of the ABF Algorithm

Suppose the index for the affected LHS symbol (`lhs_symb`) is k , where $0 \leq k < n$, and the rest of the grammar symbols are in a sublist `grammar1`.

Therefore:

```
bfList_new
= bf (grammar ++ [("x", n)])

= bf (grammar_1 ++ [(lhs_symb,k)] ++ [("x", n)])

= bf grammar_1 ++ bf [(lhs_symb,k)] ++ bf [("x", n)]
      (line 5 & definition of "++" and ":")

= [c] ++ bf [(lhs_symb,k)] ++ bf [("x", n)]
      ([c] is a list of previously computed values,
       which is the list of branching factors for
       list grammar1.

      By the hypothesis, all symbols of the grammar
      have obtained correct branching factors. And
      the branching factors for the sublist
      grammar1 are not affected by the newly-added
      symbol and the list [c] is available.)

= [c] ++ bf [(lhs_symb,k)] ++ [1]                                (line 2)

= [c] ++ [sumList (bf [(q, t)])] ++ [1]                          (line 4)
      ([ (q, t) ] represents the list of the left-most
       alternatives including the newly-added
       symbol.)

= [c] ++ [f] ++ [1]
      (Note that, the branching factor for the
       newly-added alternative is available (i.e.,
       1), and the other left-most alternatives have
       all obtained the correct branching factors
       (by the hypothesis). Therefore, the sum of
       these values are available. Let it be  $f$ .)
```

We have seen that the lists `[c]`, `[f]`, and `[1]` include the correct branching factors for the n symbols of the grammar and the newly-added alternative. List `[c]++[f]++[1]` includes correct branching factors for the symbols and the new alternative.

3. Proofs of the ABF Algorithm

Therefore, by adding one more terminal on the Right-Hand-Side of the rule, the algorithm correctly calculates the branching factors for each symbol of the grammar.

Case 2: the newly-added symbol is a non-terminal, $\langle x \rangle$, on the Right-Hand-Side of the rule. Since the index of the list starts with 0, the index of the newly-added non-terminal $\langle x \rangle$ will be n .

Note that, this newly-added non-terminal must be a symbol that has been defined in the grammar. By the restrictions for the algorithm (section 2.5), the grammar must be proper, which means all the non-terminals must be defined (appearing on the left hand side of the rule). So the sole newly-added non-terminal must be a non-terminal which is already in the grammar.

There are two cases while adding a new non-terminal:

- 1) The newly-added symbol $\langle x \rangle$ is added to an existing symbol sequences on the right hand side of the rule. Note that, this newly-added symbol will not affect the branching factors of other existing symbols.

3. Proofs of the ABF Algorithm

Therefore:

```
bfList_new
= bf (grammar ++ [("<x>", n)])

= bf grammar ++ bf [("<x>", n)]
      (line 5 & definition of "++" and ":")

= bfList ++ bf [("<x>", n)]
      (by the hypothesis)

= bfList ++ bf [("<y>", m)]
      (line 3)(where, <x>=<y>, <y> is a LHS
      non-terminal in the grammar list)

= bfList ++ [c]
      (c has been calculated correctly from the hypothesis.
      By the hypothesis, the symbols in the list
      grammar have all obtained the correct branching
      factors. Without the loss of generality, we can
      let bf [("<y>", m)] = [c], where c is a correct
      value. )
```

Since `bfList` is a list including the correct branching factors for the n symbols of the grammar (by the hypothesis) and `c` has also been calculated correctly from the hypothesis, list `bfList++[c]` includes correct branching factors for the n symbols and the newly-added non-terminal `<x>`.

2) The newly-added non-terminal `<x>` is a new alternative of the right hand side of the rule.

In this case, the newly-added symbol will affect the branching factor of the LHS symbol (e.g., `lhs_symb`) in the current rule. The branching factors for other symbols of the grammar will not change. If the branching factors for the affected LHS symbol (`lhs_symb`) and the newly-added symbol are calculated correctly, the algorithm correctly calculates the branching factors for all the symbols of the grammar.

3. Proofs of the ABF Algorithm

Suppose the index for the affected LHS symbol (`lhs_symb`) is k , where $0 \leq k < n$, and the rest of the grammar symbols are in a sublist `grammar1`.

Therefore:

```
bfList_new
= bf (grammar ++ [("<x>", n)])

= bf (grammar_1 ++ [(lhs_symb,k)] ++ [("<x>", n)])

= bf grammar_1 ++ bf [(lhs_symb,k)] ++ bf [("<x>", n)]
    (line 5 & definition of "++" and ":")

= [c] ++ bf [(lhs_symb,k)] ++ bf [("<x>", n)]
    ([c] is a list of previously computed values,
    which is the list of branching factors for
    sublist grammar1.
    By the hypothesis, all symbols of the grammar
    have obtained correct branching factors. And
    the branching factors for the sublist
    grammar1 are not affected by the newly-added
    symbol, let it be list [c].)

= [c] ++ bf [(lhs_symb,k)] ++ bf [("<y>", m)] (line 3)
    (where,  $\langle x \rangle = \langle y \rangle$ ,  $\langle y \rangle$  is a LHS non-terminal in
    the grammar list)

= [c] ++ bf [(lhs_symb,k)] ++ [e]
    (e has been calculated correctly from the
    hypothesis)
    (By the hypothesis, the symbols in the list
    grammar have all obtained the correct
    branching factors. Without the loss of
    generality, we can let  $\text{bf}[("<y>", m)] = [e]$ ,
    where  $e$  is a correct value. )

= [c] ++ [sumList (bf [(q, t)])] ++ [e] (line 4)
    ([ (q, t) ] represents the list of the left-most
    alternatives on the right hand side of the
    rule, including the newly-added non-terminal.)

= [c] ++ [f] ++ [e]
    (Note that, the branching factor for the
    newly-added alternative is available (i.e.,
    [e]), and the other left-most alternatives
```

3. Proofs of the ABF Algorithm

have all obtained the correct branching factors (by the hypothesis). Therefore, the sum of these values is available. Let it be f .)

We have seen that the lists $[c]$, $[f]$, and $[e]$ include the correct branching factors for the n symbols of the grammar and the newly-added alternative. List $[c]++[f]++[e]$ includes correct branching factors for the symbols of the grammar and the new alternative.

So, by adding one more non-terminal on the Right-Hand-Side of the rule, the algorithm correctly calculates the branching factors for each symbol of the grammar.

2. Add a new rule to the grammar, the algorithm correctly calculates the branching factors for each symbol of the grammar, i.e., the new list of branching factors for all symbols of the grammar will be obtained successfully,

$$bfList_new = bf (grammar ++ newRule)$$

There are three cases in the inductive step:

Case 1: the new rule has a non-terminal on the left hand side, and a terminal on the right hand side, i.e.,

$$\langle new_r \rangle = "x"$$

Since the index of a list starts from 0, the indexes for $\langle new_r \rangle$ and $"x"$ will be n and $(n+1)$ respectively. The list for this new rule is represented as:

$$newRule = [(\langle new_r \rangle, n), ("x", (n+1))]$$

3. Proofs of the ABF Algorithm

Therefore:

```
bfList_new
= bf (grammar ++ newRule )

= bf grammar ++ bf newRule
      (line 5 & definition of "++" and ":")

= bfList ++ bf [("<new_r>", n), ("x", (n+1))] (hypothesis)

= bfList++ bf [("<new_r>", n)]++ bf [("x", (n+1))]
      (line 5 & definition of "++" and ":")

= bfList ++ bf [("<new_r>", n)] ++ [1]          (line 2)

= bfList ++ [sumList (bf [("x", (n+1))])] ++ [1] (line 4)

= bfList ++ [sumList [1]] ++ [1]                (line 2)

= bfList ++ [1] ++ [1]                          (line 6)

= bfList ++ [1, 1]                             (definition of "++")
```

Since `bfList` is a list including the correct branching factors for the n symbols of the grammar (by the hypothesis), list `bfList++[1,1]` includes correct branching factors for the n symbols and the newly-added rule.

Therefore, by adding a new rule with one non-terminal on the left hand side and one terminal on the right hand side, the algorithm correctly calculates the branching factors for each symbol of the grammar.

Case 2: The new rule includes terminal(s) and/or known non-terminal(s) on the right hand side, i.e.,

$$\text{<new_r>} = x$$

where `<new_r>` is a known non-terminal, and `x` is an expression of sequence(s) and/or alternative(s) with known non-terminal(s) and/or terminal(s). Since the index for a list starts from 0, the indexes for `<new_r>` and `x` will be n and $(n+k)$, $k \geq 1$, respectively. The list for this new rule is represented as:

3. Proofs of the ABF Algorithm

```
newRule = [("<new_r>", n), (x, (n+k))]
```

Therefore:

```
bfList_new
= bf (grammar ++ newRule)

= bf grammar ++ bf newRule
      (line 5 & definition of "++" and ":")

= bfList ++ bf [("<new_r>", n), (x, (n+k))]      (hypothesis)

= bfList ++ bf [("<new_r>", n)] ++ bf [(x, (n+k))]
      (line 5 & definition of "++" and ":")

= bfList ++ bf [("<y>", m)] ++ bf [(z, j)]      (line 3)
  (where, <new_r>=<y>, <y> is a known non-terminal)
  (x=z, z is a known non-terminal or terminal)

= bfList ++ [c] ++ [d]
  (c, d are correct values by the hypothesis)
  (By the hypothesis, the symbols in the list grammar
   have all obtained the correct branching factors,
   and the terminals have the branching factor of 1
   (line 2)), so we can obtain the branching factors,
   bf [("<y>", m)] and bf [(z, j)], say values [c]
   and [d])
```

Since bfList is a list including the correct branching factors for the n symbols of the grammar (by the hypothesis) and values c and d are correct branching factors, list bfList ++[c]++[d] includes correct branching factors for the n symbols and the newly-added rule.

Therefore, by adding a new rule with all known non-terminal(s) and/or terminal(s), the algorithm correctly calculates the branching factors for each symbol of the grammar.

Case 3: The new rule includes a new non-terminal on the left hand side of the new rule, and known non-terminal(s) and/ or terminal(s) on the right-hand side of the new rule, i.e.,

```
<new_r> = x
```


3. Proofs of the ABF Algorithm

where $\langle \text{new_r} \rangle$ is a new non-terminal, and x is an expression of sequence(s) and/or alternative(s) with known non-terminal(s) and/or terminal(s). Since the index for a list starts from 0, the indexes for $\langle \text{new_r} \rangle$ and x will be n and $(n+k)$, $k \geq 1$, respectively. The list for this new rule is represented as:

$$\text{newRule} = [(\langle \text{new_r} \rangle, n), (x, (n+k))]$$

Note that, the non-terminals on the right hand side of the newly-added rule cannot be new because the grammar is “proper” (section 2.5), which means every non-terminal has to be defined properly (has to appear on the left hand side of the rule). So it is not allowed to include any new non-terminal on the right hand side of the newly-added rule without an accompanying definition for it.

Therefore:

```
bfList_new
= bf (grammar ++ newRule)

= bf grammar ++ bf newRule
      (line 5 & definition of “++” and “:”)

= bfList ++ bf[(“<new_r>”, n),(x, (n+k))]      (hypothesis)

= bfList ++ bf [(“<new_r>”, n)] ++ bf [(x, (n+k))]
      (line 5 & definition of “++” and “:”)

= bfList ++ bf [(“<new_r>”, n)] ++ bf [(z, j)]      (line 3)
  (where z = x, z is a known non-terminal or a
   terminal)

= bfList ++ bf [(“<new_r>”, n)] ++ [d]
  (d is a correct previously computed value , by the
   hypothesis)
  (By the hypothesis, the symbols in the list
   grammar have all obtained the correct branching
   factors, and the terminals have a branching
   factor of 1 (line 2)), so we can obtain the
   branching factor, bf [(z, j)], let it be called
   [d].)
```

3. Proofs of the ABF Algorithm

```
= bfList ++ [sumList (bf [(q, t)])] ++ [d]          (line 4)
    ([ (q, t) ] is the list of the left-most
    alternatives among the list [(z, j)] with known
    branching factors)
```

```
= bfList ++ [sumList [c]] ++ [d]                    (line 3)
    (c] is a list of previously computed values,
    which is the list of branching factors for list
    [(q, t)])
```

```
= bfList ++ [e] ++ [d]                              (line 6)
                                     (e is the sum of the list [c])
```

Since `bfList` is a list including the correct branching factors for the n symbols of the grammar (by the hypothesis) and the value `e` and `d` are correct branching factors, list `bfList++[e]++[d]` includes correct branching factors for the n symbols and the newly-added rule.

Therefore, by adding a new rule with a new non-terminal on the left hand side and known non-terminal(s) and/or terminal(s) on the right hand side of the rule, the algorithm correctly calculates the branching factors for each symbol of the grammar.

(2) Proof of correctness for step 2 (component function `subsize`, lines 8 – 14):

Base Case: there is one rule in the grammar with one non-terminal on the Left Hand Side and one terminal on the Right Hand Side of the rule, e.g.,

$$\langle g \rangle = "a"$$

In this case, the list `grammar` is represented as follows:

$$\text{grammar} = [(\langle g \rangle, 0), ("a", 1)]$$

3. Proofs of the ABF Algorithm

Therefore:

```
subsize grammar
= subsize [("<g>", 0)] ++ subsize [("a", 1)]      (line 12)

= [sumList [productList (subsize [("a", 1)])] ] ++
  subsize [("a", 1)]                             (line 11)

= [sumList [productList (subsize [("a", 1)])]] ++ [1]
                                           (line 9)

= [sumList [1]] ++ [1]                         (line 13)

= [1] ++ [1]                                  (definition of sumList)

= [1, 1]                                       (definition of "++")
```

For the base case, the algorithm correctly calculates the subsizes for each symbol of the grammar.

Inductive Step:

Hypothesis: for a grammar with n symbols, (i.e., $\#grammar = n$), the algorithm correctly calculates the subsizes for each symbol of the grammar, i.e., $subsizeList[num]$ is obtained successfully,

```
subsizeList = subsize grammar
```

Show:

1. Adding one more symbol, x , the algorithm correctly calculates the subsizes for each symbol of the grammar, i.e., the new list of subsizes for all symbols of the grammar will be obtained successfully,

```
subsizeList_new = subsize (grammar ++ [(x, n)])
```

There are two cases in the inductive step:

Case 1: the newly-added symbol is a terminal, " x ", on the Right Hand Side of

3. Proofs of the ABF Algorithm

the rule. Since the index of the list starts with 0, the index of the newly-added terminal, "x", will be n .

There are two cases while adding a new terminal, "x":

1) The newly-added terminal "x" is added to an existing symbol sequence on the right hand side of the rule. Note that, this newly-added symbol will affect the subsize of the LHS symbol (e.g., `lhs_symb`) of the current rule and will not affect the subsizes of other symbols. If the subsizes for the affected LHS symbol (`lhs_symb`) and the newly-added symbol are calculated correctly, the algorithm correctly calculates the subsizes for all the symbols of the grammar.

Suppose the index for the affected LHS symbol (`lhs_symb`) is k , where $0 \leq k < n$, and the rest of the grammar symbols are in a sublist `grammar1`.

Therefore:

```
subsizeList_new
= subsize (grammar ++ [("x", n)])

= subsize (grammar1 ++ ["<lhs_symb>", k] ++ [("x", n)])

= subsize grammar1 ++ subsize ["<lhs_symb>", k] ++
  subsize [("x", n)])
      (line 12 & definition of "++" and ":")

= [c] ++ subsize ["<lhs_symb>", k] ++ subsize [("x", n)])
  ([c] is a list of previously computed values,
   which is the list of subsizes for list
   grammar1.
   By the hypothesis, all symbols of the grammar
   have obtained correct subsizes. And the
   subsizes for the sublist grammar1 are not
   affected by the newly-added symbol.)

= [c] ++ subsize ["<lhs_symb>", k] ++ [1]          (line 9)

= [c] ++ [sumList [productList (subsize [(y, j)])]] ++ [1]
      (line 11)
  (list [y,j] represents the alternative sequences
```

3. Proofs of the ABF Algorithm

of the rule that the new symbol is added to, (including the sequence with the new symbol). The algorithm calculates the product of the subsizes of the sequence symbols, and adds the products for all alternatives of the rule together to obtain the subsize of the LHS symbol.)

= [c] ++ [d] ++ [1]

(by the hypothesis, the subsizes for the sequence symbols are correctly calculated. In addition, the subsize for the newly-added symbol is available (e.g., [1]). Therefore, the subsize for the LHS symbol is available. Let it be [d].)

Lists [c], [d], and [1] include the correct subsizes for the n symbols of the grammar and the newly-added symbol. List [c]++[d]++[1] includes correct subsizes for the symbols of the grammar and the new symbol.

Therefore, by adding a new terminal to an existing symbol sequence on the right hand side of the rule, the algorithm correctly calculates the subsizes for each symbol of the grammar.

2) The newly-added terminal "x" is a new alternative of the right hand side of the rule.

In this case, the newly-added symbol will affect the subsize of the LHS symbol (e.g., lhs_symb) in the current rule and will not change the subsizes for other symbols of the grammar. If the subsizes for the affected LHS symbol (lhs_symb) and the newly-added symbol are calculated correctly, the algorithm correctly calculates the subsizes for all the symbols of the grammar.

Suppose the index for the affected LHS symbol (lhs_symb) is k , where $0 \leq k < n$, and the rest of the grammar symbols are in a sublist grammar1.

Therefore:

3. Proofs of the ABF Algorithm

```
subsizeList_new
= subsize (grammar ++ [("x", n)])

= subsize (grammar1 ++ ["<lhs_symb>", k] ++ [("x", n)])

= subsize grammar1 ++ subsize ["<lhs_symb>", k] ++
  subsize [("x", n)])
      (line 12 & definition of "++" and ":")
= [c] ++ subsize ["<lhs_symb>", k] ++ subsize [("x", n)])
      ([c] is a list of previously computed values,
       which is the list of subsizes for list
       grammar1.
       By the hypothesis, all symbols of the grammar
       have obtained correct subsizes. And the
       subsizes for the sublist grammar1 are not
       affected by the newly-added symbol.)

= [c] ++ subsize ["<lhs_symb>", k] ++ [1]          (line 9)

= [c] ++ [sumList [productList (subsize [(y, j)])]] ++ [1]
      (line 11)
      (list [y, j] represents the alternative
       sequences of the rule that the new symbol is
       added to, (including the new alternative of the
       newly-added symbol). The algorithm calculates
       the product of the subsizes of the sequence
       symbols, and adds the products for all
       alternatives of the rule together to obtain the
       subsize of the LHS symbol.)

= [c] ++ [d] ++ [1]
      (by the hypothesis, the subsizes for the
       sequence symbols are correctly calculated. In
       addition, the subsize for the newly-added
       symbol is available (e.g., [1]). Therefore, the
       subsize for the LHS symbol is available. Let it
       be d.)
```

Lists [c], [d], and [1] include the correct subsizes for the n symbols of the grammar and the newly-added symbol. List [c]++[d]++[1] includes correct subsizes for the symbols of the grammar and the new symbol.

3. Proofs of the ABF Algorithm

Therefore, by adding a new terminal as a new alternative of the right hand side of the rule, the algorithm correctly calculates the subsizes for each symbol of the grammar.

Case 2: the newly-added symbol is a non-terminal, $\langle x \rangle$, on the right-hand-side of the rule. Since the index of the list starts with 0, the index of the newly-added non-terminal, $\langle x \rangle$, will be n .

Note that, this newly-added non-terminal must be an existing symbol that has appeared in the grammar. By the restrictions for the algorithm (section 2.5), the grammar must be proper, which means all the non-terminals must be defined (appearing on the left hand side of the rule). So the sole newly-added non-terminal cannot be a new non-terminal without proper definition.

There are two cases while adding a new non-terminal, $\langle x \rangle$:

1) The newly-added non-terminal $\langle x \rangle$ is added to an existing symbol sequence on the right hand side of the rule. Note that, this newly-added symbol will affect the subsize of the LHS symbol (e.g., `lhs_symb`) of the current rule and will not affect the subsizes of other symbols. If the subsizes for the affected LHS symbol (`lhs_symb`) and the newly-added symbol are calculated correctly, the algorithm correctly calculates the subsizes for all the symbols of the grammar.

Suppose the index for the affected LHS symbol (`lhs_symb`) is k , where $0 \leq k < n$, and the rest of the grammar symbols are in a sublist `grammar1`.

Therefore:

3. Proofs of the ABF Algorithm

```
subsizeList_new
= subsize (grammar ++ [("<x>", n)])

= subsize (grammar1 ++ [ "<lhs_symb>", k] ++ [("<x>", n)])

= subsize grammar1 ++ subsize [ "<lhs_symb>", k] ++
  subsize [("<x>", n)])
      (line 12 & definition of "++" and ":")

= [c] ++ subsize [ "<lhs_symb>", k] ++ subsize [("<x>", n)])
  ([c] is a list of previously computed values,
   which is the list of subsize for list
   grammar1.
   By the hypothesis, all symbols of the grammar
   have obtained correct subsize. And the
   subsize for the sublist grammar1 are not
   affected by the newly-added symbol.)

= [c] ++ subsize [ "<lhs_symb>", k] ++ subsize [("<y>", m)]
      (line 10)
  (where, <x>=<y>, <y> is a LHS non-terminal in
   the list grammar.)

= [c] ++ subsize [ "<lhs_symb>",k] ++ [d]
  ([d] is a previously calculated value.
   By the hypothesis, the symbols in the list
   grammar have all obtained the correct subsize.
   Without the loss of generality, let the
   subsize for [("<y>", m)] = [d])

= [c] ++ [sumList [productList (subsize [(y, j)])]] ++ [d]
      (line 11)
  (list [y,j] represents the alternative sequences
   of the rule that the new symbol is added to,
   (including the sequence with the new symbol).
   The algorithm calculates the product of the
   subsize of the sequence symbols, and adds the
   products for all alternatives of the rule
   together to obtain the subsize of the LHS
   symbol.)

= [c] ++ [e] ++ [d]
  (By the hypothesis, the subsize for the
   sequence symbols are correctly calculated. The
   subsize for the newly-added symbol is available
   (e.g., [d]). Therefore, the subsize for the LHS
   symbol is available. Say [e].)
```


3. Proofs of the ABF Algorithm

Lists $[c]$, $[e]$, and $[d]$ include the correct subsizes. List $[c]++[e]++[d]$ includes correct subsizes for the symbols of the grammar and the new symbols.

Therefore, by adding a new non-terminal to an existing symbol sequence on the right hand side of the rule, the algorithm correctly calculates the subsizes for each symbol of the grammar.

- 2) The newly-added non-terminal $\langle x \rangle$ is a new alternative of the right hand side of the rule.

In this case, the newly-added symbol will affect the subsize of the LHS symbol (e.g., lhs_symb) in the current rule. The subsizes for other symbols of the grammar will not change. If the subsizes for the affected LHS symbol (lhs_symb) and the newly-added symbol are calculated correctly, the algorithm correctly calculates the subsizes for all the symbols of the grammar.

Suppose the index for the affected LHS symbol (lhs_symb) is k , where $0 \leq k < n$, and the rest of the grammar symbols are in a sublist `grammar1`.

Therefore:

3. Proofs of the ABF Algorithm

```
subsizeList_new
= subsize (grammar ++ [("<x>", n)])

= subsize (grammar1 ++ [("<lhs_symb>", k) ++ [("<x>", n)])

= subsize grammar1 ++ subsize [("<lhs_symb>", k) ++
  subsize [("<x>", n)])
      (line 12 & definition of "++" and ":")
= [c] ++ subsize [("<lhs_symb>", k) ++ subsize [("<x>", n)]]
  ([c] is a list of previously computed values,
   which is the list of subsizes for list
   grammar1.
   By the hypothesis, all symbols of the grammar
   have obtained correct subsizes. And the
   subsizes for the sublist grammar1 are not
   affected by the newly-added symbol.)

= [c] ++ subsize [("<lhs_symb>", k) ++ subsize [("<y>", m)]
      (line 10)
  (where, <x>=<y>, <y> is a LHS non-terminal in
   the list grammar)

= [c] ++ subsize [("<lhs_symb>", k) ++ [d]
  ([d] is a previously calculated value.
   By the hypothesis, the symbols in the list
   grammar have all obtained the correct subsizes.
   Without the loss of generality, let the
   subsize for [("<y>", m)] = [d])

= [c] ++ [sumList [productList (subsize [(y, j)])]] ++ [d]
      (line 11)
  (list [y, j] represents the alternative
   sequences of the rule that the new symbol is
   added to, (including the new alternative of the
   newly-added symbol). The algorithm calculates
   the product of the subsizes of the sequence
   symbols, and adds the products for all
   alternatives of the rule together to obtain the
   subsize of the LHS symbol.)

= [c] ++ [e] ++ [d]
  (by the hypothesis, the subsizes for the
   sequence symbols are correctly calculated. In
   addition, the subsize for the newly-added
   symbol is available (e.g., [d]). Therefore, the
   subsize for the LHS symbol is available. Let it
   be [e].)
```

3. Proofs of the ABF Algorithm

Lists $[c]$, $[e]$, and $[d]$ include the correct subsizes. List $[c]++[e]++[d]$ includes correct subsizes of the symbols and the new symbol.

Therefore, by adding a new non-terminal as a new alternative of the right hand side of the rule, the algorithm correctly calculates the subsizes for each symbol of the grammar.

2. Adding a new rule to the grammar, the algorithm correctly calculates the subsizes for each symbol of the grammar, i.e., the new list of subsizes for all symbols of the grammar will be obtained successfully,

```
subsizeList_new = subsize (grammar ++ newRule)
```

There are three cases in the inductive step:

Case 1: the new rule has a non-terminal on the left hand side, and a terminal on the right hand side, i.e.,

```
<new_r> = "x"
```

The indexes for $\langle \text{new_r} \rangle$ and "x" will be n and $(n+1)$ respectively. The list for this new rule is represented as:

```
newRule = [("<new_r>", n), ("x", (n+1))]
```

Therefore:

```
subsizeList_new
= subsize (grammar ++ newRule)

= subsize grammar ++ subsize newRule
    (line 12 & definition of "++" and ":")

= subsizeList++ subsize [("<new_r>", n), ("x", (n+1))]
    (hypothesis)

= subsizeList ++ subsize [("<new_r>", n)] ++
  subsize [("x", (n+1))]
    (line 12 & definition of "++" and ":")
```

3. Proofs of the ABF Algorithm

```
= subsizeList ++ subsize [("<new_r>", n)] ++ [1]      (line 9)

= subsizeList ++
  [sumList [productList (subsize [("x", (n+1))])] ++ [1]
                                     (line 11)

= subsizeList ++ [sumList [1]] ++ [1]                  (line 13)

= subsizeList ++ [1] ++ [1]                            (definition of sumList)

= subsizeList ++ [1, 1]                                (definition of "++")
```

Since `subsizeList` is a list including the correct subsizes for the n symbols of the grammar (by the hypothesis) and `[1,1]` is a list with correct subsizes, list `subsizeList++[1,1]` includes correct subsizes for the n symbols and the newly-added rule.

Therefore, adding a new rule with one non-terminal on the left hand side and one terminal on the right hand side, the algorithm correctly calculates the subsizes for each symbol of the grammar.

Case 2: The new rule includes terminal(s) and/ or known non-terminal(s) on the right hand side, i.e.,

$$\text{<new_r>} = x,$$

where `<new_r>` is a known non-terminal, and `x` is an expression of sequence(s) and/or alternative(s) with known non-terminal(s) and/or terminal(s). The indexes for `<new_r>` and `x` will be n and $(n+k)$, $k \geq 1$, respectively. The list for this new rule is represented as:

$$\text{newRule} = [(\text{<new_r>}, n), (x, (n+k))]$$

Therefore:

3. Proofs of the ABF Algorithm

```
subsizeList_new
= subsize (grammar ++ newRule )

= subsize grammar ++ subsize newRule
    (line 12 & definition of "++" and ":")

= subsizeList ++ subsize [("<new_r>", n), (x, (n+k))]
    (hypothesis)

= subsizeList ++ subsize [("<new_r>", n)] ++
    subsize [(x, (n+k))]
    (line 12 & definition of "++" and ":")

= subsizeList ++ subsize [("<y>", m)] ++ subsize[(z, j)]
    (line 10)
    (where, <new_r>=<y>, <y> is a known non-terminal)
    (x=z, z represents a known non-terminal or a terminal)

= subsizeList ++ [c] ++ [d]
    (c and d are previously correctly calculated values.)
    (By the hypothesis, the symbols in the list grammar
    have all obtained the correct subsizes, and
    terminals have the subsize of 1 (line 9)), so we can
    obtain the subsizes for: subsize [("<y>", m)] and
    subsize [(z, j)], say [c] and [d]. )
```

Since `subsizeList` is a list including the correct subsizes for the n symbols of the grammar (by the hypothesis) and lists `[c]` and `[d]` are lists with correct subsizes, the list `subsizeList++[c]++[d]` includes correct subsizes for the n symbols and the newly-added rule.

Therefore, if adding a new rule with all known non-terminals and/or terminals, the algorithm correctly calculates the subsizes for each symbol of the grammar.

Case 3: The new rule includes a new non-terminal on the left hand side of the new rule, and known non-terminal(s) and/or terminal(s) on the right-hand side, i.e.,

$$\text{<new_r>} = x$$

where `<new_r>` is a new non-terminal, and `x` is an expression of sequence(s) and/or alternative(s) with known non-terminal(s) and/or terminal(s). The indexes

3. Proofs of the ABF Algorithm

for $\langle \text{new_r} \rangle$ and x will be n and $(n+k)$, $k \geq 1$, respectively. The list for this new rule is represented as:

$$\text{newRule} = [(\langle \text{new_r} \rangle, n), (x, (n+k))]$$

Note that, the non-terminals on the right hand side of the newly-added rule cannot be new because the algorithm should be “proper” (section 2.4), which means every non-terminal has to be defined properly (appear on the left hand side of the rule). So, it is not allowed to include any new non-terminal on the right hand side of the newly-added rule without further definition for it.

Therefore:

3. Proofs of the ABF Algorithm

```
subsizeList_new
= subsize (grammar ++ newRule )

= subsize grammar ++ subsize newRule
      (line 12 & definition of "++" and ":")

= subsizeList ++ subsize [("<new_r>", n), (x, (n+k))]
      (hypothesis)

= subsizeList ++ subsize [("<new_r>", n)] ++
  subsize [(x, (n+k))]
      (line 12 & definition of "++" and ":")

= subsizeList ++ subsize [("<new_r>", n)] ++
  subsize [(z, j)] (line 10)
  (where x=z and z is a known non-terminal or a
   terminal)

= subsizeList ++ subsize [("<new_r>", n)] ++ [d]
  (d is a previously correctly calculated value.)
  (By the hypothesis, the symbols in the list grammar
   have all obtained the correct subsizes, and the
   terminals have the subsize of 1 (line 9)), so we can
   obtain the subsize [(z, j)], say [d].)

= subsizeList ++
  [sumList [productList (subsize [(q, t)])]] ++ [d]
      (line 11)
  ([ (q, t) ] represents the alternative sequences among
   the list [(z, j)]. The subsizes for the sequence
   symbols are multiplied, the products of the
   alternative sequences are summed up.)

= subsizeList ++ [sumList ([productList [c]]) ] ++ [d]
      (line 10)
  ([c] is a list of previously correctly calculated
   values, denoting the subsizes for the sequence
   symbols.)
  (By the hypothesis, the symbols in the list
   grammar have all obtained the correct subsizes.)

= subsizeList ++ [sumList [e]] ++ [d] (line 13)
      (definition of productList)
      (let the product of list [c] be e.)

= subsizeList ++ [f] ++ [d]
      (definition of sumList.)
      (Let the sum of list [e] be f )
```

3. Proofs of the ABF Algorithm

Since `subsizeList` is a list including the correct subsizes for the n symbols of the grammar (by the hypothesis) and lists `[f]` and `[d]` are lists with correct subsizes, the list `subsizeList++[f]++[d]` includes correct subsizes for the n symbols and the newly-added rule.

Therefore, if adding a new rule with a new non-terminal on the left hand side and known non-terminal(s) and/or terminal(s) on the right hand side of the rule, the algorithm correctly calculates the subsizes for each symbol of the grammar.

(3) Proof of correctness for step 3 (component function `occur_rhs`, lines 15 – 18):

Base Case: there is one rule in the grammar with one non-terminal on the left hand side and one terminal on the Right Hand Side of the rule, e.g.,

$$\langle g \rangle = "a"$$

In this case, the list `grammar` will be represented as:

$$\text{grammar} = [(\langle g \rangle, 0), ("a", 1)]$$

Therefore:

```
occur_rhs grammar
= occur_rhs [(\langle g \rangle, 0)] ++ occur_rhs [("a", 1)] (line 17)
= [0] ++ occur_rhs [("a", 1)]
                                     (line 16, skip left-hand symbol)
= [0] ++ [productList [subsizeList!1]] (line 16)
= [0] ++ [productList [1] ]
                                     (terminals have the subsize of 1, step 2)
= [0] ++ [1] (definition of productList)
= [0, 1] (definition of "++")
```


3. Proofs of the ABF Algorithm

For base case, the algorithm correctly calculates the number of occurrences for RHS symbols of the grammar.

Inductive step:

Hypothesis: for a grammar with n symbols, (i.e., `#grammar = n`), the algorithm correctly calculates the number of occurrences for right-hand-side symbols of the grammar, i.e., `occur_rhsList[num]` is obtained successfully,

```
occur_rhsList = occur_rhs grammar
```

Show:

1. Adding one more symbol, x , to the right hand side of the rule, the algorithm correctly calculates the number of occurrences of right-hand-side symbols of the grammar, i.e., the new list of the number of occurrences for right-hand-side symbols of the grammar will be obtained successfully,

```
occur_rhsList_new = occur_rhs (grammar ++ [(x, n)])
```

Note that, this newly-added RHS symbol must be a terminal or a non-terminal that has been defined in the grammar. The reason is that the grammar must be proper (section 2.5), which means that all non-terminals of the grammar must be properly defined (appearing on the left hand side of the rule).

There are two cases while adding a symbol to the right hand side of the rule:

1) The new symbol is added as a new alternative of the rule. In this case, the newly-added symbol will not affect the number of occurrences for other symbols of the grammar.

Therefore:

3. Proofs of the ABF Algorithm

```
occur_rhsList_new
= occur_rhs (grammar ++ [(x, n)])

= occur_rhs grammar ++ occur_rhs [(x, n)]
      (line 17 & definition of "++" and ":")

= occur_rhsList ++ occur_rhs [(x, n)]      (hypothesis)

= occur_rhsList ++ [productList [subsizeList ! m]]
      (line 16)
      (take the subsizes of the symbols of the sequence)

= occur_rhsList ++ [productList [c]]
      (c is a known value.)
      (If x is a terminal, the subsize for x is 1.
       If x is a known non-terminal, its subsize has
       been set in step 2.)

= occur_rhsList ++ [c]      (definition of productList)

      (Since x is the only symbol of the alternative
       sequence, the product of the subsizes of the
       sequence is c. )
```

Since `occur_rhsList` is a list including the correct number of occurrences of RHS symbols of the grammar (by the hypothesis) and list `[c]` is a list with correct values of the number of occurrences, list `occur_rhsList++[c]` includes correct values of the number of occurrences for the n symbols and the newly-added symbol.

Therefore, when a new symbol is added as a new alternative on the right hand side of the rule, the algorithm correctly calculates the number of occurrences for each symbol of the grammar.

2) The newly-added symbol is added to an existing symbol sequence on the right hand side of the rule. Note that, this newly-added symbol will affect the number of occurrences for the symbols of the sequence. The number of occurrences for other symbols will not change. If the values of the number of occurrences for the affected sequence symbols and the newly-added symbol are calculated correctly, the algorithm correctly calculates the number of occurrences

for all the symbols of the grammar.

Suppose the affected sequence symbols are in list `grammar2`, and the rest symbols of the grammar are in list `grammar1`.

Therefore:

```
occur_rhsList_new
= occur_rhs (grammar ++ [(x, n)])

= occur_rhs grammar1 ++ occur_rhs grammar2 ++
  occur_rhs [(x, n)]
  (line 17 & definition of "++" and "(:)")

= [c] ++ occur_rhs grammar2 ++ occur_rhs [(x, n)]
  ([c] is a list includes correctly calculated values.
  By the hypothesis, the number of occurrences for the
  symbols of the grammar have all been calculated
  correctly. And the symbols of list grammar1 are
  not affected by the newly-added symbol)

= [c] ++ occur_rhs grammar2 ++
  [productList [subsizeList ! m]]
  (line 16)
  (take the subsizes of the symbols of the sequence)

= [c] ++ occur_rhs grammar2 ++ [productList [d]]
  (d is a known value representing the subsize for
  each symbol of the sequence. Because, if the newly-
  added symbol is a terminal, its subsize is 1. If
  the newly-added symbol is a known non-terminal, its
  subsize is available in step 2.)

= [c] ++ occur_rhs grammar2 ++ [e]
  (definition of productList)
  (e is a known value which is the product of the
  subsizes of the symbols of the sequence.)

= [c] ++ [f] ++ [e]
  ([f] is a list with known values representing the
  number of occurrences for the sequence symbols.
  Actually, each item of the list [f] has the same
  value e, which is the number of occurrences of the
  newly-added symbol. Note that, all the symbols of a
  sequence have the same number of occurrences.)
```

3. Proofs of the ABF Algorithm

Since lists $[c]$, $[f]$, and $[e]$ are lists with correct values of the number of occurrences, list $[c]++[f]++[e]$ includes correct values of the number of occurrences for the n symbols and the newly-added symbol.

Therefore, when adding a new symbol to an existing sequence on the right hand side of the rule, the algorithm correctly calculates the number of occurrences for each symbol of the grammar.

2. Adding a new rule to the grammar, the algorithm correctly calculates the number of occurrences for the right-hand-side symbols of the grammar, i.e., the new list of the number of occurrences for right-hand-side symbols of the grammar will be obtained successfully, i.e.:

```
occur_rhsList_new = occur_rhs (grammar ++ newRule)
```

The new rule is expressed in the list as:

```
newRule = [("<new_r>", n), (x, (n+k))],
```

where x is an expression of alternative(s) and/or sequence(s) of terminal(s) and/or non-terminal(s). The indexes for <new_r> and x are n and $(n+k)$, $k \geq 1$, respectively.

Note that, the non-terminals on the right hand side of the newly-added rule cannot be new because the algorithm should be “proper” (section 2.5), which means every non-terminal has to be properly defined (appear on the left hand side of the rule). So, it is not allowed to include any new non-terminal on the right hand side of the newly-added rule without further definition for it.

Therefore:

3. Proofs of the ABF Algorithm

```
occur_rhsList_new
= occur_rhs (grammar ++ newRule)

= occur_rhs grammar ++ occur_rhs newRule
      (line 17 & definition of "++" and ":")

= occur_rhsList ++ occur_rhs [(" <new_r>", n), (x, (n+k))]
      (hypothesis)

= occur_rhsList ++ occur_rhs [(" <new_r>", n)] ++
  occur_rhs [(x, (n+k))]
      (line 17 & definition of "++" and ":")

= occur_rhsList ++ [0] ++ occur_rhs [(x, (n+k))] (line 16)

= occur_rhsList ++ [0] ++ [productList [subsizeList!m]]
      (line 16)

= occur_rhsList ++ [productList [c]]
  ([c] is a list of known values, which are subsizes
   of the sequence symbols.
   x is an expression consisting of terminal(s)
   and/or known non-terminal(s).
   The subsizes for terminals are 1.
   The subsizes for known non-terminals are available
   in step 2.
   So the list of the subsizes for the symbols of the
   right hand side of the newly-added rule is
   available, say list [c])

= occur_rhsList ++ [d] (definition of productList)
  (d is a known value, which is the product of the
   subsizes of the symbols of the sequence)
```

Since `occur_rhsList` is a list including the correct number of occurrences of right-hand-side symbols of the grammar (by the hypothesis) and list `[d]` is a list with correct values of the number of occurrences, the list `occur_rhsList++[d]` includes correct values of the number of occurrences for the n symbols and the newly-added rule.

Therefore, when adding a new rule to the grammar, the algorithm correctly calculates the number of occurrences for the right-hand-side symbols of the grammar.

3. Proofs of the ABF Algorithm

- (4) Proof of correctness for step 4 (component function `occur_lhs` and `factor`, lines 19 – 29):

Base Case: there is one rule in the grammar with one non-terminal on the left hand side and one terminal on the right hand side of the rule, e.g.,

`<g> = "a"`

In this case, the list `grammar` will be presented as follows:

`grammar = [("<g>", 0), ("a", 1)]`

Therefore:

```
occur_lhs grammar
= occur_lhs [("<g>", 0), ("a", 1)]
= occur_lhs [("<g>", 0)] ++ occur_lhs [("a", 1)]           (line 23)
= occur_lhs [("<g>", 0)] ++ [0]                             (line 22)
= [subsizeList! 0] ++ [0]                                    (line 20)
= [c] ++ [0]
    (c is the subsize of the start symbol. It is a
     known value, because subsizeList is available
     in step 2)
= [c, 0]                                                     (definition of "++")
```

```
factor grammar
= factor [("<g>", 0), ("a", 1)]
= factor [("<g>", 0)] ++ factor [("a", 1)]                 (line 27)
= [occur_lhsList[("<g>",0)]!0 / subsizeList !0] ++
  factor [("a", 1)]                                       (line 25)
= [subsizeList!0 / subsizeList!0] ++
  factor [("a", 1)]                                       (line 20)
    (subsizeList is available in step 2)
```

3. Proofs of the ABF Algorithm

```
= [1] ++ factor [("a", 1)]
      (Note that, multiplication factor for the
      first rule is 1.)

= [1] ++ factor [(" <g>", 0)]                      (line 26)

= [1] ++ [1]
      (The multiplication factor for a RHS symbol is
      the same as that for the LHS non-terminal of
      the rule.)

= [1, 1]                      (definition of "++")
```

For the base case, the algorithm correctly calculates the number of occurrences for left-hand-side symbols and the multiplication factors for the right-hand-side symbols of the grammar.

Inductive step:

Hypothesis: for a grammar with n symbols, (i.e., `#grammar = n`), the algorithm correctly calculates the number of occurrences for the left-hand-side symbols of the grammar, i.e., `occur_lhsList[num]` is obtained successfully,

```
occur_rhsList = occur_rhs grammar
```

Also, the algorithm correctly calculates the multiplication factors for right-hand-side symbols of the grammar, i.e., `factList[num]` is obtained successfully,

```
factList = factor grammar
```

Show:

1. When adding one more symbol, x , to the right hand side of the rule, the algorithm correctly calculates the number of occurrences for the left-hand-side symbols of the grammar, i.e., the new list of the number of occurrences for left-hand-side symbols of the grammar will be obtained successfully,

```
occur_lhsList_new = occur_rhs (grammar ++ [(x, n)])
```

3. Proofs of the ABF Algorithm

Also, the algorithm correctly calculates the multiplication factors for the right-hand-side symbols of the grammar.

```
factorList_new = factor (grammar ++ [(x, n)])
```

There are two cases while adding a symbol to the right hand side of the rule.

1) The new symbol is a terminal, “x”, on the right hand side of the rule. In this case, the newly-added symbol will not affect the number of occurrences for the LHS symbols and the multiplication factors for other RHS symbols of the grammar.

```
occur_lhsList_new
= occur_lhs (grammar ++ [(“x”, n)])

= occur_lhs grammar ++ occur_lhs [(“x”, n)]
  (line 23 & definition of “++” and “:”)

= occur_lhsList ++ occur_lhs [(“x”, n)]      (hypothesis)

= occur_lhsList ++ [0]                        (line 22)
  (For a RHS symbol, the number of occurrences for a LHS
   symbol is 0)
```

By the hypothesis, occur_lhsList is a list with correct number of occurrences for LHS symbols. [0] is a correct value of number of occurrences of a LHS symbol for a RHS symbol. Therefore, occur_lhsList++[0] is a list includes correct values of the number of occurrences for LHS symbols of the grammar and the algorithm correctly calculates the number of occurrences for the LHS symbols and the newly-added symbol.

3. Proofs of the ABF Algorithm

```
factorList_new
= factor (grammar ++ [("x", n)])

= factor grammar ++ factor [("x", n)]
    (line 9 & definition of "++" and ":" )

= factorList ++ factor [("x", n)]          (hypothesis)

= factorList ++ factor [(y, m)]            (line 26 )
    (y is the LHS non-terminal of the same rule as "x")

= factorList ++ [c]
    (c is a known value, which is the multiplication
    factor of the symbol y.
    (by the hypothesis, multiplication factors for the
    symbols of the grammar have been all calculated
    correctly. Let the multiplication factor for y be
    c.)
```

List `factorList` includes correct multiplication factors for the symbols of the grammar (by the hypothesis), and list `[c]` includes correct multiplication factor for the newly-added symbol. Therefore, list `factorList++[c]` includes correct multiplication factors for symbols of the grammar and the algorithm correctly calculates the multiplication factors for the n symbols of the grammar and the newly-added symbol.

2) The new symbol is a known non-terminal, $\langle x \rangle$, on the right hand side of the rule. In this case, the newly-added symbol will affect the number of occurrences of the LHS symbol ($\langle \text{sym_lhs} \rangle$, where $\langle \text{sym_lhs} \rangle = \langle x \rangle$) and the multiplication factors for the rule where this symbol ($\langle x \rangle$) appears on the left hand side of the rule. The number of occurrences and the multiplication factors for other LHS symbols will not change. So, if the values of the number of occurrences for the newly-added RHS symbol ($\langle x \rangle$) and the LHS symbol ($\langle \text{sym_lhs} \rangle = \langle x \rangle$) are calculated correctly, the algorithm calculates the number of occurrences for the LHS symbols of the grammar and the newly-added symbol correctly.

Suppose the affected LHS symbol is $\langle \text{sym_lhs} \rangle$, whose index is k , where

3. Proofs of the ABF Algorithm

$0 \leq k < n$. Note that, $\langle \text{sym_lhs} \rangle = \langle x \rangle$. However, $\langle \text{sym_lhs} \rangle$ is on the LHS of the rule, and $\langle x \rangle$ is a newly-added RHS symbol. The other symbols of the grammar are in sublist `grammar1`. Therefore,

```
occur_lhsList_new
= occur_lhs (grammar ++ [("<x>", n)])

= occur_lhs (grammar1 ++ [("<sym_lhs>", k)] ++
  [("<x>", n)])

= occur_lhs grammar1 ++ occur_lhs [("<sym_lhs>", k)] ++
  occur_lhs [("<x>", n)]
      (line 23 & definition of "++" and ":")

= [c] ++ occur_lhs [("<sym_lhs>", k)] ++
  occur_lhs [("<x>", n)]
    ([c] is a list including the previously calculated
     values of the number of occurrences for the LHS
     symbols of the grammar.
     By the hypothesis, the number of the occurrences
     for the LHS symbols are available.)

= [c] ++ occur_lhs [("<sym_lhs>", k)] ++ [0]    (line 22)
    (For a RHS symbol, the number of occurrences for a
     LHS symbol is 0)

= [c] ++ [sumList [(occur_rhsList !h) *
  ((factor [(y, h)])!0) ] ++ [0]                (line 21)
  (The number of occurrences for a LHS symbol is the
   sum of all the instances of that symbol occurring
   on the RHS of all rules, each multiplied by the
   multiplication factor.)

= [c] ++ [sumList [d * e]] ++ [0]
  (d and e are known values representing the number
   of occurrences and the multiplication factor for
   the RHS symbol.
   From step 3, the number of occurrences for RHS
   symbols are available.
   By the hypothesis, the multiplication factors for
   the RHS symbols of the grammar are correctly
   calculated.)

= [c] ++ [f] ++ [0]                (definition of sumList)
                                     (f is a known value)
```

3. Proofs of the ABF Algorithm

Lists $[c]$, $[f]$, and $[0]$ include correct values of the number of occurrences of the LHS symbols. Therefore, list $[c]++[f]++[0]$ includes correct values of the number of occurrences of the symbols of the grammar and the algorithm correctly calculates the number of occurrences of the symbols of the grammar and the newly-added symbol.

In addition, this newly-added non-terminal ($\langle x \rangle$) will affect the multiplication factors for the rule where this symbol ($\langle x \rangle$) appears on the left hand side of the rule. Suppose the LHS symbol of the affected rule is $\langle \text{sym_lhs} \rangle$, whose index is k , where $0 \leq k < n$. Note that, $\langle \text{sym_lhs} \rangle = \langle x \rangle$. However, $\langle \text{sym_lhs} \rangle$ is on the LHS of the rule, and $\langle x \rangle$ is a newly-added RHS symbol. The RHS symbols of the affected rule are in sublist `grammar3`. The other symbols of the grammar are in sublist `grammar2`. Note that, the multiplication factors for symbol $\langle \text{sym_lhs} \rangle$ and symbols in `grammar3` are affected by the newly-added non-terminal. The multiplication factors for symbols in `grammar2` will not change.

Therefore:

```
factorList_new
= factor (grammar ++ [(" $\langle x \rangle$ ", n)])

= factor (grammar2 ++ [(" $\langle \text{sym\_lhs} \rangle$ ", k)] ++ grammar3 ++
  [(" $\langle x \rangle$ ", n)] )

= factor grammar2 ++ factor [(" $\langle \text{sym\_lhs} \rangle$ ", k)] ++
  factor grammar3 ++ factor [(" $\langle x \rangle$ ", n)]

= [c] ++ factor [(" $\langle \text{sym\_lhs} \rangle$ ", k)] ++
  factor grammar3 ++ factor [(" $\langle x \rangle$ ", n)]
  ([c] is a list including previously calculated
   values representing the multiplication factors for
   the symbols in grammar2 which do not change.
   By the hypothesis, the multiplication factors for
   the symbols of the grammar have all been
   calculated correctly.)

= [c] ++ [(occur_lhs[(" $\langle \text{sym\_lhs} \rangle$ ", k)]) /
  (subsize [(" $\langle \text{sym\_lhs} \rangle$ ", k)])] ++
  factor grammar3 ++ factor [(" $\langle x \rangle$ ", n)]           (line 25)
```

3. Proofs of the ABF Algorithm

(The multiplication factor of a LHS symbol is obtained by dividing the number of occurrences of the symbol on the left hand side of the rule by its sub-language size of that symbol.)

= [c] ++ [d] ++ factor grammar3 ++ factor [("<x>", n)]
(*d* is a known value. The number of occurrences for the LHS symbol <sym_lhs> is available from above proof for occur_lhsList_new. The sub-language size for the symbol <sym_lhs> is available in step 2.)

= [c] ++ [d] ++ [e] ++ factor [("<x>", n)] (line 26)
([*e*] is a list including known values.
grammar3 is a list including the RHS symbols of the affected rule. From line 26, the multiplication factors for the RHS symbols are the same as that of the LHS symbol. Therefore, each item in list [*e*] has the value *d*, which is the multiplication factor of symbol <sym_lhs>.)

= [c] ++ [d] ++ [e] ++ factor [("<y>", n)] (line 26)
(<*y*> is the LHS non-terminal of the same rule as the newly-added symbol, <*x*>)

= [c] ++ [d] ++ [e] ++ [*f*]
(*f* is a known value, which is the multiplication factor of the symbol <*y*>.)
(by the hypothesis, multiplication factors for the symbols of the grammar have been all calculated correctly. And the multiplication factor for <*y*> is not affected, let it be *f*.)

Lists [*c*], [*d*], [*e*], and [*f*] include correct multiplication factors for the *n* symbols and the newly-added symbol. Therefore, list [*c*]++[*d*]++[*e*]++[*f*] includes correct multiplication factors for the symbols of the grammar and the algorithm correctly calculates the multiplication factors for the symbols of the grammar and the new symbol.

Therefore, when adding one symbol on the Right-Hand-Side of the rule, the algorithm correctly calculates the number of occurrences for the left-hand-side

3. Proofs of the ABF Algorithm

symbols and the multiplication factors for the right-hand-side symbols of the grammar.

2. When adding a new rule to the grammar, the algorithm correctly calculates the number of occurrences for the left-hand-side symbols of the grammar, i.e., the new list of the number of occurrences for left-hand-side symbols of the grammar will be obtained successfully,

```
occur_lhsList_new = occur_lhs (grammar ++ newRule)
```

Also, the multiplication factors for the right-hand-side symbols of the grammar will be calculated correctly,

```
factorList_new = factor (grammar ++ newRule)
```

The new rule is expressed in list as follows:

```
newRule = [("<new_r>", n), (x, (n+k))],
```

where x is an expression of alternative(s) and/or sequence(s) of terminal(s) and/or non-terminal(s). The indexes for <new_r> and x are n and $(n+k)$, $k \geq 1$, respectively.

Note that, the non-terminals on the right hand side of the newly-added rule cannot be new because the algorithm should be “proper” (section 2.5), which means every non-terminal has to be properly defined (appear on the left hand side of the rule). So, it is not allowed to include any new non-terminal on the right hand side of the newly-added rule without further definition for it.

There are two cases while adding a new rule.

1) There are only terminal(s) on the right hand side of the new rule. In this case, the newly-added rule will not affect the number of occurrences for other LHS symbols and the multiplication factors for other RHS symbols of the grammar.

3. Proofs of the ABF Algorithm

```

occur_lhsList_new
  = occur_lhs (grammar ++ newRule)

  = occur_lhs grammar ++ occur_lhs newRule
    (line 23 & definition of "++" and ":")

  = occur_lhsList++ occur_lhs [("<new_r>", n), (x, (n+k))]
    (hypothesis)

  = occur_lhsList ++ occur_lhs [("<new_r>", n)] ++
    occur_lhs [(x, (n+k))]
    (line 23 & definition of "++" and ":")

  = occur_lhsList ++ occur_lhs [("<new_r>", n)] ++ [0]
    (line 22)

  = occur_rhsList ++ [sumList [(occur_rhsList!h) *
    ((factor [(y, h)]) !0)|
    (y,h) <- grammar;
    y = <new_r>;
    isRHS !h ]
    ++ [0]
    (line 21)
    (The number of occurrences for a LHS symbol is the
    sum of all the instances of that symbol occurring on
    the RHS of all rules, each multiplied by the
    multiplication factor.)

  = occur_lhsList ++ [sumList [d*e]] ++ [0]
    (d and e are known values representing the number of
    occurrences and the multiplication factor for the
    RHS symbol.
    From step 3, the number of occurrences for RHS
    symbols are available.
    By the hypothesis, the multiplication factors for
    the RHS symbols of the grammar are correctly
    calculated.)

  = occur_lhsList ++ [f] ++ [0]      (definition of sumList)
    (f is a known value)

```

By the hypothesis, list `occur_lhsList` includes correct values for the number of occurrences of the LHS symbols. Lists `[f]` and `[0]` include correct values of the number of occurrences of the LHS symbols. Therefore, list `occur_lhsList++[f]++[0]` includes correct values of the number of

3. Proofs of the ABF Algorithm

occurrences of the symbols of the grammar and the new rule. And the algorithm correctly calculates the values of the number of occurrences of the symbols of the grammar and the newly-added rule.

```
factorList_new
= factor (grammar ++ newRule)

= factor grammar ++ factor newRule
      (line 27 & definition of "++" and ":")

= factorList ++ factor [("<new_r>", n), (x, (n+k))]
      (hypothesis)

= factorList ++ factor [("<new_r>", n)] ++
  factor [(x, (n+k))]
      (line 27 & definition of "++" and ":")

= factorList ++ [(occur_lhs [("<new_r>", n)])!0 /
      (subsizeList!n)]
      ++ factor [(x, (n+k))]      (line 25)
(The multiplication factor of a LHS symbol
is obtained by dividing the number of occurrences
of the symbol on the left hand side of the rule by
its subsize of that symbol.)

= factorList ++ [c] ++ factor [(x, (n+k))]
      (c is a known value.
The number of occurrences of the LHS symbol
"<new_r>" is available in the above proof for
occur_lhs.
The subsizeList is available in step 2.)

= factorList ++ [c] ++ [d]      (line 26)
(list [d] includes known values that represent
the multiplication factors for the RHS symbols of
the new rule. The multiplication factors for the
RHS symbols are the same as that of the LHS
symbol, i.e., c.)
```

By the hypothesis, the list `factorList` includes correct multiplication factors for the n symbols of the grammar. Lists `[c]` and `[d]` include correct multiplication factors for the symbols of the newly-added rule. Therefore, list `factorList++[c]++[d]` includes correct multiplication factors for the n

symbols of the grammar and the new rule. And the algorithm correctly calculates the multiplication factors for the symbols of the grammar and the newly-added rule.

Therefore, adding one new rule with only terminals on the right hand side, the algorithm correctly calculates the number of occurrences for the left-hand-side symbols and the multiplication factors for the right-hand-side symbols of the grammar.

2) The right hand side of the new rule includes non-terminal(s). In this case, the newly-added rule will affect the number of occurrences of the LHS symbol(s) and the multiplication factors for the rule(s) where the non-terminal(s) (on the right hand side of the new rule) appear on the left hand side of the rule(s). The number of occurrences and the multiplication factors for other symbols will not change. So, if the values of the number of occurrences and the multiplication factors for the new rule and the affected symbol(s) are calculated correctly, the algorithm calculates the number of occurrences for the LHS symbols and the multiplication factors for the RHS symbols of the grammar correctly.

Suppose the affected LHS symbol(s) are in list `sym_lhs`, and the RHS symbol(s) of the affected rule(s) are in list `sym_rhs`. Note that, list `sym_lhs` actually includes the same non-terminals of the right hand side of the new rule. The other symbols of the grammar are in list `grammar1`. And the number of occurrences and the multiplication factors for the symbols in `grammar1` will not change.

Therefore:

3. Proofs of the ABF Algorithm

```
occur_lhsList_new
= occur_lhs (grammar ++ newRule)

= occur_lhs (grammar1 ++ sym_lhs ++ sym_rhs ++ newRule)

= occur_lhs grammar1 ++ occur_lhs sym_lhs ++
  occur_lhs sym_rhs ++
  occur_lhs [("<new_r>", n), (x, (n+k))]
              (line 23 & definition of "++" and ":")

= [c] ++ occur_lhs sym_lhs ++ occur_lhs sym_rhs ++
  occur_lhs [("<new_r>", n)] ++ occur_lhs [(x, (n+k))]
  ([c] is a list including previously calculated
   values of the number of occurrences for the LHS
   symbols.
   By the hypothesis, the number of occurrences for
   the LHS symbols of the grammar have all been
   correctly calculated. And that for the list
   grammar1 are not affected by the newly-added
   rule.)

= [c] ++ occur_lhs sym_lhs ++ [0] ++
  occur_lhs [("<new_r>", n)] ++ occur_lhs [(x, (n+k))]
              (line 22)
  (For RHS symbols, the values of the number of
   occurrences for LHS symbols are 0.)

= [c] ++ [sumList [(occur_rhsList !h) *
                    ((factor [(y, h)])!0) ]
  ++ [0] ++
  occur_lhs [("<new_r>", n)] ++ occur_lhs [(x, (n+k))]
              (line 21)
  (The number of occurrences for a LHS symbol is the
   sum of all the instances of that symbol occurring
   on the RHS of all rules, each multiplied by the
   multiplication factor.)

= [c] ++ [sumList [d * e]] ++ [0] ++
  occur_lhs [("<new_r>", n)] ++ occur_lhs [(x, (n+k))]
  (d and e are known values representing the number
   of occurrences and the multiplication factor for
   the RHS symbols.
   From step 3, the number of occurrences for RHS
   symbols is available.
   By the hypothesis, the multiplication factors for
   the RHS symbols of the grammar are correctly
   calculated.)
```

3. Proofs of the ABF Algorithm

```
= [c] ++ [f] ++ [0] ++  
    occur_lhs [("new_r", n)] ++ occur_lhs [(x, (n+k))]  
                                (definition of sumList)  
                                (f is a known value)  
  
= [c] ++ [f] ++ [0] ++  
    [sumList [(occure_rhsList !j)*((factor [(y, j)]) !0) ]  
    ++ occur_lhs [(x, (n+k))]  
    (The number of occurrences for a LHS symbol is the  
    sum of all the instances of that symbol occurring  
    on the RHS of all rules, each multiplied by the  
    multiplication factor.)  
  
= [c] ++ [f] ++ [0] ++  
    [sumList [ r*t ] ] ++ occur_lhs [(x, (n+k))]  
    (r and t are known values representing the number  
    of occurrences and the multiplication factor for  
    the RHS symbols.  
    From step 3, the number of occurrences for RHS  
    symbols is available.  
    By the hypothesis, the multiplication factors for  
    the RHS symbols of the grammar are correctly  
    calculated.)  
  
= [c] ++ [f] ++ [0] ++  
    [s] ++ occur_lhs [(x, (n+k))]  
                                (definition of sumList)  
                                (s is a known value)  
  
= [c] ++ [f] ++ [0] ++ [s] ++ [0] (line 22)  
    (For RHS symbols, the values of the number of  
    occurrences for LHS symbols are 0.)
```

Lists [c], [f], [0], [s], and [0] include the correct values for the number of occurrences of the LHS symbols of the grammar and the newly-added rule with non-terminals on the right hand side of the rule. Therefore, list [c]++[f]++[0]++[s]++[0] includes correct values of the number of occurrences for the LHS symbols of the grammar and the new rule. And the algorithm correctly calculates the number of occurrences for the LHS symbols of the grammar and the newly-added rule with non-terminals on the right hand side of the rule.

Therefore:

3. Proofs of the ABF Algorithm

```
factorList_new
= factor (grammar ++ newRule)

= factor (grammar1 ++ sym_lhs ++ sym_rhs ++ newRule)

= factor grammar1 ++ factor sym_lhs ++ factor sym_rhs ++
  factor [("<new_r>", n), (x, (n+k))]
      (line 27 & definition of "++" and ":")

= [c] ++ factor sym_lhs ++ factor sym_rhs ++
  factor [("<new_r>", n)] ++ factor [(x, (n+k))]
  ([c] is a list including previously calculated
   multiplication factors for the symbols.
   By the hypothesis, the multiplication factors for
   the symbols of the grammar have all been
   correctly calculated. And that for the list
   grammar1 are not affected by the newly-added
   rule.)

= [c] ++ [(occur_lhs sym_lhs) / (subsize <sym_lhs)] ++
  factor sym_rhs ++
  factor [("<new_r>", n)] ++ factor [(x, (n+k))]
      (line 25)
  (The multiplication factor of a LHS symbol is
   obtained by dividing the number of occurrences
   of the symbol on the left hand side of the rule
   by its sub-language size of that symbol.)

= [c] ++ [d] ++ factor sym_rhs ++
  factor [("<new_r>", n)] ++ factor [(x, (n+k))]
  (d is a known value.
   The number of occurrences for the LHS symbols in
   list sym_lhs is available from above proof for
   occur_lhsList_new.
   The subsizes for the symbols in list
   sym_lhs are available in step 2.)

= [c] ++ [d] ++ [e] ++
  factor [("<new_r>", n)] ++ factor [(x, (n+k))]
      (line 26)
  ([e] is a list including known values
   representing the multiplication factors of the
   affected RHS symbols.
  sym_rhs is a list including the RHS symbols of
  the affected rule(s). From line 26, the
  multiplication factors for the RHS symbols are
  the same as that of the LHS symbol. )
```

3. Proofs of the ABF Algorithm

```
= [c] ++ [d] ++ [e] ++ [(occur_lhs [("<new_r>", n)] /  
                        (subsize [("<new_r>", n)])]  
  ++ factor [(x, (n+k))]  
                                     (line 25)
```

(The multiplication factor of a LHS symbol is obtained by dividing the number of occurrences of the symbol on the left hand side of the rule by its subsize of that symbol.)

```
= [c] ++ [d] ++ [e] ++ [f] ++ factor [(x, (n+k))]  
(f is a known value.  
  The number of occurrences for the LHS symbol  
  <new_r> is available from above proof for  
  occur_lhsList_new.  
  The sub-language size for the symbols is  
  available in step 2.)
```

```
= [c] ++ [d] ++ [e] ++ [f] ++ [g]  
([g] is a list including known values,  
  representing the multiplication factors of the  
  RHS symbols of the new rule.  
  From line 26, the multiplication factors for the  
  RHS symbols are the same as that of the LHS  
  symbol. )
```

Lists [c], [d], [e], [f], and [g] include correct multiplication factors of the symbols. List [c]++[d]++[e]++[f]++[g] includes correct multiplication factors of the symbols and the new symbol.

Therefore, the algorithm correctly calculates the multiplication factors of the symbols of the grammar and the newly-added rule with non-terminals on the right hand side of the rule.

Therefore, when adding a new rule to the grammar, the algorithm correctly calculates the number of occurrences for the left-hand-side symbols and the multiplication factors for the right-hand-side symbols of the grammar.

(5) Proof of correctness for step 5, step 6, and step 7 (lines 30 – 33):

These steps use the intermediate results from the above four steps (steps 1 to 4) with arithmetic calculation based on the definition of the Average Branching Factor (in section 2.4). Since we have proved the correctness of the steps 1 to 4, steps 5 to 7 are correct.

3.5 Proof of Complexity

We consider the worst-case time complexity for the ABF algorithm with respect to the size of the grammar.

We assume the following operations require a constant amount of time.

- (1) Arithmetic basic operation: $+$, $-$, $*$, $/$.
- (2) Comparison operation: $>$, \geq , $<$, \leq , $=$.
- (3) Logic operation: `and`, `or`, `not`.
- (4) Assignment expression.
- (5) Reading in a character from a text file.

Since there are seven steps in the algorithm ABF which are executed in sequence, the worst of the time complexity among the seven steps will be the worst-case time complexity for the algorithm ABF.

Referring to the Miranda code in section 3.2, the worst-case time complexity for the algorithm ABF is analyzed for each component function (steps of the algorithm). Suppose there are n symbols in the grammar, i.e., the length of the list `grammar` is n , the worst-case time complexity is analyzed as follows:

1) Worst-case time complexity for step 1 (component function `bf`, lines 1 – 7):

Step 1 calculates the branching factors for all of the symbols of the grammar and the results are stored in the list `bfList`.

3. Proofs of the ABF Algorithm

Line 5 deals with the list `grammar` by processing elements (symbols) one by one. In each round of iteration, one symbol is processed and the length of the `grammar` list decreases by 1 until it reaches 0. Therefore, it takes $O(n)$ time to deal with the n symbols of the grammar. Then we need to consider the cost of calculating the branching factor for each symbol, which occurs in lines 1, 2, 3, and 4.

Lines 1 and 2 are base cases of the component function `bf`, which return constant values and execute in constant time $O(1)$.

Line 3 searches the grammar list to find a non-terminal and returns its branching factor. There are n symbols in the grammar. In the worst case, the search of the grammar list will take $O(n)$ time. Also, it may mutually recursive call the component function `bf` in Line 4.

Line 4 calls line 6 to sum up the branching factors for all leftmost-alternatives of the rule, which costs $O(n)$ time.

Line 7 runs the whole program.

The maximum cost for calculating the branching factor for one symbol involves the recursive calls in lines 3 and 4, which costs $O(n \cdot n)$ time. Therefore, the worst-case time complexity for step 1 to calculate the branching factors for n symbols of the grammar is $O(n \cdot n \cdot n) = O(n^3)$.

2) Worst-case time complexity for step 2 (component function `subsize`, lines 8 – 14):

Step 2 calculates the subsizes for all symbols of the grammar and the results are stored in the list `subsizeList`.

Line 12 deals with the list `grammar` by processing elements (symbols) one by one. In each round of iteration, one symbol is processed and the length of the list

3. Proofs of the ABF Algorithm

decreases by 1 until it reaches 0. Therefore, it takes $O(n)$ time to deal with the n symbols of the grammar. Then we need to consider the cost for calculating the subsize for one symbol, which occurs in lines 8, 9, 10, and 11.

Lines 8 and 9 are base cases for the component function `subsize`, which return constant values and execute in constant time $O(1)$.

Line 10 searches the grammar list to find a non-terminal and returns its subsize for the current symbol. The search in `grammar` list takes $O(n)$ time. Also, it mutually recursive calls the algorithm in line 11.

Line 11 calls line 6 and line 13 to sum the product of the subsizes of all the sequences of the rule. The worst-case cost for `sumList` and `productList` is $O(n)$ each. The cost for line 11 is $O(n*n)$.

Line 14 runs the whole program.

The maximum cost for calculating the subsize for one symbol involves the mutual recursive calls in lines 10 and 11, which take $O(n*n*n)$ time. Therefore, the worst-case time complexity for step 2 to calculate the subsizes for n symbols of the grammar is $O(n * n*n*n) = O(n^4)$.

3) Worst-case time complexity for step 3 (component function `occur_rhs`, lines 15 – 18):

Step 3 calculates the number of occurrences for the symbols on the Right Hand Side of the rules and the results are stored in the list `occur_rhsList`.

Line 17 deals with list `grammar` by processing elements (symbols) one by one. In each round of iteration, one symbol is processed and the length of the list decreases by 1 until it reaches 0. Therefore, it takes $O(n)$ time to deal with the n symbols of the grammar. Then we need to consider the cost for calculating the number of occurrences for one RHS symbol, which occurs in lines 15 and 16.

Line 15 is the base case for the component function `occur_rhs`, which returns a constant value and executes constant time $O(1)$.

Line 16 calls the function `productList` (line 13) to calculate the number of occurrences for one RHS symbol by calculating the product of the subsize of each RHS symbol in the sequence. The `subsizeList` is available in step 2. Function `productList`(line 13) costs $O(n)$ time.

The maximum cost of calculating the number of occurrences for one RHS symbol occurs in line 16, which takes $O(n)$ time. Therefore, the worst-case time complexity for step 3, to calculate the number of occurrences for all the RHS symbols of the grammar, is $O(n * n) = O(n^2)$.

4) Worst-case time complexity for step 4 (component function `occur_lhs` and `factor`, lines 19 – 28):

Step 4 calculates the number of occurrences for each symbol on the left hand side of the rules, and calculates the multiplication factors for the symbols on the right hand side of the rules.

Referring to the program in sub-section 3.2, lines 23 and 27 deal with list `grammar` for the number of occurrences for LHS symbols and multiplication factors for RHS symbols by processing elements (symbols) one by one. In each round of the iteration, one symbol is processed and the length of the list decreases by 1 until it reaches 0. Therefore, the component functions `occur_lhs` and `factor` both need $O(n)$ time to deal with the n symbols of the grammar. Then we need to consider the cost for processing one symbol in the two component functions, which occurs in lines 19, 20, 21, 22, 24, 25, and 26.

Lines 19 and 20 are base cases of the component function `occur_lhs`, which return constant values and execute in constant time $O(1)$.

3. Proofs of the ABF Algorithm

In line 22, the component function `occur_lhs` directly returns a constant value with a cost of constant time $O(1)$.

Line 21 calls the function `sumList` to sum the number of occurrences of the RHS symbols, each multiplied by its multiplication factor. The function `sumList` takes $O(n)$ time. The number of occurrences of the RHS symbols are available in step 3, so it needs constant time to retrieve the value (`occur_rhsList !h`).

The calculation for multiplication factor occurs in lines 24, 25, and 26. In line 24, the component function `factor` returns a constant value at a cost of constant time $O(1)$. Line 26 searches `grammar` list for the LHS symbol in the same rule as the current symbol and returns its multiplication factor, which costs $O(n)$ time in the worst case.

The multiplication factor for each LHS symbol is calculated from the number of occurrences of the LHS symbol divided by its subsize (in line 25). Line 25 invokes recursion by calling component function `occur_lhs`. Since the grammar is non-recursive, each recursive call occurs in component function `occur_lhs` and `factor` will call for a different symbol. There are n symbols in the grammar, so the recursion will occur $O(n)$ times.

In step 4, the two component functions `occur_lhs` and `factor` are mutually recursive. The maximum cost for manipulating one symbol in step 4 occurs in lines 21, and 25, or line 26, which costs $O(n \cdot n)$ time in the worst case. Therefore, the worst-case time complexity for step 4 to calculate the number of occurrences for all the LHS symbols and the multiplication factors for the RHS symbols of the grammar is $O(n \cdot n \cdot n) = O(n^3)$.

5) Worst-case time complexity for step 5:

Step 5 goes through all the symbols of the grammar and labels the decision points. There are n symbols in the grammar. The worst-time complexity for step 5 is $O(n)$.

6) Worst-case time complexity for step 6:

Step 6 tabulates the obtained values for the decision points. For a grammar with n symbols, the worst-time complexity for step 6 is $O(n)$.

7) Worst-case time complexity for step 7:

Step 7 involves basic arithmetic operations, which have a worst-case time complexity of $O(1)$.

In summary, the worst-case costs for steps 1 to 7 are: $O(n^3)$, $O(n^4)$, $O(n^2)$, $O(n^3)$, $O(n)$, $O(n)$, and $O(1)$ respectively. The seven steps are executed in sequence in the ABF algorithm. Therefore, the worst-case time complexity for the algorithm ABF is the maximum cost of the seven steps, i.e., $O(n^4)$.

4. SEMANTIC CONSTRAINT GRAMMARS (SCG)

As discussed in sub-sections 1.2.2 and 2.4, several researchers have argued that grammars with lower ABFs are likely to have better recognition accuracy than those with higher ABFs (Hauptmann et. al., 1988), (Young et. al., 1989), (Young, 1990), (Waibel and Lee, 1990), (Edelkamp and Korf, 1998), and (Morimoto and Takahashi, 2008, 2009). In this section, we provide further evidence that this is the case by modifying a grammar to syntactically encode semantic constraints, and thereby reduce language size and ABF, and then we compare the ABFs of the original grammar and the “semantically constrained” version of it with experimental results of speech recognition accuracy which were obtained as part of the candidate’s Master’s work (Shi, 2003b).

We begin by defining context-free grammars and semantic constraint grammars, and provide examples of each of them. We then analyze the two grammars using various grammar metrics including the ABF. We compare the results of the analysis with the results of the experiment conducted as part of the Master’s work. We conclude that encoding semantic constraints in the syntax of a grammar reduces the ABF and increases speech recognition accuracy. This work supports the claim that semantic constraint grammars may be a useful approach in speech recognition grammar design. The work also adds evidence to the claim, made by others, that the ABF is a good indicator of speech recognition accuracy.

4.1 Context-Free Grammars (CFG)

A grammar defines a language by identifying the set of valid sequences of terminals (*sentences* of the language). Formally, a *Context Free Grammar (CFG)* G is a quadruple $G = (N, T, P, S)$, where:

- (1) N is a finite set of *non-terminal* symbols,
- (2) T is a finite set of *terminal* symbols, $N \cap T = \emptyset$,
- (3) P is a finite set of *production rules*,
 $(P: \langle N \rangle = (N \cup T)^*)$,
- (4) S is the *start symbol*, $S \in N$.

A rule in a CFG has a non-terminal symbol representing a single atomic grammatical category on the left-hand side, and a sequence of non-terminals and terminals (words) on the right-hand side (Moore, 1999) and (Amaya *et al.*, 1999). The single non-terminal on the left-hand side of a CFG rule can be freely replaced by the right-side symbols, and this gives rise to the name “Context-Free Grammar” (Blackburn and Striegnitz, 2002).

Figure 4.1.1 shows a sample CFG grammar written in JSGF (Sun, 2000), which defines a language including sentences, such as *a boy opened the door*.

```

<S> = <NP> <VP>;
<NP> = <Det> <N>;
<VP> = <V> <NP>;
<Det> = the | a;
<N> = boy | door | window;
<V> = opened | closed;

```

Figure 4.1.1: a sample CFG

4.2 Semantic Constraint Grammars (SCG)

Language features have been effectively studied and applied in language models to improve speech-recognition performance by reducing the number of possible utterances and prioritizing utterance hypotheses. Takezawa et al. (1991) state that the accuracy of speech recognition heavily depends on the type of linguistic knowledge used. Seneff et al. (1995), Hermannsdottir (1996), Moore (1999), and Harper et al. (2000) claim that “good” language features are necessary to achieve high accuracy in speech recognition with moderate to large vocabularies (hundreds to tens of thousands of words).

Syntax and semantics are two important linguistic components. Syntax defines the way in which linguistic elements (words) are put together to form constituents (as phrases or clauses). Semantics is concerned with meaning. It is possible that a sentence is syntactically correct but semantically incorrect. For example, the sentence `a window closed a door` is syntactically correct, with respect to the grammar above in Figure 4.1.1, because it complies with the syntax of

$$\langle S \rangle = \langle NP \rangle \langle VP \rangle ;$$

However, it does not make sense in the real world. Therefore, it is reasonable to expect that the exclusion of the semantically incorrect utterances (although may be syntactically correct) in a speech application may improve the recognition accuracy.

A Unification Grammar (UG) is an augmented or annotated Context-Free Grammar (CFG) by applying some restriction properties to a CFG in a syntactic notation. With the constraints unified to the grammar, a UG is more expressive and more concise than a traditional CFG in representing semantics (Moore, 1999). With the constraints, a UG can help reduce the system's perplexity, hence improve the recognition accuracy.

The following is an example rule of a UG from (Moore, 1999):

4. Semantic Constraint Grammars (SCG)

S: [tensed=yes] → NP: [person=P, num= N]

VP: [tensed=yes, person=P, num=N]

This example illustrates the use of feature constraints by a UG (such as, person=P, num=N). Thus, the UG guarantees that the person and number features of Noun Phrase (NP) and Verb Phrase (VP) must agree with each other. For example, the sentence *He is a student* is correct by this grammar, whereas the sentence *He are students* is not.

Frost (2002) proposed a Semantic Constraint Grammar (SCG), which encodes semantic constraints directly in the syntax of a traditional Context-Free Grammar (CFG). The integration of semantic constraints in the syntactic rules naturally reduces the language size, therefore should improve speech-recognition accuracy. For example, the query *which moon discovered mars* may be accepted by a simple CFG, which is initially used to define a language for a database query processor, which includes the rule:

```
<question> = which <nounphrase> <verbphrase>;
```

However, in the specified domain, moons cannot discover mars. Therefore, this query is syntactically correct, but semantically incorrect. We can replace the syntactic rule above with a SCG rule as follows:

```
<question> =  
    which <animate_nounphrase> <animate_verbphrase>  
    | which <inanimate_nounphrase> <inanimate_verbphrase>;
```

This SCG rule requires the agreement of `animate_nounphrase` with `animate_verbphrase` and `inanimate_nounphrase` with `inanimate_verbphrase`. Therefore, the example query *which moon discovered mars* will not be accepted by the SCG because the inanimate noun *moon* needs an inanimate verb phrase while *discovered mars* is an

animate verb phrase. In this way, semantically incorrect utterances are excluded by the SCG, and the perplexity of the language is reduced, so that speech-recognition accuracy is improved.

The disadvantage of the SCG is the increase in the size of the grammar, owing to the inclusion of semantic constraints in the syntax. The larger size of the grammar implies more difficulty in grammar design and makes the speech system more difficult to maintain. However, this disadvantage can be overcome to some extent by subdividing a large complex SCG grammar into small SCGs covering smaller domains. This technique is discussed in section 6, with respect to a Public Domain SpeechWeb, in which several applications covering small domains with small SCGs are hyperlinked, so that the user can “browse” a SpeechWeb by navigating through a web of hyperlinked small speech applications.

4.3 An Example of a CFG and a Related SCG

To further explain and compare Context-Free Grammars (CFGs) and Semantic Constraint Grammars (SCGs), we present examples next.

The sample CFG and SCG are constructed for the same domain (the very small subset of the Solar System), with similar vocabularies. However, the SCG defines a smaller language than that defined by the CFG. Although both languages include queries such as `who discovered phobos`, `which moon orbits mars`, etc. However, as discussed in sub-section 4.2, a query such as `which moon discovered mars` is covered by the CFG but is not covered by the SCG.

The example CFG and SCG are written in Java Speech Grammar Format (JSGF) (Sun, 2000).

4.3.1 The example CFG

The example CFG (Figure 4.3.1) defines three types of queries with respect to the solar system and a group of simple greetings as follows:

- (1) The queries starting with the linking verbs, such as `is earth a planet` and `is mars discovered by hall`.
- (2) Questions starting with general question words, such as `does titan orbit mars` or `did bernard discover jupiter`.
- (3) Queries starting with special question words such as: `how many moons orbit jupiter` or `which moons orbit Jupiter`.
- (4) The CFG also includes some simple greetings, such as `hello`, and `goodbye`.

Note that, the following queries will also be accepted by the CFG grammar for their correct syntax, even though they are semantically incorrect:

```
which man orbits titan?
```

```
which moon discovered earth?
```

An example CFG in JSGF format is shown in Figure 4.3.1. We include the rule numbers for reference, although they are not part of the JSGF notation.

4. Semantic Constraint Grammars (SCG)

```
/* solar_CFG.gram */
grammar solar_CFG ;

1. public <s>
    = <linkingvb> <termph> [<transvb> by] <termph>
    | <linkingvb> <termph> <termph>
    | <quest> <sent>
    | (who | what) <verbph>
    | (which | how many) <nouncla> <verbph>
    | <simple>;

2. <sent>
    = <termph> <verbph>;

3. <termph>
    = <stermph>
    | <stermph> (and | or) <stermph>;

4. <stermph>
    = <pnoun>
    | <detph>;

5. <verbph>
    = <transvbph>
    | <intransvb>;

6. <transvbph>
    = (<transvb> | <linkingvb> <transvb> by) <termph>;

7. <intransvb>
    = spin | spins | spun | exist | exists | existed;

8. <transvb>
    = orbit | orbits | orbited | discover | discovers
    | discovered | find | finds | found;

9. <detph> = <det> <nouncla>;

10. <nouncla>
    = <adj> <cnoun>
    | <cnoun>;

11. <cnoun>
```

4. Semantic Constraint Grammars (SCG)

- ```
 = people | planet | moon;

12. <pnoun>
 = bernard | bond | cassini | dollfus | fountain
 | galileo | hall | herschel | huygens | kowal
 | kuiper | arsen | lassell | melotte | nicholson
 | perrine | pickering | earth | jupiter | mars
 | mercury | neptune | pluto | saturn | uranus
 | venus | almathea | ariel | callisto | charon
 | deimos | dione | enceladus | europa | ganymede
 | hyperion | iapetus | io | janus | jupitereighth
 | jupitereleventh | jupiterfourteenth | jupiterninth
 | jupiterseventh | jupitersixth | jupitertenth
 | jupiterthirteenth | jupitertwelfth | luna | mimas
 | miranda | nereid | oberon | phobos | phoebe | rhea
 | saturnfirst | tethys | titan | titania | triton
 | umbriel;

13. <det>
 = a | an | every | one | two | three | four;

14. <adj>
 = red | atmospheric;

15. <linkingvb>
 = is | was | are | were ;

16. <quest>
 = did | do | does;

17. <simple>
 = hello | hi there | how are you | fine, thanks
 | goodbye | bye- bye ;
```

*Figure 4.3.1: example CFG with the domain of the Solar system*

### 4.3.2 The example SCG

As discussed in sub-section 4.2, a SCG specifies semantics as well as syntax by encoding semantic constraints in the syntactic rules. The example SCG given below in Figure 4.3.2 is defined with the same domain (the small subset of the solar system) and a similar dictionary to the CFG in Figure 4.3.1. Similar to the CFG, the SCG also covers three types of queries in the solar system and a group of simple greetings as follows:

- (1) Queries starting with the linking words. For example, `was mars discovered by hall, or is jupiter a planet.`
- (2) Queries starting with general question words, such as, `did hall discover mars, and does titan orbit earth.`
- (3) Queries starting with special question words, such as `how many moons orbit earth, or who discovered jupiter.`
- (4) Simple greetings like `hello` and `goodbye`.

Figure 4.3.2 shows that a SCG grammar requires semantic agreement among the components of each sentence. In particular, some actions can only be initiated by or applied to animate objects and some actions can only initiated by or applied to inanimate objects. Only when the components of an action agree in semantics as well as syntax, is the sentence covered by the SCG grammar as a correct utterance. Therefore, the queries such as `which moon discovered venus` will not be accepted by a SCG recognizer.

The example SCG is given in Figure 4.3.2. The grammar is written in JSGF. The rule numbers are included for reference:

#### 4. Semantic Constraint Grammars (SCG)

---

```
/* solar_SCG.gram */

grammar solar_SCG;

1. public <s>
 = <linkingvb> <termphrase_verbphrase>
 | is <pnoun> <pnoun>
 | is <pnoun> (a|an) <nouncla>
 | is <pnoun> (a|an) <nouncla> or (a|an) <nouncla>
 | <quest> <sent>
 | (who) <animate_verbph>
 | (what) <inanimate_verbph>
 | (which | how many) <nouncla_verbph>
 | <simple>;

2. <termphrase_verbphrase>
 = <nonhuman_termph_planet> <transvb_by_termph>
 | <nonhuman_termph_moon> <animate_transvb> by
 <human_termph>;

3. <transvb_by_termph>
 = <animate_transvb> by <human_termph>
 | <inanimate_transvb> by <nonhuman_termph_moon>;

4. <sent>
 = <human_termph> <animate_verbph>
 | <nonhuman_termph_moon> <inanimate_verbph_active>
 | <nonhuman_termph_planet> <inanimate_verbph_passive>;

5. <nouncla_verbph>
 = <human_nouncla> <animate_verbph>
 | <nonhuman_nouncla_moon> <animate_verbph_passive>
 | <nonhuman_nouncla_planet> <animate_verbph_passive>
 | <nonhuman_nouncla_moon> <inanimate_verbph_active>
 | <nonhuman_nouncla_planet> <inanimate_verbph_passive>;
```

6. `<inanimate_verbph>`  
    `= <inanimate_verbph_active>`  
    `| <inanimate_verbph_passive>;`
7. `<human_stermph>`  
    `= <human_pnoun>`  
    `| <human_detph>;`
8. `<nonhuman_stermph_planet>`  
    `= <nonhuman_pnoun_planet>`  
    `| <nonhuman_detph_planet>;`
9. `<nonhuman_stermph_moon>`  
    `= <nonhuman_pnoun_moon>`  
    `| <nonhuman_detph_moon>;`
10. `<human_termph>`  
    `= <human_stermph>`  
    `| <human_stermph> (and|or) <human_stermph>;`
11. `<nonhuman_termph_planet>`  
    `= <nonhuman_stermph_planet>`  
    `| <nonhuman_stermph_planet> (and|or)`  
    `<nonhuman_stermph_planet> ;`
12. `<nonhuman_termph_moon>`  
    `= <nonhuman_stermph_moon>`  
    `| <nonhuman_stermph_moon> (and|or)`  
    `<nonhuman_stermph_moon>;`
13. `<animate_verbph>`  
    `= <animate_transvbph>;`
14. `<inanimate_verbph_active>`  
    `= <inanimate_transvbph_active>`  
    `| <intransvb>;`
15. `<inanimate_verbph_passive>`  
    `= <inanimate_transvbph_passive>`  
    `| <intransvb>;`

#### 4. Semantic Constraint Grammars (SCG)

---

- 16. <animate\_verbph\_passive>  
    = <linkingvb> <animate\_transvb> by <human\_termph>;
- 17. <animate\_transvbph>  
    = <animate\_transvb>  
        (<nonhuman\_termph\_planet> | <nonhuman\_termph\_moon>);
- 18. <inanimate\_transvbph\_active>  
    = <inanimate\_transvb> <nonhuman\_termph\_planet>;
- 19. <inanimate\_transvbph\_passive>  
    = <linkingvb> <inanimate\_transvb> by  
        <nonhuman\_termph\_moon>;
- 20. <human\_detph>  
    = <det> <human\_nouncla>;
- 21. <nonhuman\_detph\_planet>  
    = <det> <nonhuman\_nouncla\_planet>;
- 22. <nonhuman\_detph\_moon>  
    = <det> <nonhuman\_nouncla\_moon>;
- 23. <nouncla>  
    = <human\_nouncla>  
        | <nonhuman\_nouncla\_planet>  
        | <nonhuman\_nouncla\_moon>;
- 24. <human\_nouncla>  
    = <human\_cnoun>;
- 25. <nonhuman\_nouncla\_planet>  
    = <adj> <nonhuman\_cnoun\_planet>  
        | <nonhuman\_cnoun\_planet>;
- 26. <nonhuman\_nouncla\_moon>  
    = <adj> <nonhuman\_cnoun\_moon>  
        | <nonhuman\_cnoun\_moon>;
- 27. <pnoun>  
    = <nonhuman\_pnoun\_planet>  
        | <nonhuman\_pnoun\_moon>  
        | <human\_pnoun>;

#### 4. Semantic Constraint Grammars (SCG)

---

28. <human\_cnoun>  
    = people;
29. <nonhuman\_cnoun\_planet>  
    = planet;
30. <nonhuman\_cnoun\_moon>  
    = moon;
31. <intransvb>  
    = spin | spins | spun | exist | exists | existed ;
32. <animate\_transvb>  
    = discover | discovers | discovered | find | finds |  
      found;
33. <inanimate\_transvb>  
    = orbit | orbits | orbited;
34. <nonhuman\_pnoun\_planet>  
    = earth | jupiter | mars | mercury | neptune | Pluto  
      | saturn | uranus | venus ;
35. <nonhuman\_pnoun\_moon>  
    = almathea | ariel | callisto | charon | deimos | dione  
      | enceladus | europa | ganymede | hyperion | iapetus  
      | io | janus | jupitereighth | jupitereleventh  
      | jupiterfourteenth | jupiterninth | jupiterseventh  
      | jupitersixth | jupitertenth | jupiterthirteenth  
      | jupitertwelfth | luna | mimas | miranda | nereid  
      | oberon | phobos | phoebe | rhea | saturnfirst  
      | tethys | titan | titania | triton | umbriel ;
36. <human\_pnoun>  
    = bernard | bond | cassini | dollfus | fountain  
      | galileo | hall | herschel | huygens | kowal | kuiper  
      | larsen | lassell | melotte | Nicholson  
      | perrine | pickering;
37. <adj>  
    = red | atmospheric;

```
38. <det>
 = a | an | every | one | two | three | four;
39. <linkingvb>
 = is | was | are | were;
40. <quest>
 = did | do | does;
41. <simple>
 = hello | hi, there | how are you | good, thanks
 | fine, thanks | have a good day | goodbye | bye-bye;
```

*Figure 4.3.2: a sample SCG with the domain of the Solar system*



### 4.4 Analyze the CFG and SCG Using Grammar Metrics

To analyze and compare the CFG and the SCG using the grammar metrics discussed in section 2, we used the two grammars as input to the ABF application described in sub-section 2.9. The results for the grammars given in Table 2.9 are shown in Table 4.4 below:

Table 4.4: grammar metrics of sample CFG and SCG

| Grammar | # of rules | # of symbols | # of Non-terminals | # of Terminals | # of Decision Points | Language Size        | ABF   |
|---------|------------|--------------|--------------------|----------------|----------------------|----------------------|-------|
| CFG     | 17         | 160          | 50                 | 110            | 19                   | $1.73 \cdot 10^{11}$ | 52.42 |
| SCG     | 41         | 262          | 133                | 129            | 53                   | $1.51 \cdot 10^9$    | 33.99 |

Table 4.4 shows that:

(1) The SCG is larger than the CFG.

With the same domain and the same lexicon, the CFG includes 17 rules, and we need 41 rules to define the corresponding SCG. Consequently, the SCG includes more symbols than the CFG. For example, the CFG has 160 symbols, which include 50 non-terminals and 110 terminals. The corresponding SCG uses 262 symbols in definition, among which 133 are non-terminals, and 129 are terminals. In addition, the SCG has more decision points (i.e., 53) than the CFG (i.e., 19).

The reason for the larger grammar of SCG than CFG is that the SCG grammar includes more constraints than the CFG grammar, which needs more rules and symbols (terminals and/or non-terminals) in the definition.

For example, the CFG (Figure 4.3.1) defines a `term phrase` as follows:

```

3. <termph>
 = <stermph>
 | <stermph> (and | or) <stermph>;

```

#### 4. Semantic Constraint Grammars (SCG)

---

However, the SCG (Figure 4.3.2) specifies the `term phrase` with respect to the semantic types (constraints), as follows:

```
10. <human_termph>
 = <human_stermph>
 | <human_stermph> (and|or) <human_stermph>;
11. <nonhuman_termph_planet>
 = <nonhuman_stermph_planet>
 | <nonhuman_stermph_planet> (and|or)
 <nonhuman_stermph_planet>;
12. <nonhuman_termph_moon>
 = <nonhuman_stermph_moon>
 | <nonhuman_stermph_moon> (and|or)
 <nonhuman_stermph_moon>;
```

In the above example, rule 3 of the example CFG (Figure 4.3.1) includes 6 symbols, 4 non-terminals, and 2 terminals while defining a term phrase. However, the example SCG (Figure 4.3.2) needs 3 rules (rules 10, 11, 12), 18 symbols, 12 non-terminals, and 6 terminals to define a term phrase.

Therefore, a SCG may include more symbols (terminals and/or non-terminals) than a CFG with the same vocabulary.

(2) The SCG defines a smaller language than the CFG.

The size of the language defined by the SCG is  $1.51 \cdot 10^9$ , while the language size defined by the CFG is  $1.73 \cdot 10^{11}$ . With the same domain and vocabulary, the CFG defines a language 114 times larger than the SCG.

(3) The ABF of the SCG is smaller than that of the CFG.

The results in Table 4.4 show that, with the same domain and the same lexicon, the ABF of the SCG (i.e., 33.99) is smaller than the ABF of the CFG (i.e., 52.42), which is 35.16% reduction.

#### **4.5 Results from an Experiment Investigating Speech- Recognition Accuracy**

In the Master's work (Shi, 2003b), we conducted a preliminary experiment to investigate the significance of grammar design in speech recognition. In this experiment, six grammars and two people (one English male and one non-English female) were involved. Six grammars were: a semantic grammar, a syntactic grammar, a word-sequence grammar, an extended semantic grammar, an extended syntactic grammar, and an extended word-sequence grammar. Note that, the semantic grammar and the syntactic grammar in the experiment are the same grammars in Figure 4.3.2 and Figure 4.3.1 respectively in this report.

In the experiment, the subjects (people) spoke to the experimental system at a normal speed, pronouncing every word as clearly as possible, like a normal user to a speech recognition system. We also included the training part in the beginning of the experiment so that the subjects were able to get used to the testing system and made their pronunciation acceptable to the system in order to minimize the effect of the order in which the grammars were tested.

The testing utterance inputs were categorized into the following three sets:

- (1) a semantics set, which was composed of the questions that were both semantically and syntactically correct (e.g., `is titania a moon`);
- (2) a syntax set, which consisted of the questions that were only syntactically correct, but semantically incorrect (e.g., `which moon discovered mars`);

- (3) a word-sequence set, which covered the utterances that were neither semantically nor syntactically correct, they were only word sequences (e.g., `moon is discovered mars`).

The experiment was carried out using IBM WebSphere Voice Server SDK on Windows XP (Home edition). The experiment application was written in VoiceXML (VoiceXML Forum, 2004). All experiments were conducted consistently in the same experimental location with the same background noise.

The experimental results were marked as “Correct”, “Incorrect” (“Mis-recognition”), and “Not Recognized”. The experiments were analyzed with respect to each experimental subject for each grammar using each testing utterance set. We have included some of the experimental results from (Shi, 2003b) in sub-section 4.6 of this report and summarize them as follows:

- (1) The semantic grammar has the best recognition accuracy for semantically and syntactically correct utterances. It defines the smallest language, but is the most complicated grammar.
- (2) The syntactic grammar has the mid performance in accuracy, language size, and grammar complexity, among these three types of grammar.
- (3) The recognition accuracy of word-sequence grammars is very low, but word-sequence grammars are the most robust grammar, and may provide some useful information when the user inputs an “unexpected” utterance. The grammar of word sequences is the simplest one, which covers the largest language.

#### **4.6 Comparison of ABFs and Speech Recognition Accuracy Results**

Hauptmann et. al. (1988), Young et al. (1989), Young (1990), Waibel and Lee (1990), Edelkamp and Korf (1998), and Morimoto and Takahashi (2008, 2009) have argued that grammars with lower ABFs are likely to have better recognition accuracy than those with higher ABFs. We have presented the first algorithm to

correctly compute the ABF directly from a grammar in section 2. In addition, we have briefly reviewed the experiment investigating speech recognition accuracy in sub-section 4.5. To further compare the ABFs and speech recognition accuracy, we will review the results for the ABFs and the investigation of speech recognition accuracy in the rest of this sub-section.

We have applied the ABF algorithm in section 2 to calculate the ABFs and have also computed other grammar metrics (i.e., number of rules, number of symbols, number of non-terminals, number of terminals, number of decision points, size of the language). The results of applying the ABF algorithm to nine grammars are given in sub-section 2.9. We present the results from sub-section 2.9 for the grammars that are very similar (with some minor differences in vocabulary) to the grammars used in the recognition accuracy experiments in (Shi, 2003b) and compare the results.

Excerpt of Table 2.9: results of applying the ABF algorithm

| No. | Grammar                     | # of<br>Rules | # of<br>Symbols | # of Non-<br>Terminals | # of<br>Terminals | # of Dec<br>Points | Language<br>Size     | ABF    |
|-----|-----------------------------|---------------|-----------------|------------------------|-------------------|--------------------|----------------------|--------|
| 6   | SCG Grammar<br>Figure 4.3.2 | 41            | 262             | 133                    | 129               | 53                 | $1.51 \cdot 10^9$    | 33.99  |
| 7   | CFG Grammar<br>Figure 4.3.1 | 17            | 160             | 50                     | 110               | 19                 | $1.73 \cdot 10^{11}$ | 52.42  |
| 8   | Word-seq Gram<br>Appendix E | 12            | 184             | 77                     | 107               | 46                 | $9.14 \cdot 10^{19}$ | 188.99 |

The results show that:

- (1) The SCG is the largest among these three grammars.
- (2) The SCG defines the smallest language among these three grammars.
- (3) The ABF of the SCG is the smallest among these three grammars.
- (4) The word sequence grammar defines the largest language among these three grammars.

#### 4. Semantic Constraint Grammars (SCG)

---

- (5) The ABF for the word sequence grammar is the largest among these three grammars.

The experiment of investigating speech recognition accuracy was conducted with respect to subjects (users), grammars, and testing full sentence spoken inputs. The results were recorded as “Correct”, “Incorrect” (i.e., Mis-recognized), and “Not Recognized”. Some experimental results (from Shi, 2003b) are presented as follows:

Table 7.3.2 (1): the “Correct” feature using the semantics set

| Grammars        | Person #1 | Person #2 | Average |
|-----------------|-----------|-----------|---------|
| Semantic (SCG)  | 82.2      | 68.5      | 75.35   |
| Syntactic (CFG) | 80.1      | 52.7      | 66.4    |
| Word Sequence   |           | 12.3      | 12.3    |

Table 7.3.2 (2): the “Incorrect” feature using the semantics set

| Grammars        | Person #1 | Person #2 | Average |
|-----------------|-----------|-----------|---------|
| Semantic (SCG)  | 4.8       | 2.7       | 3.75    |
| Syntactic (CFG) | 15.1      | 13.7      | 14.4    |
| Word Sequence   |           | 59.6      | 59.6    |

The above data show that if the user asks the queries that are both semantically correct and syntactically correct, for both subjects,

- (1) The semantic grammars have the highest correct recognition rate and the lowest incorrect recognition (mis-recognition) rate.
- (2) The word-sequence grammar has significantly less accuracy and highest mis-recognition rate among these three grammars.

In summary, Table 2.9 shows that the ABF and the language size decrease in the following order: word sequence grammar, CFG, and SCG. Not surprisingly, Table 7.3.2 (1) and table 7.3.2 (2) from the experiment show the same order of

these grammars for the increase in speech recognition accuracy and the decrease in the mis-recognition.

This result further supports the claim that SCGs may be a useful approach in speech recognition design by reducing the ABFs and increasing speech recognition accuracy. In addition, this work adds evidence to the claim, made by others, that the ABF is a good indicator of speech recognition accuracy.

#### **4.7 Summary**

In this section we began by describing context-free grammars (CFGs) and Semantic Constraint Grammars (SCGs). We then gave an example of a CFG and a related SCG. We analyzed the two grammars using our software to determine grammar and language size and ABF. We then compared these analytic results with experimental results for the two grammars and a word sequence grammar that had been obtained previously as part of the candidate's Master's. The comparison of ABFs and recognition accuracies of the three grammars lends further evidence to support the claim, made by others, that the ABF is a good indicator of recognition accuracy.

## **5. AUTOMATIC GENERATION OF SPEECH-RECOGNITION GRAMMARS FROM RELATIONAL DATABASE SCHEMAS**

### **5.1 Related Work and Its Shortcomings**

#### *5.1.1 Related work*

Meng and Siu (1999, 2002), Wang and Acero (2003a, 2006) state that creating domain-specific grammars has been a major bottleneck in the development of spoken-language systems. Grammar creation techniques are classified into the following three types: 1) statistical approaches, 2) knowledge-based approaches, and 3) combinations of the two.

Corpus-based statistical approaches have been widely used in grammar creation (Meng and Siu, 1999) and (Caskey et al., 2003). The corpus may be annotated or un-annotated. The grammar creation algorithms automatically capture patterns from the data (Meng and Siu, 2002). The advantages of the statistical approach include that the generated grammar can closely represent the real input data (Meng and Siu, 2002). The main disadvantage of this approach is the very high cost of acquiring the large amount of data needed.

An alternative approach, sometimes referred to as the “knowledge-based approach”, uses experts to design the grammars and subsequently test and modify them (Wang and Acero, 2006). However, this incurs another cost – that of the expert developer with adequate in-depth knowledge of linguistics and the application domain.

Some research has attempted to combine the statistical and knowledge-based approaches (e.g. Wang and Acero, 2001).

In order to reduce the cost of development of speech-enabled applications, researchers have been working for over ten years on tools to facilitate the



process, such as, Pargellis et al. (1999), Gavalda (2000), Glass and Weinstein (2001), Wasinge (2001), and Mané and Levin (2005).

A discussion of this research and a survey of other similar work on automatic generation of speech-recognition grammars is given in Appendix B. The following is extracted from that survey.

### *5.1.2 Shortcomings of related work*

Meng and Siu (1999, 2002) present a statistical data-driven approach for semi-automatic grammar induction from unannotated corpora within a specific domain.

They use an iterative procedure to spatially and temporally cluster the unannotated words from a corpus of sentences in a restricted domain. When words have similar left and right contexts, these words will be grouped together by spatially clustering as they may consist of words with similar semantics. The temporal clustering groups the words with tend to co-occur sequentially. The automatically produced CFG grammars are further manually revised to improve quality.

The shortcomings of this approach are that:

- (1) It is costly to obtain the large domain-specific corpus.
- (2) Even a large amount of data may be “sparse” relative to the target grammar as it may not cover all of language constructions.
- (3) The extensive experiments are costly.
- (4) It needs extra effort for post-processing to manually revise the generated grammars to improve the quality. The post-processing may involve: (a) replacing the non-terminal symbols with semantic meaningful labels, (b) consolidating grammar categories of the same semantic class, and (c) pruning irrelevant non-terminals and terminals.

Xia (2001) presents two systems that automatically generate grammars for natural language processing. One system is named *LexOrg*, which generates grammars automatically from several types of specification. The second system is called *LexTract*, which extracts Lexicalized Tree Adjoining Grammars (LTAGs) and Context-Free Grammars (CFGs) from Treebanks.

The generated grammars in (Xia, 2001) are Lexicalized Adjoining Grammars (LTAGs) which are based on the Tree-Adjoining Grammar (TAG) formalism that is defined by Joshi et al. (1975). Elementary trees are the primitive elements of an LTAG grammar. Each elementary tree is anchored by a lexical item. The trees can be combined by either *substitution* or *adjunction*. Substitution replaces a frontier node with another tree whose top node has the same label. Adjunction inserts an auxiliary tree into the center of another tree.

The shortcomings of the work in (Xia, 2001) are as follows:

- (1) The process of grammar generation using LexOrg or LexTract requires high linguistic expertise.
- (2) The generated LTAGs are not ready to use for speech applications that adapt the widely-used CFG formalism. Although it is possible to build a CFG from an LTAG, it requires special tools and extra work to accomplish this transformation.
- (3) Xia (2001) describes the advantages of LexTract over LexOrg and other traditional grammar construction (e.g., manual grammar writing). However, it requires the access to the Treebanks, which may be a barrier for some users.

Wang and Acero (2001, 2005) present a machine-aided grammar authoring system that combines the knowledge-based and data-driven approaches. This approach uses domain-specific semantics, a library grammar, syntactic constraints, and a small amount of semantically annotated example sentences.

The work in (Wang and Acero, 2001 and 2005) is similar to our work, but differs in some important ways:

- (1) Wang and Acero (2001, 2005) adapt the knowledge-assisted data-driven statistical modeling to author CFG grammars. We analyze and interpret database schemas to generate CFG and SCG grammars.
- (2) Wang and Acero (2001, 2005) use a semantic schema in XML to express domain-specific information. The developer needs to have a very good understanding of XML to define the semantic classes and slots. In our approach, we use a semantic specification to include the domain-specific information for grammar generation. Our approach requires only a basic knowledge of XML.
- (3) In (Wang and Acero, 2001 and 2005), the developer needs to annotate the data with information from the semantic schema in order to reduce the search space. The amount of the annotated training data affects the quality of the learned grammar, which is not necessary in our approach.
- (4) They use a CFG grammar template to generate CFG grammars. We have both CFG and SCG grammar templates, the latter of which provides better speech-recognition accuracy.

From our survey (Appendix B), we have observed that there is no existing approach for creating grammars easily and cheaply from database schemas. We propose a new approach for automatically generating speech-recognition grammars from relational database schemas. In our approach, we attempt to overcome some of the shortcomings existing in other work. For example, we include the work of post-processing in (Meng and Siu, 1999 and 2002) in the built-in grammar-generation system as an optimization component to save developers' effort, expedite the grammar development, and improve the quality of the generated grammar. We will discuss in detail the new approach in the following subsections.

## 5.2 The New Approach

Seneff (1992), and Wang and Acero (2001) have determined that the high-level syntactic structures of many speech-recognition grammars are similar for applications of similar type (e.g., database query applications) and differ only in the lexicon and in syntactically-expressed semantic constraints. Based on such assumption, we propose a new approach for automatically generating speech-recognition grammars from relational database schemas for database query applications.

We build a syntactic grammar template to express the common parts of the recognition grammars for database query processors and create an environment through which grammar developers can generate the application-specific part of the grammar automatically by entering application-specific data.

The new approach consists of the following three key components:

- (1) a built-in Grammar Template,
- (2) a built-in Grammar-Generation Engine,
- (3) a user-defined Semantic Specification.

The Grammar Template and the Grammar-Generation Engine constitute the built-in Grammar-Generation System, which are built-in components that developers can directly use them. The user-defined Semantic Specification provides application-specific information to generate an application-specific speech-recognition grammar, which needs developers' definition.

We consider two types of grammar generation: CFG grammar generation and SCG grammar generation. A SCG is a CFG grammar that encodes semantic constraints directly in the syntactic rules to naturally reduce the language size and therefore should improve speech-recognition accuracy. The CFG grammar generation requires a CFG Grammar-Generation System, including a CFG Grammar Template and a CFG Grammar-Generation Engine. The SCG grammar

generation needs a SCG Grammar-Generation System, including a SCG Grammar Template and a SCG Grammar-Generation Engine. The two Grammar-Generation Systems can share one user-defined Semantic Specification for the same application domain.

The process of automatic CFG/ SCG grammar generation is summarized as follows:

The developer creates the application-specific Semantic Specification and feeds it to the built-in CFG/ SCG Grammar-Generation System. The CFG/ SCG Grammar-Generation Engine analyzes the Semantic Specification and database schemas, interprets them for linguistic information, fills out the built-in CFG/ SCG grammar template, then a new CFG/ SCG grammar is generated automatically.

We will discuss the Semantic Specification and the CFG/ SCG Grammar-Generation System in sub-sections 5.3, 5.4, and 5.5 respectively.

### 5.3 Semantic Specification

The Semantic Specification is created by the developer. It provides necessary application-specific information for Grammar-Generation Systems to build the corresponding CFG and/ or SCG grammars automatically. It is useful in many other aspects in speech-application development. It can be regarded as the specification for a language-enabled application. Once the Semantic Specification is defined, grammar and application logic development can proceed simultaneously according to the specification. A Semantic Specification also contributes to the dialog design and management of the speech application (Wang and Acero, 2001).

To create the Semantic Specification, we borrow some concepts from ER Modeling (Entity-Relationship Modeling), where a table is considered equivalent of an *entity type*. Therefore, we are able to describe the properties for an entity

## 5. Automatic Generation of Speech Grammars from Relational Database Schema

---

type using columns of the table in the Semantic Specification. The activities and relationships among entity types are examined among tables.

The Semantic Specification is expressed using natural English language. XML tags (W3C, 2009) are used as delimiters. For example, the domain-related information about `people's last name` is expressed in the Semantic Specification as follows:

```
<description>
 People's last name is LastName </description>
```

Where `<description>` and `</description>` are delimiter tags in XML format. `People` is a table's name, `LastName` is a column's name of the table `People`. `People` is also an entity type from the point of view of a developer. It represents a semantic class. We consider each table as a *semantic type* in our approach. Details about semantic types will be discussed in subsection 5.5.

A Semantic Specification mainly contains the following three parts:

- (1) Basic database information, such as database vendor, database name, table names, and table aliases.
- (2) Database schema description, which describes the properties (columns) for each table (entity type).
- (3) Definition of the activities/ relationships among tables (entity types) using XML expressions.

For the sake of simplicity, we take an example database with three tables: `People`, `Planet`, and `Moon`. Each table has two columns. The domain is a small subset of the solar system, which is the same as that for the example CFG/ SCG in section 4. The database schema is shown as follows in Figure 5.3:

## 5. Automatic Generation of Speech Grammars from Relational Database Schema

---

- 1) People(*LastName*, *FirstName*)
- 2) Planet(*PlanetName*, *PlanetColor*)
- 3) Moon(*MoonName*, *MoonColor*)

*Figure 5.3: example Database schema for the Solar system*

Regarding a table as an entity type, the properties of an entity type are described in natural English language using the table name and its column names in Figure 5.3 (1). Table names and column names are in italic font.

- 1) *People's* last name is *LastName*
- 2) *People's* first name is *FirstName*
- 3) *Planet's* name is *PlanetName*
- 4) *Planet* is *PlanetColor*
- 5) *Moon's* name is *MoonName*
- 6) *Moon* is *MoonColor*

*Figure 5.3 (1): properties of entity types*

The possible activities among the tables (entity types) in the specific domain are described in Figure 5.3 (2). The table names and column names are in italic font. Here and hereafter, the description of activities and/or relationships in the domain mainly focuses on describing the activities and/or relationships between entities, and ignores the agreement of person and number in English sentence expression.

- 1) *People* discover *planet*
- 2) *People* discover *moon*
- 3) *Moon* orbit *planet*
- 4) *Planet* exist
- 5) *Moon* exist
- 6) *Planet* spin
- 7) *Moon* spin

*Figure 5.3 (2): possible activities/ relationships among tables*

Usually, a database is designed using meaningful symbols for database schema. It is also quite common that abbreviations or other symbols are used as table names or column names in database design. Since we regard a table as an entity type and we would like a meaningful name for an entity type, the user needs to provide meaningful aliases for each table in the Semantic Specification. We use table names and column names for data retrieval from the database. The meaningful table aliases will be more convenient for automatically generating grammars for easy reading. In this thesis report, we may use “table name” instead of “table alias” for easy understanding, because they can be easily exchanged with each other when needed.

In reality, we distinguish animate from inanimate in language expression. It emphasizes that some actions can be initiated by the animate but not by the inanimate, and vice versa. In Figure 5.3 (2), the activities and relationships are described without any ambiguities. Meanwhile, we need to specify the descriptions for *wh-questions*. The reason is that, some *wh-question* words, like *who*, *whom*, and *whose*, can only be applied to the animate, not the inanimate, and some *wh-question* words, like *what*, can only be associated with the inanimate in some situation. Therefore, we include the situations of using the *wh-question* words in the Semantic Specification to further strengthen the semantic constraints.

With the above concerns, i.e.:

- 1) database information,
- 2) description of properties of entity types (Figure 5.3 (1)),
- 3) activities and/or relationships among entity types (Figure 5.3 (2)),  
and
- 4) XML tags,



## 5. Automatic Generation of Speech Grammars from Relational Database Schema

---

we have the following example Semantic Specification in Figure 5.3 (3).

## 5. Automatic Generation of Speech Grammars from Relational Database Schema

---

```
<!------- part 1: Basic database information ----->
<database>
 <DB_vendor> ORACLE </DB_vendor>
 <DB_name> solar_system </DB_name>
</database>
<!-- list table names (entity types) with their aliases
(symbols after "AS"), which are more meaningful in natural
language -->
<table>
 <table_name> people AS people </table_name >
 <table_name> planet AS planet </table_name >
 <table_name > moon AS moon </table_name >
</table >

<!------- Part 2: description of the database schema ----->
<!-- describe the properties for each table (entity type)-->
<Property>
 <People>
 <description>
 People's last name is LastName
 </description>
 <description>
 People's first name is FirstName
 </description>
 </People>
 <Planet>
 <description>
 Planet's name is PlanetName
 </description>
 <description>
 Planet is PlanetColor
 </description>
 </Planet>
</Property>
```

## 5. Automatic Generation of Speech Grammars from Relational Database Schema

---

```
 </description>
 </Planet>
 <Moon>
 <description>
 Moon's name is MoonName
 </description>
 <description>
 Moon is MoonColor
 </description>
 </Moon>
</Property>

<!-------Part 3: definition of the activities/
relationships among tables (entity types) ----->
<Activity>
 <description> People discover planet </description>
 <description> Who discover planet </description>
 <description> People discover moon </description>
 <description> Who discover moon </description>
 <description> Moon orbit planet </description>
 <description> What orbit planet </description>
 <description> What does moon orbit </description>
 <description> Planet exist </description>
 <description> Moon exist </description>
 <description> What exist </description>
 <description> Planet spin </description>
 <description> Moon spin </description>
 <description> What spin </description>
</Activity>
```

*Figure 5.3 (3): example Semantic Specification*

## 5. Automatic Generation of Speech Grammars from Relational Database Schema

---

Note that, the database vendor and name are available for a given database. Table names are available with the given database. Regarding tables as entity types, the activities and/or relationships upon them can be discussed. With good knowledge of the application, a developer will be able to summarize the possible activities and/or relationships among the tables (entity types) within the domain.

Therefore, given the database schema, a developer can create the Semantic Specification in a short time with little difficulty. The reasons are further summarized as follows:

1. XML (eXtensible Markup Language) (W3C, 2009) is a commonly-used format by a developer to represent data. Its basic syntax is often used to share information among computers, applications, and organizations. In addition, in our Semantic Specification, we mainly use the XML tags to delimit the English descriptions, which does not require extensive knowledge of XML.
2. The basic database-related information, such as database vendor, database names, table names, and column names (part one of the Semantic Specification), is easily obtained for a given database.
3. The Semantic Specification is based on the database schemas rather than the data in the database. The size of a database schema is much smaller than that of data in the database.
4. The Semantic Specification authoring is language independent in the sense that it does not specify linguistic expressions. Therefore, it is easy for a developer with good knowledge of an application to define a Semantic Specification.
5. The description of the relationships and activities among tables is similar to ER Modeling (Entity-Relationship Modeling) or UML representation (Unified Modeling Language representation) in software engineering, which are familiar concepts to many software developers.

## 5. Automatic Generation of Speech Grammars from Relational Database Schema

---

From the point of view of ER Modeling, a table is equivalent to an entity type. In addition, it is reasonable to assume that the developer knows something about the specific application domain. Then, by considering a table as an entity type, a developer will be able to define the activities and/or relationships upon the tables (entity types) in the specific domain.

The Semantic Specification has some similarity to UML representation. In a UML representation, a *table* is equivalent to a *class*. Relationships between tables in a Semantic Specification are expressed as relationships between classes in a UML representation.

The comparison among the Semantic Specification, ER Modeling, and UML representation is shown in Table 5.3.

Table 5.3: comparison among the Semantic Specification, ER Modeling, and UML Representation

Semantic Specification	ER Modeling	UML Representation
Table	Entity Type	Class
Column	Attribute	Attribute
Activities/ relationships between tables	Activities/ relationships between entity types	Behavior/ Relationships between classes

#### 5.4 CFG Grammar-Generation System

The CFG Grammar-Generation System includes two main components, a CFG Grammar Template and a CFG Grammar-Generation Engine.

CFG Grammar-Generation Engine analyzes the user-defined Semantic Specification and database schemas, interprets the linguistic information, fills out the built-in CFG Grammar Template, then generates a complete CFG grammar.

In sub-sections 5.4 and 5.5, we will take a small subset of the solar system as an example domain to demonstrate how CFG and SCG grammars are generated automatically from relational database schemas with a given user-defined Semantic Specification. The reason for taking the small subset of the solar system as the example domain is that we have used this domain to illustrate the example CFG and SCG grammars in sub-section 4.3. The automatically generated CFG and SCG grammars are in Appendix G and Appendix H respectively.

The basic concept of generating CFGs and SCGs is the same. Our approach assumes that the high-level syntactic structures of many speech-recognition grammars are similar for applications of similar type (e.g., database query applications) and only differ in the vocabulary and in syntactically-expressed semantic constraints. Therefore, we can extract the common parts of the grammars as a grammar template and fill out the grammar template with the application-specific information to generate the application-specific grammar.

In the process of grammar generation, the main task that needs the developer's attention is to build the Semantic Specification to describe the application-specific information, including basic database information, description of database schemas, relationships and activities among tables. With the same domain, the Semantic Specifications for a CFG grammar and a SCG grammar are the same. We show later how semantic constraints are further used in the SCG.

The Grammar-Generation System (CFG/ SCG) is a customized built-in system for automatic CFG/ SCG grammar generation, which consists of a CFG/ SCG Grammar Template and a Grammar-Generation Engine.

To generate a CFG/SCG grammar automatically, the developer calls the CFG/ SCG Grammar-Generation Engine to interpret the user-defined Semantic Specification and fill out the CFG/ SCG Grammar Template to construct a new application-specific CFG/ SCG grammar.

### *5.4.1 The CFG Grammar Template*

As discussed in sub-section 5.2, a grammar template defines the common parts of speech-recognition grammars for one type of speech application. It is a built-in component of the CFG Grammar-Generation System. The developer does not need to know the CFG grammar template. With the built-in CFG Grammar Template, CFG Grammar-Generation Engine, and the user-provided Semantic Specification, a new application-specific CFG Grammar will be constructed automatically.

Figure 5.4.1 is an example CFG grammar template for database query applications.

## 5. Automatic Generation of Speech Grammars from Relational Database Schema

---

```
/* CFG_template.jsgf */

grammar CFG_template ;

public <s>
 = <linkingvb> <termph> [<transvb> by] <termph>
 | <linkingvb> <termph> <termph>
 | <quest> <sent>
 | (who | what) <verbph>
 | (which | how many) <nouncla><verbph>
 | <simple>;

<sent>
 = <termph> <verbph>;

<termph>
 = <stermph>
 | <stermph> (and | or) <stermph>;

<stermph>
 = <pnoun>
 | <detph>;

<verbph>
 = <transvbph>
 | <intransvb>;

<transvbph>
 = (<transvb> | <linkingvb> <transvb> by) <termph>;

<detph>
 = <det> <nouncla>;

<nouncla>
 = <adj> <cnoun>
 | <cnoun>;
```

*Figure 5.4.1: CFG grammar template*

Where <s> is the start symbol of the grammar, <quest> stands for question, <sent> for sentence, <nouncla> for noun clause, <verbph> for verb phrase, <termph> for term phrase, <stermph> for single term phrase, <transvbph> for transitive verb phrase, <intransvb> for intransitive verb phrase, and



## 5. Automatic Generation of Speech Grammars from Relational Database Schema

---

<det<sub>ph</sub>> for determiner phrase. This CFG grammar template only works for the database query applications. The grammar template needs further modifications for other types of speech applications.

### 5.4.2 The CFG Grammar-Generation Engine

The CFG Grammar-Generation Engine is the other important component of the CFG Grammar-Generation System, which accomplishes the task of reading the Semantic Specification, analyzing the database schemas, interpreting the linguistic information for grammar generation, filling out the CFG Grammar Template, and outputting the newly-built CFG Grammar.

The CFG Grammar-Generation System is implemented on PC (Processor: 2.0 GHZ, Memory: 3GB, Hard Drive: 250GB) with Windows XP (Home Edition) operating system, using the Java programming language (JSDK 1.4.2) as the development tool, MySQL as the database management system, and JDBC technology to connect the database in the Java programming language.

The process of generating the CFG grammar involves the following six steps:

Step 1: read in the Semantic Specification.

The domain-specific Semantic Specification is fed to the CFG Grammar-Generation Engine and tokenized. Basic Database information is obtained. The Database information for the example solar system is shown in Table 5.4.2 (1):

Table 5.4.2 (1): database information

DB_Vendor	DB_name
ORACLE	Solar_system

## 5. Automatic Generation of Speech Grammars from Relational Database Schema

---

The table names and the corresponding aliases are shown in Table 5.4.2 (2):

Table 5.4.2 (2): table names and aliases

Table_Id	Table_name	Table_alias
1	People	People
2	Planet	Planet
3	Moon	Moon

Step 2: connect the database.

With the basic database information (e.g., database vendor and database name), the CFG Grammar-Generation Engine selects the corresponding connection driver to connect the database. In this thesis work, the CFG / SCG grammars can be generated from the following three types of databases: Microsoft Access database, Oracle database, and MySQL database. The source code for database connections in java programming language is in Appendix E.

Step 3: analyze the database schemas.

Normalization is the process of efficiently organizing data in a database with the goals of eliminating redundant data and ensuring that data dependencies are correct. It is reasonable to assume that the database for grammar generation meets the minimum criteria of First Normal Form (1NF), which includes only atomic values in each field. Therefore, the data in each field of the database will be further analyzed and included as terminals in the newly-generated grammars.

The second part of the Semantic Specification (Figure 5.3 (3)) (i.e., `<Property> ...</<Property>` section) describes the properties of the tables. In fact, it is a more-detailed description of the database schema. Tables are regarded as entity types. Each table is listed under the `<property>` section using table names as tags, such as `<People> </People>`. Columns of the

## 5. Automatic Generation of Speech Grammars from Relational Database Schema

---

table are described as properties of the entity type (table), and are listed under the corresponding table name using the tags of `<description>` `</description>`.

Using a simple language to describe the properties lowers the requirement of the linguistic knowledge for a developer, thus eases the developer's work in addressing the Semantic Specification for automatic grammar generation. The Grammar-Generation Engine analyzes the Semantic Specification and the database schema, then interprets the linguistic information for CFG grammar generation.

In many linguistic grammars, words are classified based on the following eight parts of speech: verb, noun, pronoun, adjective, adverb, preposition, conjunction, and interjection (MacFadyen, 2010). To fill out the CFG template and build up the CFG grammar, the CFG Grammar-Generation Engine needs information of the parts of the speech of the data in database, which can be obtained by analyzing the database schemas that are described in the Semantic Specification.

In the following part of this section, we discuss the eight parts of speech and explain how CFG Grammar-Generation Engine obtains linguistic information from the Semantic Specification and fills out the CFG template to construct the CFG grammar.

However, a database may not include instances of the eight parts of speech. For example, the database may not include the words such as `of` or `the`. Therefore, the CFG Grammar-Generation Engine has to follow the syntactic clues and comply with syntactic rules in grammar generation.

### (1) Part of speech - verb

A verb or compound verb asserts something about the subject of the sentence and expresses actions, events, or states of being (MacFadyen, 2010). The verb

or compound verb is the critical element of the predicate of a sentence. In syntax, a *transitive verb* is a verb that requires a direct object to complete its meaning. Verbs that do not require an object are called *intransitive*.

In CFG grammar generation, we distinguish transitive verbs from intransitive verbs. For example, in the description `moon orbit planet`, the verb `orbit` is a transitive verb. In sentence `moon spin`, the verb `spin` is an intransitive verb.

We will discuss more about verbs, relationships, and activities among entities later in step (4) of this section.

### (2) Part of speech - noun

A noun can occur as the main word in the subject of a clause, the object of a verb, or the object of a preposition (MacFadyen, 2010).

There are many types of nouns in linguistics. We use *proper noun* and *common noun* in the CFG template. *Proper nouns* include the names of people, days of the week, months, historical documents, institutions, organizations, religions, and their adherents. A *common noun* is a noun referring to a person, place, or thing in a general sense (MacFadyen, 2010).

For example, in the description, *People's first name is FirstName*, the CFG Grammar-Generation Engine determines that `People` is a table name, `FirstName` is a column name, and the data in column `FirstName` can be used as proper nouns in a sentence. So, the CFG Grammar-Generation Engine accesses the database, retrieves all the data in column `FirstName`, and appends them to the definition rule of the non-terminal `<pnoun>` (stands for *proper noun*) in the CFG template (Figure 5.4.1). The *definition rule* of a non-terminal is a rule with this non-terminal on the left hand side, terminals and/or non-terminals in the form of alternative(s) and/ or sequence(s) on the right hand side of the rule to define the non-terminal. For example, the data from column

## 5. Automatic Generation of Speech Grammars from Relational Database Schema

---

<Firstname> will be appended to the definition rule of the non-terminal <pnoun> as follows:

```
<pnoun> = bernard | bond | cassini | dollfus;
```

While the CFG Grammar-Generation Engine proceeds with the Semantic Specification file, the proper nouns, which are data from corresponding columns of the database, are appended to the syntax rule defining the non-terminal (<pnoun>) in the CFG template (Figure 5.4.1).

The table names, i.e., the entity types, are considered as *common nouns*. They are added to the rule for common nouns (<cnoun>) in the CFG template (Figure 5.4.1) as follows:

```
<cnoun> = people | planet | moon;
```

### a) Determiner

A determiner is a word or affix that belongs to a class of noun modifiers that expresses the reference of a noun, including quantity (MacFadyen, 2010).

For example, Hall discovered *three* moons, the word *three* is a determiner. We include determiners (i.e., <det>) in the CFG grammar generation. The example definition of determiners is as follows:

```
<det> = a | an | every | one | two | three | four;
```

### (3) Part of speech - adjective

An adjective modifies a noun or a pronoun by describing, identifying, or quantifying words. An adjective usually precedes the noun or the pronoun that it modifies (MacFadyen, 2010).

Refer to the example Semantic Specification in Figure 5.3 (3), in the description *Planet is PlanetColor*, the CFG Grammar-Generation Engine recognizes

## 5. Automatic Generation of Speech Grammars from Relational Database Schema

---

that `Planet` is a table name, `PlanetColor` is a column name. The data from column `PlanetColor`, such as `red` and `blue`, is retrieved and appended to the rule for the non-terminal `<adj>` (i.e., adjective) in the CFG template (Figure 5.4.1), as follows:

```
<adj> = red | blue;
```

### a) Article

An article combines with a noun to indicate the type of reference being made by the noun. There are three articles in the English language, namely *the*, *a*, and *an*. Some resources consider there are two articles, which are *the* and *a/an*. Among the classical parts of speech, articles are considered a special category of adjectives (Lynch and Brizee, 2010). In our grammars, we use the identifier "*determiner*" to include the words "*a*", "*every*", "*the*", "*one*", "*two*" etc, This categorization is common in Computational Linguistics"

### (4) Part of speech - adverb

An adverb is a word that can modify a verb, an adjective, another adverb, a phrase, or a clause, except nouns. An adverb is used to indicate manner, time, place, cause, or degree. It can answer questions such as *how*, *when*, *where*, *how much* (MacFadyen, 2010).

For example, the word *fast* is an adverb in the sentence *Planet spin fast*. Adverbs are not included in our example application.

The words of the following four parts of speech: *preposition*, *pronoun*, *conjunction*, and *interjection*, may or may not appear in the database. The CFG Grammar-Generation Engine has to compose the CFG grammar by following the syntactic clues and syntactic constraints.

### (5) Part of speech - preposition

A preposition is used to link nouns, pronouns, and phrases to other words in a sentence. The word or phrase that the preposition introduces is called the *object* of the preposition. A preposition usually indicates the temporal, spatial, or logical relationship of its object to the rest of the sentence (MacFadyen, 2010).

For example, in the sentence `There are minerals on Mars`, the word `on` is the preposition.

### (6) Part of speech - pronoun

A pronoun is a word that can replace a noun or another pronoun. Pronouns can make the sentences less cumbersome and less repetitive (MacFadyen, 2010). Example pronouns are like `he`, `you`, `we`, and so on.

### (7) Part of speech - conjunction

Conjunctions are used to link words, phrases, and clauses (MacFadyen, 2010). For example, in the sentence `Tom and Jerry are friends`, the word `and` is a conjunction.

### (8) Part of speech - interjection

An interjection is a word added to a sentence to convey emotion. It is not grammatically related to any other part of the sentence. Usually, an interjection is followed with an exclamation mark (MacFadyen, 2010).

Step 4: extract the activities and relationships.

Verbs are perhaps the most important part of composing a sentence, expressing ideas, describing an activity of an object, or a relationship between objects. They are a critical element of the predicate of a sentence. Therefore, verbs act as a significant role in grammar generation.

## 5. Automatic Generation of Speech Grammars from Relational Database Schema

---

The second part (`<Property> </Property>`) of the Semantic Specification (Figure 5.3 (3)) describes the database schema information concerning tables and columns where columns are regarded as the properties of tables. However, it does not provide any information about the activities of the entity type itself or the relationships among entity types (tables). Usually, within an application domain, the entities do not stand alone. They are related by some relationships or activities to one another, which are discussed in the (`<activity> ...</activity>`) part of the Semantic Specification (Figure 5.3 (3)).

For example, in the small subset of the solar system example, the entity types `planet` and `moon` are related by the activity `orbit`, e.g., `moon orbit planet` (Figure 5.3 (3)). The system determines that `orbit` is a *transitive verb*. A *transitive verb* requires a subject and an object. A transitive verb describes an action that is initiated by the subject with the object as a receiver. In other words, a transitive verb requires an object to complete the action, which is not required by an *intransitive verb*.

While interpreting the Semantic Specification, the Grammar-Generation Engine analyzes the activities/ relationships between entity types (tables). Meanwhile, it recognizes that the subject and the object of the activity/ relationship description are both tables. For the above example (`moon orbit planet`), the Grammar-Generation Engine interprets the verb `orbit` is a transitive verb, `moon` and `planet` are tables. The word `orbit` is added to the definition rule of the non-terminal `<transvb>` (i.e., transitive verb) as follows:

```
<transvb> = orbit;
```

Since tables are regarded as entity types, the table aliases (`moon` and `planet`) are regarded as *common nouns* and put into the definition of the non-terminal `<cnoun>` (i.e., common noun), as follows:



## 5. Automatic Generation of Speech Grammars from Relational Database Schema

---

```
<cnoun> = moon | planet;
```

In addition, an entity may accomplish some activities by itself, without any other entities involvement. Examples are: `planet spin` and `moon exist`. In such cases, the verbs do not require or cannot take any object. Therefore, we assert that such types of verbs, e.g., `spin` and `exist`, are *intransitive verbs*. The CFG Grammar-Generation Engine will append them to the definition rule of the non-terminal `<intransvb>` (i.e., intransitive verb) as follows:

```
<intransvb> = spin | exist;
```

In this way, the activities and relationships within the domain are analyzed and the verbs are appended to the definition rules of the corresponding non-terminals for transitive verbs or intransitive verbs.

Step 5: fill out the CFG Grammar Template.

The process of filling the CFG template takes place through the whole process of CFG grammar generation. As we have seen, while the CFG Grammar-Generation Engine interprets the database schema, it recognizes parts of speech from the database schema, and appends the data to the definition rules of the corresponding non-terminals, such as `<pnoun>`, `<det>`, `<transvbph>`, `<intransverbph>`, `<adj>`, and `<adv>`. Meanwhile, table aliases are regarded as entity types and appended to the definition of common noun (i.e., `<cnoun>`).

While the grammar generation process comes to the part of the description of activities and relationships in the Semantic Specification, the CFG Grammar-Generation Engine recognizes the transitive verbs and intransitive verbs, and puts them into the corresponding rules for non-terminal such as `<transvb>` (for transitive verbs) and `<intransvb>` (for intransitive verbs).

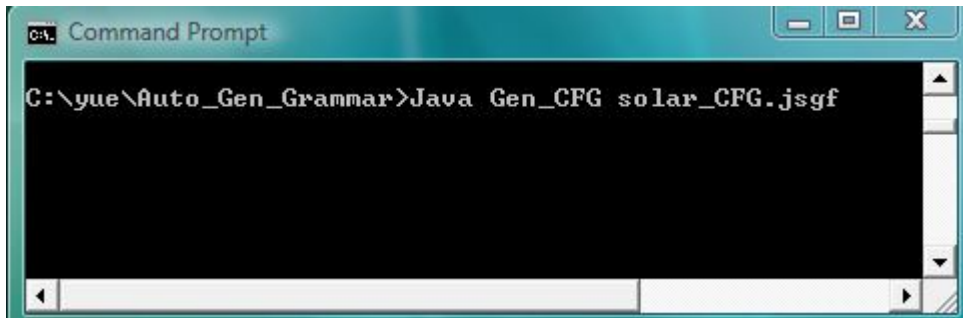
## 5. Automatic Generation of Speech Grammars from Relational Database Schema

---

The database may not include all the information for constructing the CFG grammar. For example, the words `of` and `that` may not show up in the database. The CFG Grammar-Generation Engine needs to fill out the grammar template by following the syntactic rules, constraints, and clues to build the complete CFG grammar.

Step 6: output the CFG grammar.

While calling the CFG Grammar-Generation Engine, the user can specify the location and the name of the output CFG file in the command line. The screenshot in Figure 5.4.2 is an example command to generate a CFG grammar.



*Figure 5.4.2: screenshot - example command to generate a CFG grammar*

Note that, `Gen_CFG` is the name of the CFG Grammar-Generation Engine. `solar_CFG.jsgf` is the name of the new CFG grammar which is saved in the same directory of the CFG Grammar-Generation System. The new CFG Grammar (`solar_CFG.jsgf`) is ready for use in a speech application for database queries with the domain of the small subset of the solar system.

### 5.5 The SCG Grammar-Generation System

Similar to the CFG Grammar-Generation System, the SCG Grammar-Generation System also includes two main built-in components, the SCG Grammar Template and the SCG Grammar-Generation Engine. With a similar workflow to the CFG Grammar Generation, the SCG Grammar-Generation Engine analyzes the user-

defined Semantic Specification and the database schemas, fills out the SCG Grammar Template, and generates a complete SCG grammar for speech applications with the specified domain.

The process of generating a SCG is similar to the process of generating a CFG. The significant difference between a CFG and a SCG is that the CFG is concerned only with the correctness of syntax, while the SCG also encodes semantic constraints. Therefore, the SCG Grammar-Generation System includes more semantic information than CFG Grammar-Generation System.

In this section, we use the same domain as that in the CFG grammar generation in section 5.4 (i.e., the small subset of the solar system) so that we can make a comparison between the two processes of automatic grammar generation. That also makes it possible for further analysis (sub-section 5.6) the two newly generated grammars (CFG and SCG) and the example manually scripted CFG and SCG grammars (in sub-sections 4.3.1 and 4.3.2).

### 5.5.1 The SCG Grammar Template

In the SCG grammar template in Figure 5.5.1, we introduce *semantic types* in order to specify semantic agreement between the subject, predicate, and object in a sentence. *Semantic types* classify words by semantics instead of syntactic function. In this thesis work, a *semantic type* is usually classified by an entity type (table). All of the data in the same table has the same semantic type, which is, represented by the table name. For the example domain in section 5.3, we classify the objects in the small subset of the solar system into the following three types: 1) `people`, 2) `planet`, and 3) `moon`.

In addition to the part of speech, each word (e.g., data in the database) in the domain is associated with a semantic type. A sentence is considered correct only when it complies with the syntactic constraints and the semantic type constraints. This means that only a word of the semantic type `people` can initiate the action

## 5. Automatic Generation of Speech Grammars from Relational Database Schema

---

discover. Therefore, a sentence such as `Bond discovered Mars` is covered, but `Mars discovered Jupiter` is not covered by the SCG.

In the SCG Grammar Template in Figure 5.5.1, we use `type_k` ( $1 \leq k \leq N$ ) to denote a semantic type, and keep the agreement between semantic types by using `type_k`. In the solar system example, there are three semantic types in the specified domain (i.e.,  $N = 3$ ). We assign `type_1` to `people`, `type_2` to `planet`, and `type_3` to `moon`. To generalize the SCG grammar template, we simply use `type_i` to direct the SCG Grammar-Generation Engine to list all the possible semantic types in the domain, `type_1, ..., type_N`.

In addition, we introduce `type_who`, `type_what`, `type_which`, and `type_how_many` to categorize the words: `who`, `what`, `which`, and `how many`.

## 5. Automatic Generation of Speech Grammars from Relational Database Schema

---

Figure 5.5.1 below is the example SCG Grammar Template.

```
/* ***** */

/* SCG_template.jsgf */
/* Assuming there are n semantic types, denoted as $type_k$
 ($1 \leq k \leq n$).
 Using $type_i$ to list all the semantic types $type_1$,
 $type_2$, ..., $type_n$.
 Use $type_k$ to specify some specific semantic type */
/* ***** */

grammar scg_template ;

public <s>
 = <linkingvb> <termph_verbph>
 | <quest> <sent>
 | is <pnoun> <pnoun>
 | is <pnoun> (a|an) <nouncla>
 | is <pnoun> (a|an) <nouncla> or (a|an) <nouncla>
 | (who) <verbph_type_who>
 | (what) <verbph_type_what>
 | (which) <nouncla_verbph_type_which>
 | (how many) <nouncla_verbph_type_how_many>
 | <greetings>;

<termph_verbph>
 = <termph_type_suc> <transvb_type_i> by <termph_type_pre>;

<sent>
 = <termph_type_i> <verbph_type_i>;

<termph_type_k>
 = <stermph_type_k> | <stermph_type_k> (and|or)
 <stermph_type_k>;

<stermph_type_k>
 = <pnoun_type_k> | <detph_type_k>;

<detph_type_k>
 = <det> <nouncla_type_k>;

<nouncla>
 = <nouncla_type_i>;
```

## 5. Automatic Generation of Speech Grammars from Relational Database Schema

---

```
<nouncla_type_k>
 = <cnoun_type_k>
 | <adj_type_k> <cnoun_type_k>;

<verbph_type_k>
 = <transvb_type_k> <termph_type_suc>
 | <intransvb_type_k>;

<nouncla_verbph_type_k>
 = <nouncla_type_pre> <verbph_type_k>
 | <nouncla_type_suc> <verbph_passive_type_k>;

<verbph_passive_type_k>
 = <linkingvb> <transvb_type_k> [by <termph_type_pre>];

<pnoun>
 = <pnoun_type_i> ;

<cnoun>
 = <cnoun_type_i> ;
```

*Figure 5.5.1: SCG grammar template*

Note that, some parts of the SCG grammar template are the same as and have the same meanings as that in the CFG grammar template, such as <s>, <quest>, <sent>, and <nouncla>.

### 5.5.2 SCG Grammar-Generation Engine

Similar to the CFG Grammar-Generation Engine, the SCG Grammar-Generation Engine first takes the user-defined Semantic Specification as input and analyzes it to obtain the corresponding syntactic and semantic information. The SCG Grammar-Generation Engine then fills out the SCG grammar template, and then builds and outputs the new SCG grammar for use in the speech application.

The SCG Grammar-Generation System is implemented on PC (Processor: 2.0 GHZ, Memory: 3GB, Hard Drive: 250GB) with Windows XP (Home Edition) operating system, using the Java programming language (JSDK 1.4.2) as the development tool, MySQL as the database management system, and JDBC technology to connect the database in the Java programming language.

The steps of generating the SCG grammar are similar to that of generating a CFG grammar. However, encoding of semantic constraints in the syntax makes the SCG grammar generation more complicated than the CFG grammar generation. In the following discussion of the process of building a SCG, we will briefly review the same steps as the CFG grammar generation, and elaborate the differences from the CFG grammar generation.

Step 1: read in the Semantic Specification.

Same as that in sub-section 5.4.2.

Step 2: connect the database.

Same as that in sub-section 5.4.2.

Step 3: analyze the database schemas.

Using the same methodology of sub-section 5.4.2, the SCG Grammar-Generation Engine recognizes the parts of speech of the words (e.g., data in database). However, different from the CFG grammar, the SCG grammar has to comply with semantic constraints in addition to syntactic rules. Therefore, the SCG Grammar-Generation Engine needs also to consider the semantic types for each word to satisfy the semantic constraints. Therefore, each word in the database is associated with two features: part of speech and semantic type.

To classify and obtain the information of semantic types, we review the description of database schemas, activities and relationships among entity types in the Semantic Specification. For example, in the Semantic Specification (Figure 5.3 (3)), in the third part (<Activity>...</Activity>), one activity is expressed as follows:

```
<description> People discover planet </description>
```

The statement between the tags <description> and </description> is the same as in the action *discover* using semantic types in section 5.5.1, i.e., *people* discover *planet*, where *people* and *planet* are semantic types.

Note that, while we regard a table as an entity type, we have already considered the data of a table as being in the same semantic category. Thus we can determine that the tables (entity types) of the database can work as the role of semantic types in SCG grammar generation. Therefore, the semantic types are easily obtained by recognizing table names (aliases).

While the SCG Grammar-Generation Engine fills out the SCG grammar template, it not only considers the part of speech of the word, but also puts the word to the right category by its semantic type. For example, the entity *people* has the following property:



## 5. Automatic Generation of Speech Grammars from Relational Database Schema

---

*People's first name is FirstName.*

The SCG Grammar-Generation Engine recognizes that `People` is a table, `FirstName` is a column of table `People`, and the data in the column `FirstName` are proper nouns. Meanwhile, the data in the column `FirstName` of table `People` are assigned with the semantic type of `People`. We assume semantic types for `people`, `planet`, and `moon` are assigned with `type_1`, `type_2`, and `type_3` respectively. Then the data from the column `FirstName` of table `people` is appended to the definition of non-terminal `<pnoun_type_1>` (i.e., proper noun, semantic type 1), as follows:

```
<pnoun_type_1> = Bernard | bond | cassini | dollfus;
```

Note that, in CFG grammar generation, we only use `<pnoun>` to denote proper nouns. In SCG grammar generation, we use `<pnoun_type_k>` ( $1 \leq k \leq n$ ) to specify the proper nouns associated with their semantic types to guarantee the semantic agreements.

Step 4: extract the activities and relationships.

The SCG Grammar-Generation Engine interprets the third part (`<Activity></Activity>`) in the Semantic Specification, and recognizes the transitive verbs or intransitive verbs by examining whether the predicate verb takes any object. The SCG Grammar-Generation Engine indicates the semantic types for the verb, subject, and object. The semantic type of an object is dependent on the verb. Therefore, it is the verb that determines the semantic types of the subject and object of a sentence. We make a record of the semantic types of the subject and object depending on the activity (verb).

```
subject predicate object.
```

## 5. Automatic Generation of Speech Grammars from Relational Database Schema

---

Note that, the above three parts must have specified semantic types. Note that, an intransitive verb does not have any object, so the “semantic type of successor (object)” for an intransitive verb will be “NULL”. Also note that, the subject and/or object could be a noun phrase with modifiers for the noun. The modifiers require the same semantic type as the noun.

The SCG Grammar-Generation Engine interprets the syntactic and semantic constraints from the Semantic Specification and the database schemas, and records the information in a table. Table 5.5.2 shows the example syntactic and semantic constraints for the example solar system domain.

Table 5.5.2: example syntactic and semantic constraints

Constraint_ID	subject	predicate (Verb)	Object
1	People	Discover	Planet
2	People	Discover	Moon
3	Moon	Orbit	Planet
4	Planet	Exist	NULL
5	Moon	Exist	NULL
6	Planet	Spin	NULL
7	Moon	Spin	NULL

This table determines the correct format for a valid sentence. It is also possible to encode linguistic agreement (e.g., number and person) in the grammar. However, this would significantly increase the size of the grammar and we will not discuss this in this report.

With the above table, a syntactically and semantically correct sentence can be determined easily. For example, an utterance like `Bond discovered jupiter` will be accepted by the generated SCG grammar for the reasons as follows:

The subject `Bond` belongs to the semantic type of `people` and the object `jupiter` has the semantic type of `planet`. By constraint 1 in Table 5.5.2, the

## 5. Automatic Generation of Speech Grammars from Relational Database Schema

---

transitive verb `discover` requires a subject (preceding word) with the semantic type of `people` and an object (successor) with the semantic type of `planet`. Thus, the utterance `Bond discovered jupiter` is correct by the SCG.

However, the utterance like `Mars discovered jupiter` will be considered wrong by the SCG grammar, because by `constraint_1` and `constraint_2` in Table 5.5.2, the predicate `discover` requires the subject (preceding word) with the semantic type of `people`. In the example utterance, the subject `mars` does not belong to the semantic type `people`. Therefore, such an utterance will be considered incorrect by the SCG grammar.

By excluding the semantically incorrect utterances, a SCG grammar should improve the speech-recognition accuracy.

Step 5: fill out the SCG Grammar Template.

Similar to the process of CFG grammar generation, the SCG Grammar-Generation Engine fills out the SCG grammar template. When it deals with the description of the entity properties, it puts the data from the database into the rules defining corresponding non-terminals, based on their parts of speech and semantic types. In addition, the syntactic rules and clues are taken into account in SCG grammar generation.

Step 6: optimize the SCG grammar.

Note that, in the SCG template (Figure 5.5.1), we use `type_k` to specify the semantic type to guarantee the agreement among semantic constraints. We use `type_i` to list all the possible semantic types. There is a shortcoming of this method. If some semantic type is unsuitable in the domain, the grammar template will expand with an empty rule for this `type_k`, which will not affect the correctness of the generated grammar, but will affect the size of the grammar so that it increases the difficulty for grammar maintenance. In addition, it may

decrease the performance of the recognizers implementing the grammar. For example, in the SCG grammar template (Figure 5.5.1), there is a rule defined as follows:

```
<nouncla_type_k>
 = <cnoun_type_k>
 | <adj_type_k> <cnoun_type_k>; (1)
```

While the SCG Grammar-Generation Engine proceeds with the SCG grammar generation, it interprets this rule (1) as the following rule (2):

```
<nouncla_type_1>
 = <cnoun_type_1>
 | <adj_type_1> <cnoun_type_1>; (2)
```

Supposing `type_1` is the semantic type `people`, we notice that it is not suitable to say `color people`. Therefore, in the SCG grammar generation, the SCG Grammar-Generation Engine will generate the following rule:

```
<adj_type_1> = ; (3)
```

This empty rule will not affect the correctness of the generated SCG grammar with respect to recognition of appropriate utterances. However, it increases the size of the generated grammar and makes it more complex, which is not good for maintenance and grammar optimization. Therefore, SCG Grammar-Generation Engine needs to optimize the generated SCG grammar.

In the optimization process, the SCG Grammar-Generation Engine scans the new generated grammar for the empty rules and removes them and their associated alternatives. If the alternative is the only choice of the rule, the whole rule is removed and the process needs to trace further for the left-hand-side non-terminal and removes its appearance(s) in other rules. The process continues until all empty rules and their associated rule(s)/ alternative(s) are removed.

## 5. Automatic Generation of Speech Grammars from Relational Database Schema

---

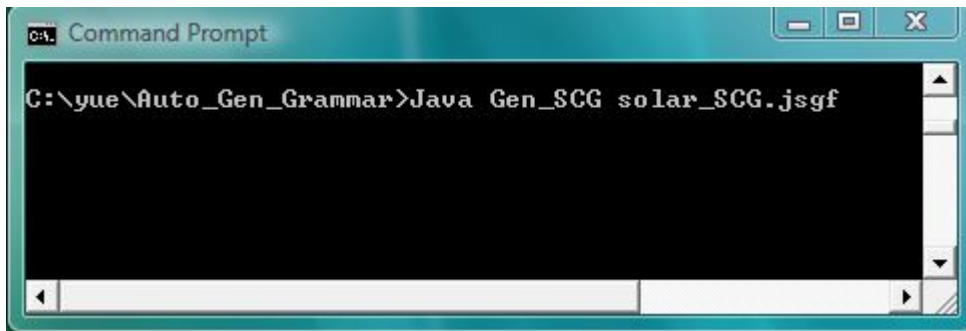
For the above example, the rule (3), `<adj_type_1> = ;` is removed from the generated SCG grammar, and the alternative `| <adj_type_1>` `<cnoun_type_1>` in rule (2) is removed too. Then, the updated rule (2) is like the following (4):

$$\langle \text{nouncla\_type\_1} \rangle = \langle \text{cnoun\_type\_1} \rangle; \quad (4)$$

The complete generated SCG grammar example is at Appendix H.

Step 7: output the SCG grammar.

As in CFG grammar generation, the user can specify the location and file name for the output SCG grammar in the command line. In the example screenshot of Figure 5.5.2, `Gen_SCG` is the name of the SCG Grammar-Generation Engine, and `solar_SCG.jsgf` is the name of the generated SCG grammar. Without specifying the location, the newly-generated SCG grammar `solar_SCG.jsgf` is saved in the current directory, which is the directory of SCG Grammar-Generation Engine.



*Figure 5.5.2: screenshot - example command to generate a SCG grammar*

## 5.6 Analysis of the Automatically-Generated CFG and SCG Using the Grammar Metrics

We have analyzed and compared the automatically generated CFG and SCG grammars with each other and with previous manually-crafted grammars, using the set of grammar metrics discussed in section 2. Similar to the analysis in sub-section 4.4, the newly-generated CFG and SCG grammars were analyzed using the ABF application (section 2.9). A set of grammar metrics were calculated and output, including the Average Branching Factor (ABF), the number of rules of the grammar, the number of symbols, the number of non-terminals, the number of terminals, the number of decision points, and the language size. To facilitate the comparison, in Table 5.6, we include the results for the manually-crafted CFG and SCG grammars (from sub-sections 4.3.1 and 4.3.2) and the results for the newly-generated CFG and SCG grammars.

Table 5.6: comparison of manually-scripted and automatically-generated CFG and SCG grammars

Grammar	Rule #	Symb ol #	Non-terminal#	Termi nal#	Decision Point #	Language Size	ABF
Manual-CFG (Figure 4.3.1)	17	160	50	110	19	$1.73 \times 10^{11}$	52.42
Auto-Gen CFG (Appendix G)	17	160	50	110	19	$1.73 \times 10^{11}$	52.42
Manual-SCG (Figure 4.3.2)	41	262	133	129	53	$1.51 \times 10^9$	33.99
Auto-Gen SCG (Appendix H)	41	278	136	142	61	$1.56 \times 10^9$	33.48

Table 5.6 shows that, with the same domain, the automatically generated CFG has similar features as that of the manually-scripted CFG, and the automatically generated SCG is a little bigger and has a slightly smaller ABF than the manually-scripted SCG. The automatically generated SCG consists of more symbols (terminals and/or non-terminals) than the manually scripted SCG. The reason is that in automatic grammar generation, the Grammar Generation

## 5. Automatic Generation of Speech Grammars from Relational Database Schema

---

Engine expands all the possible cases though some cases may be not needed or can be combined with other cases in manual grammar scripting.

The automatically generated CFG is similar to the manually-developed CFG because the CFG grammar template was derived from the manually-developed CFG. In the CFG grammar generation, the CFG grammar generation engine interprets the parts of speech of the words in the database and fills in the corresponding non-terminals of the CFG grammar template. For example, word `bernard` will be added to the non-terminal `<pnoun>` of the CFG template.

```
<pnoun> = bernard ;
```

Note that, this process does not change the structure of the CFG template. Therefore the automatically generated (complete) CFG is similar as the manually-scripted CFG, given the same vocabulary.

In the SCG grammar automatic generation, the SCG grammar generation engine expands the SCG grammar template with respect to semantic types and interprets the database schemas to generate a new SCG grammar. In the example, the automatically generated grammar is slightly different from the hand-crafted grammar. The reason is that, by examining both grammars, we noticed that, the manually-crafted SCG grammar sometimes only differentiates between two semantic types (i.e., `animate` from `inanimate`), while in the automatic SCG grammar generation, the system always automatically expands the SCG grammar template by three types (i.e., `people`, `planet`, `moon`).

For example, in rule 1 of the manual SCG, the `who/what` queries are defined as:

```
| (who) <animate_verbph>
| (what) <inanimate_verbph>
```

However, in the automatically generated SCG, `who/what` queries are specified with respect to three semantic types

## 5. Automatic Generation of Speech Grammars from Relational Database Schema

---

```
| (who) <verbph_type_1>
| (what) <verbph_type_2>
| (what) <verbph_type_3>
```

Therefore, the example automatically generated SCG includes slightly more constraints than the original manually-crafted SCG grammar. This also explains the slight difference in grammar sizes and metrics between these two grammars.

The manually-scripted SCG grammar also differentiates between the three semantic types (i.e., `people`, `planet`, `moon`) for the description of their activities/relationships among these three entity types (semantic types). Therefore, the differences of the metrics between the manual and automatic SCG grammars are minor.

In this section, what we have presented is only one small example. However, this preliminary attempt demonstrates the viability of automatic generation of recognition grammars that have comparable ABFs to hand-crafted grammars.

### 5.7 Comparison to Related Work

With our extensive survey, we have noticed that there has not been much work carried out on automatic grammar generation. To be specific, there is little work on automatic grammar generation from relational database schemas. Voxeo (2006) introduces an approach to create simple dynamic speech grammars from databases using the ColdFusion server side language.

The basic idea of (Voxeo, 2006) is to create a dynamic-grammar template and use database queries to retrieve information from database. The generated grammar is a very simple grammar which uses database queries to retrieve information from databases for dialogs or prompts in a VoiceXML file. For example, the generated grammar may allow users to state his/her favorite movies. The names of the movies in the database and can be retrieved by using database queries.



In contrast, our approach can automatically generate more expressive speech-recognition grammars from database schemas.

### 5.8 Summary

In this section, we have presented a new approach to automatically generate CFG and SCG grammars from relational database schemas. The new approach is based on the assumption that the high-level syntactic structures of many speech-recognition grammars are similar for applications of similar type (e.g., database query applications) and differ only in the lexicon and in syntactically-expressed semantic constraints.

We have applied the approach to a simple database schema and have automatically generated recognition grammars which have similar properties to manually crafted grammars with respect to a set of grammar metrics. Although no general conclusions can be drawn from this limited experiment, it does provide some evidence that the proposed approach deserves further investigation.

## 6. A PUBLIC-DOMAIN SPEECHWEB

### 6.1 Introduction

In this section, we illustrate the ease with which grammar-based speech applications can be created and deployed on the web. We do this by building a small grammar-based speech application and show how it can be easily added to a “Public Domain SpeechWeb”. We begin with a short history of SpeechWebs and the Public-Domain SpeechWeb architecture.

The example speech application is called “Read-A-Book”, which can read the book “Sleeping Beauty” by users’ voice command. The grammar for this application was analyzed and the ABF was computed in sub-section 2.9.

### 6.2 SpeechWebs

A SpeechWeb (Frost and Chitte 1999) is a collection of hyperlinked speech applications which are accessed by end-users through speech browsers running on local machines. Navigation from one application to another is also through speech commands such as `can I speak to Geoman` which causes the remote speech application to send information back to the local speech browser which causes it to be redirected to a, possibly new, remote web server which hosts the `Geoman` speech application.

### 6.3 The LRRP SpeechWeb Architecture

Up to 2004, three architectures that were used to provide speech access to distributed applications (Frost et al., 2004).

- (1) The first architecture uses speech interfaces (screen readers) to interact with the conventional web.
- (2) The second architecture is the RRRP architecture, which is often used by call centers. RRRP stands for Remote Recognition/ Remote Processing, which means that the user calls, and the speech recognition is processed at the call center.

- (3) The third architecture is the LRLP architecture, which stands for Local speech Recognition/Local Processing. With this architecture, hyperlinked VoiceXML pages are downloaded to client machines for execution. Note that VoiceXML(VXML) is a Voice eXtensible Markup Language, which is an XML-based markup language for building distributed voice applications, much as HTML is a markup language for creating distributed visual applications (W3C, 2007a). VoiceXML documents define the applications as a set of dialog states by including commands for prompting user speech inputs, for invoking recognition grammars, for outputting synthesized voices, and for directing the user from one state to the other state.

Although these three architectures are important in providing speech access to distributed knowledge and applications, they all have shortcomings as a basis for a SpeechWeb consisting of speech applications that are developed and deployed by users that do not have expertise in language processing (Frost et al., 2004): 1) Since the conventional web is mainly constructed for visual browsing much of a conventional web page content is inaccessible through a screen reader. 2) In the second architecture, the processing and recognition both occur at the remote provider site, which is not accessible to non-expert application developers. 3) In the third architecture, it requires significant expertise to build applications purely in VXML. Also, speech recognition and application processing locally excludes the light-weight user devices.

To overcome the above shortcomings, Frost et al. (2004) proposed a new architecture, called the LRRP architecture, to access hyperlinked speech-accessible knowledge sources that are distributed over the internet. LRRP stands for, Local speech Recognition and Remote Processing.

In the LRRP architecture, the user's voice input is recognized locally by a voice browser on the local machine, the recognized text is sent to the remote

application for processing, and the result is returned to the local device and output as synthesized voice.

### 6.4 A Public-Domain SpeechWeb

In 2005, Frost described an architecture for a Public-Domain SpeechWeb in the Communications of the ACM (Frost 2005). The architecture is based on the LRRP architecture and allows users who do not have expertise in language processing to create and deploy hyperlinked speech applications using freely available software and commonly used communication protocols. The basic idea is that a speech browser, written in VXML, resides on the end-user device and the speech application, which can be written in any programming or scripting language, resides on a remote server. A “session” begins by the local speech browser requesting the download of the application-dependent recognition grammar from the remote server. The local speech recognition engine is then tailored for the specific speech application. End-user spoken input is then recognized locally and the corresponding text is sent to the remote application for processing. Text is sent back to the local device and either output as synthesized speech or used to cause the local device to contact a different speech application and download a new recognition grammar.

The author of this thesis, together with Mr. Xiaoli Ma helped Dr. Frost (Frost, Ma, and Shi 2007) reengineer the Public-Domain SpeechWeb software so that the speech browser is written in X+V, a multi-modal markup language (VoiceXML Forum, 2004) and the freely available Opera multi-modal web browser (Opera, 2010) to more easily create and deploy hyperlinked speech applications to the Public-Domain SpeechWeb.

X+V (i.e., XHTML + Voice) is a markup language which combines XHTML with a subset of VoiceXML (VXML) so that it can bring spoken interaction to standard web content in multi-modal applications (VoiceXML Forum, 2004). XHTML is an eXtensible HyperText Markup Language, which has the same expressive power as HTML, but also conforms to XML syntax (W3C, 2007b).

The Opera web browser is a freely downloadable multi-modal web browser (Opera, 2010). By some simple configuration, the Opera web browser downloads and uses the free IBM speech-recognition plug-in and can then execute X+V pages.

The SpeechWeb architecture described in (Frost, Ma, and Shi, 2007) is shown in Figure 6.4:

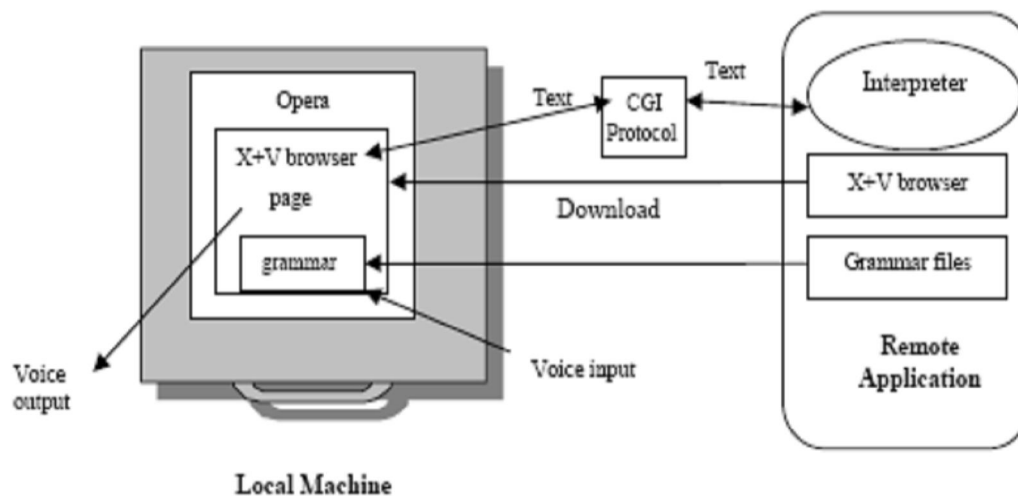


Figure 6.4: LRRP SpeechWeb architecture (Frost, Ma, and Shi, 2007)

In this architecture, the speech applications (i.e. the interpreter, a specialized copy of the X+V browser, and the grammar file for the application) reside on conventional remote web servers. Each application consists of a recognition grammar and an interpreter. The *grammar* defines the input language of the application. The *interpreter* is a program that takes the recognized text as input and returns a text result which is returned to the local browser. The interpreter can be written in any language. The copy of the X+V browser which resides on the remote server (as part of the application) is specialized for the application with a special greeting, and also contains the URLs of the interpreter and the grammar file.

When the user starts the Opera X+V browser on the local machine and contacts a remote speech application, the copy of the X+V browser is returned from the remote application. This browser is tailored to the speech application by having three application-specific parts: 1) A greeting, 2) the URL of the recognition grammar, and 3) the URL of the interpreter. Next, the recognition grammar from the remote application is downloaded and used to tailor the recognition engine of the browser. The user's voice input is recognized by the local browser, and the recognition result is sent as text to the remote application. The interpreter residing on the remote application accepts the text input, processes it, and returns the result as text to the X+V browser on the local machine. The result is then output to the user as synthesized voice.

The reason why each application has its own version of the X+V browser is to overcome what appears to be a bug in the X+V interpreter. Recognition grammars cannot be changed when an X+V script is executed. When this bug is fixed, a single X+V browser can be used on the local device to access different speech applications. The greeting message, URL of the recognition grammar, and URL of the interpreter for each application could then be stored in a file at the remote location associated with the application.

If the user's command is a request to access another speech application, the voice input is recognized as such by the current speech application, whose interpreter returns the URL of the new speech application. The browser recognizes the URL link and then redirects to the new speech application. A new recognition grammar is downloaded and the above process continues until the user requests to leave the SpeechWeb.

The advantages of using LRRP architecture as the basis of the Public-Domain SpeechWeb are discussed in (Frost, 2005) as follows:

- (1) It improves speech-recognition accuracy and efficiency. Only the application-specific speech-recognition grammar is downloaded and applied, which is efficient and improves speech-recognition accuracy.
- (2) Speech applications can be written in any language with input and output that conforms to the Web communication protocol.
- (3) It is suitable for expert users and users who do not have expertise in language processing. People who do not have expertise in language processing can create simple applications with canned answers to user queries. Advanced developers can build complex applications on powerful server-side machines.

### 6.5 The Example of a Speech Application

In this thesis, we include an example to demonstrate the ease of creating and deploying a speech application on the Public-Domain SpeechWeb. The example speech application is a very simple application called `Read-A-Book`, which allows speech access to the book `Sleeping Beauty`. The application can read the book by page or by chapter. Also, the user can command the application to read the pages referring to some specific words. In addition, the user can ask some book-related questions, such as `what is the title of the book` or `who is the author of the book`. Meanwhile, the user can get assistance from the system by asking for `help`, `what can I say`, or `what do you know`.

The example speech application was chosen to illustrate the ease with which grammar-based speech applications can be created and deployed using the Public-Domain SpeechWeb architecture. It is very simple and does not have many commands, nor does it do any natural-language processing. More powerful speech applications have been built, and added to the Public-Domain SpeechWeb, by the research group at the University of Windsor and are briefly described in Frost et al (2008). However, the same methods are used to provide the speech recognition capability and to deploy these applications as that are used by the `Read-A-Book` application discussed here.

Similar to navigating from a conventional website to another website on the Internet, a user can navigate through speech applications on the SpeechWeb. For example, by asking `can I talk to judy`, the user leaves the current speech application to access the `judy` speech application.

The hardware and software requirements for speech-application development include any PC or handheld computer that can run a version of the Opera browser which supports X+V, a microphone and speakers for voice input and output. The Opera multi-modal web browser is freely downloaded at Opera (2010), and configured for voice capability.

We need to create four files to build a speech application and deploy it on the existing SpeechWeb: (1) an XML file, (2) a grammar file, (3) an interpreter file, and (4) a CGI file (Frost et al, 2007). Taking the `Read-A-Book` application as an example, we now discuss each of the four files:

### (1) The XML file.

The XML file is a copy of the X+V web browser created by Frost, Ma and Shi (2007) and subsequently modified by Frost, Karaki, et al (2008). It can be obtained from the SpeechWeb website at <http://cs.uwindsor.ca/~speechweb/>. The .xml file is modified by changing the greeting, the URL of the recognition grammar, and the URL of the interpreter for the `Read-A-Book` application. This modification took only a few minutes. The .xml file for the `Read-A-Book` application is given in Appendix I.

### (2) The recognition grammar file.

Speech-recognition grammars vary from application to application. The grammars are written in JSGF format to define the input language of the speech application. The name of the grammar file needs to be consistent with the URL in the XML file.



In the Read-A-Book example, the system firstly greets and responds to user's greetings like, Hello or Hi there. The user can command the application to read the book by user's preference. For example, the user can ask the system to read the book starting from specified places, such as the beginning, page numbers, chapter numbers, or referring to some words, such as `please read page two` or `read chapter five`. The user can also ask some questions related to the book, such as who is the author of the book, and what is the title of the book. If the user encounters difficulty while using the application, s/he can ask for help at any time, like `help`, `what do you know`, or `what can I say`.

The recognition grammar for application Read-A-Book is given in Appendix C. This example grammar is very simple. More expressive grammars can be found at the SpeechWeb website <http://cs.uwindsor.ca/~speechweb/>.

### (3) The interpreter file.

The interpreter is a program that takes the recognized text as input and outputs text after processing the input. For the Public-Domain SpeechWeb, the interpreter must reside in the same directory as the .xml file and must have the extension .cgi as cgi is the communication protocol used by the Public-Domain SpeechWeb (note that for interpreters built using interpreted programming languages an additional script with the .cgi extension must also be created as discussed later).

For the Read-A-Book application, the interpreter is a simple program. For example, if the user says `hello`, the local browser converts it to text and sends it to the interpreter which sends the text `hi` back to the local browser which outputs it using synthesized voice. If the user says `read page ten`, the interpreter responds to the user with the content of the book at page ten and the following pages until the user requests `stop` or inputs another commands. In the example Read-A-Book application, without interruption from the user, the system will

read five continuous pages by default and prompt the user to `continue` or `stop`. The interpreter program interprets the user's text queries/ commands and returns the answers to the user by synthesized voice or directs the user to corresponding speech applications as required.

Interpreters can be written in any programming language provided that the input and output are handled by the standard input/output features of the language. The interpreter for the example speech application `Read-A-Book` was written using Miranda, a non-strict purely functional programming language. An excerpt of the `Read-A-Book` interpreter is in Appendix J.

(4) The CGI file (which is only required when the interpreter is written in an interpreted language)

Because Miranda is an interpreted language, a `.cgi` script is also required as the `.cgi` file which invokes the Miranda interpreter when the script is sent input from the local speech browser.

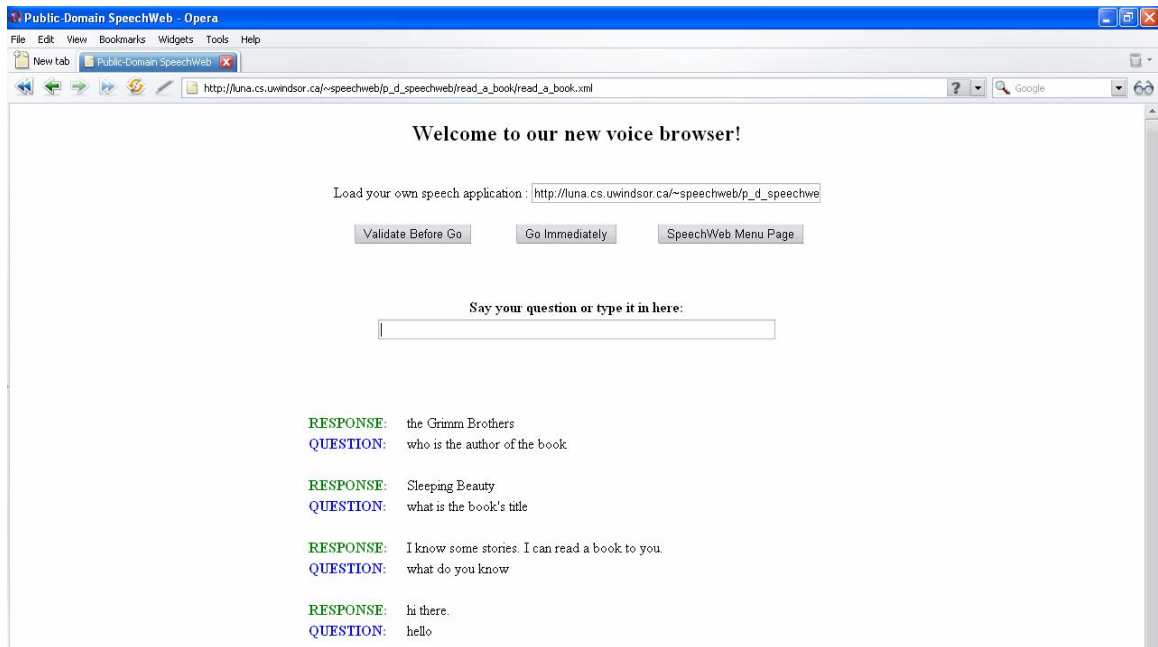
The `.cgi` file can be written in Unix or any other scripting language which is supported by the web server on the remote compute server. The `.cgi` file for the `Read-A-Book` application is a Unix script which can be found in Appendix K.

In order to deploy the `Read-A-Book` application on the Public-Domain SpeechWeb, the four files are placed in a directory which is accessible through a web server running on a compute server linked to the Internet.

To start the `Read-A-Book` speech application, the user directs the Opera browser to the XML file at the URL for the `Read-A-Book` application. When the opera multi-modal web browser starts, the application greets the user by voice saying `Hello, how are you? I am going to read a book for you.`

A sample screenshot of the application is in Figure 6.5. More screenshots for the conversation of the `Read-A-Book` application are in Appendix L.

## 6. A Public-Domain SpeechWeb



*Figure 6.5: the screenshot of Read-A-Book application.*

In this example conversation, the user says `hello`, and the application responds with `hi there`. The user then asks `what do you know`. The application responds with `I know some books. I can read a book for you`. The user continues to ask `what is the book's title`. The application answers, `Sleeping Beauty`. The user further asks, `who is the author of the book`, the application answers `the Grimm Brothers`.

On the screen, the question (user's command) is displayed first. The response (from the application) is displayed above the question on screen. The subsequent conversations are displayed in the same way, and the screen scrolls down.

### 6.6 Summary

A SpeechWeb is an augmentation of the conventional web. It extends the concept and the usage of the traditional web. It provides speech access to specially-created applications for people with visual disabilities and for situations where hands-free access is necessary.

In this section, we first introduced the concept of a SpeechWeb. We then presented the LRRP (Local Recognition/ Remote Processing) SpeechWeb architecture (Frost et al., 2004) and explained its advantages. Next, we presented an example speech application, *Read-A-Book*, to demonstrate how easily a speech application can be constructed and deployed on the Public-Domain SpeechWeb using freely available software and commonly used communication protocols.

## 7. CONCLUSION

### 7.1 Proof of the Thesis

We began with the thesis that natural-language speech-recognition grammars are amenable to methodical analysis and design techniques. In particular:

- (1) Various grammar metrics, including the Average Branching Factor (ABF) can be computed automatically and efficiently.
- (2) Semantic constraints can be encoded in syntax rules in order to decrease language size and ABF.
- (3) Recognition grammars can be created automatically from relational database schemas and application specifications.
- (4) Readily-available speech-recognition technology and commonly-used communication protocols can be used by non-expert as well as expert users to create and deploy speech applications.

We have proven each part of the thesis by constructing algorithms and software. Such proofs are informal and are really “proof of concept”. However, formal mathematic proofs were given showing termination, correctness and polynomial complexity of the ABF algorithm.

### 7.2 Future Work

In this thesis report, we have proposed a novel and efficient algorithm of computing the Average Branching Factor (ABF) directly from speech-recognition grammars to assist the analysis and design of speech-recognition grammars. However, this algorithm has the following three constraints: 1) the grammar must be proper, 2) the grammar must be 1-lookahead, and 3) the grammar must be non-recursive. For many speech-recognition applications, these constraints can be easily accommodated. However, it should be noted that if our algorithm can be modified to overcome the last constraint (i.e., non-recursive), then it would have application in many other areas of A.I. The reason for this is that many problems that involve search (for example constraint-satisfaction, pattern-

recognition, planning, etc.) can be defined as parsing and/or grammar expansion (derivation) problems. The Average Branching Factor (ABF) is one metric that can be used when comparing different problems and different language-based solutions to those problems. Therefore, it would be useful to investigate how to extend the algorithm to accommodate recursive grammars. However, the approach would likely be very different as recursive grammars would generate infinite languages.

It would be useful to consider the addition of probability values to the alternatives in the syntax rules. The reason for this is that it is likely, in the near future, those speech-recognition engines will become available that are based on probabilistic grammars. In such grammars, each alternative in each syntax rule is labeled with a probability value that represents the likelihood of that alternative matching the input. These probability values are then taken into account together with the results of matching the phonetic properties of the next segment of the input with those of the words in the director sets of the alternative branches of the production rule. No research appears to have yet been carried out on relating average branching factors, probability values, and recognition accuracy. We would like to investigate into the use of probabilistic grammars when recognition engines that are based on probabilistic grammars become readily available.

We have discussed, analyzed, and compared the CFG and SCG grammars in this thesis. It would be useful to study and apply more semantic constraints in future work to improve the recognition accuracy and assist the design and development of speech-recognition grammars.

We have proposed a novel approach to generate the CFG and SCG grammars automatically from relational database schemas. The automatically-generated CFG/ SCG grammars can be used to the speech applications or speech interfaces for database queries. In future work, it would be useful to develop methods to automatically generate other types of speech grammars.

## 7. Conclusion

---

Finally, the example speech application which we developed using the SpeechWeb architecture illustrates the ease with which such applications can be constructed and deployed on the Internet. In future work, it would be useful to investigate methods which integrate the automatic generation of recognition grammars from database schemas with other components so that a complete speech query interface to a given Oracle database could be automatically generated and deployed on the Internet.

## BIBLIOGRAPHY

1. Abbott, K. (2001) Voice Enabling Web Applications: VoiceXML and Beyond. *Published by CMP books (April 2001), and by Apress (November 2001).*
2. Aho, A.V., Sethi, R., and Ullman, J.D. (1986) Compilers: Principles, Techniques, and Tools; Addison-Wesley Publish Company; (Everywhere). *Reprinted with corrections March, 1988; ISBN 0-201-10088-6.*
3. Akiba, T. and Itou, K (2000) Semi-Automatic language Model Acquisition without Large Corpora. *Proceedings of the ICSLP*, 4, 49-52.
4. Alshawi, H. (1992). The Core Language Engine, *MIT Press*, Cambridge.
5. Angel Voice Site, VUI Design – Introduction to VUI Design, [www.angel.com](http://www.angel.com) .
6. Amaya, F., Benedf, J. M. and Sánchez, J.A. (1999) Learning of Stochastic Context-Free Grammars From Bracketed Corpora By Means of Re-estimation Algmtorithms. *VIII Symposium on Pattern Recognition and Image Analysis*. 1, 119-126.
7. Apache Software Foundation (2005) Voice As a User Interface- Case Study and Lessons Learned, <http://www.redbooks.ibm.com/redpapers/pdfs/redp3893.pdf> .
8. Arai, K., Wright, J., Riccardi, G., and Gorin, A. (1998) Grammar Fragment Acquisiton Using Syntactic and Semantic Clustering. *Proceedings of the ICSL. Proc.*, 5, 2051-2054.
9. Aust, H., Oerder, M., Seide, F., and Steinbiss, V. (1995) The Philips Automatic Train Timetable Information System. *Speech Communication* 17, 249-262.
10. Baggia, P. (January 2006) Impact of Standards on Automatic Speech Recognition, <http://www.speechtechmag.com/issues/industry/12821-1.html#top>
11. Baggia, P., Castagneri, G., and Danieli, M. (2000). Field Trials of The Italian ARISE Train Timetable System. *Speech Conference on Spoken Language Processing*, 550-553.
12. Baker, J. (1979) Trainable Grammar for Speech Recognition. In *D. Klatt & J.*



- Wolf (eds.) Speech Communication papers for the 97th Meeting of the Acoustical Society of America, 547-550.*
13. Balakrishna, M. (2007) Exploiting High-Level Knowledge Resources for Speech Recognition With Applications to Interactive Voice Response Systems. *Ph.D. Dissertation*, University of Texas at Dallas.
  14. Bangalore, S., and Johnston, M. (2003) Balancing Data-driven and Rule-based Approaches in the Context of a Multimodel Conversational System. *In Automatic Speech Recognition and Understanding (ASRU'03)- IEEE Workshop.* 221-226.
  15. Banzhaf, W., Nordin, P., Keller, R.E., and Francone, F.D. (1998) Genetic Programming: An Introduction. *Morgan Kaufmann Publisher.*
  16. Barnard, E., Halberstadt, A., Kotelly, C., and Phillips, M. (1999) A Consistent Approach to Designing Spoken-dialog Systems. *In Proceedings of the Automatic Speech Recognition and Understanding Workshop (ASRU99)*, Keystone, Colorado.
  17. Bernsen, N.O., Dybkjær, H., and Dybkjær L. (1995) Exploring the limits of system-directed dialogue. Dialogue Evaluation of the Danish Dialogue System. *Proceedings of Eurospeech '95*, Madrid, 1457-1460.
  18. BeVocal Inc., <http://cafe.bevocal.com/>
  19. Biber, C. and Kozminski, A. (2005) Six Steps for Creating a Speech Recognition Application or Speech-Enabling Your DTMF IVR. <http://www.microsoft.com/speech/community/newsletter/articles/0405article.msp> .
  20. Bigham, J., Prince, C.M., and Ladner, R.E. (2008) WebAnywhere: A Screen Reader On-the-Go. *In Proceedings of the 2<sup>nd</sup> Cross-Disciplinary Conference on Web Accessibility, W4A 2008.* Beijing, China. 73-82.
  21. Blackburn, P. and Striegnitz, K. (2002) <http://www.coli.uni-sb.de/~kris/nlp-with-prolog/html/>.
  22. Bouzid, A. (2006a) VUI Design- VUI 101, *Angel Voice Site Newsletter.* <http://www.angel.com/ivr/university/vui2.jsp> .

23. Bouzid, A. (2006b) 10 tips for crafting an effective opening prompt, *Angel Voice Site Newsletter* . <http://www.angel.com/ivr/university/vui20.jsp> .
24. Bouzid, A. (2006c) VUI Design – Site Builder Tips for Better VUI, *Angel Voice Site Newsletter*. <http://www.angel.com/ivr/university/vui9.jsp> .
25. Bouzid, A. (2006d) VUI Design – The VUI View: Top 10 VUI No-Nos, *Angel Voice Site Newsletter*. <http://www.angel.com/ivr/university/vui10.jsp>.
26. Bouzid, A. (2006e) VUI Design – Why VUI Design is More Challenging than GUI Design, *Angel Voice Site Newsletter*,  
<http://www.angel.com/ivr/university/vui11.jsp> .
27. Bouzid, A. (2006f) VUI Design –Effective Voice Prompts, *Angel Voice Site Newsletter*, <http://www.angel.com/ivr/university/vui4.jsp> .
28. Bouzid, A. (2006g) VUI Design – Effective Voice Menus, *Angel Voice Site Newsletter*, <http://www.angel.com/ivr/university/vui3.jsp> .
29. Boyce, S. (2000) Natural Spoken Dialogue Systems for Telephony Applications, *Communications of the ACM*, 43(9): 29-34.
30. Boyce, S. (1999) Spoken Natural Language Dialogue Systems: User Interface Issues For the Future. In D. Gardner-Bonnea, ed., *Human factors and voice interface systems*, Norwell, MA: Kluwer Academic Publishers. 205-235.
31. Myers, B., Hudson, S., and Pausch, R. (2000) Past, Present, and Future of User Interface Software Tools. *ACM Transactions on Computer-Human Interaction* 7 (1) 3-28.
32. Brown, P. F., Pietra, V., deSouza, P., Lai, J., and Mercer, R (1992) Class-based n-gram models of natural language. *Computational Linguistics*, 18. 467-479.
33. Cai, O. and Hamaker, J. (2008) Grammar Confusability Metric for Speech Recognition. *United States Patent*. IPC8 Class: AG10L1100FI, USPC Class: 7042701, Assignees: Microsoft Corporation, Origin: RedMond, WA US.
34. Callison-Burch, C., Osborne, M. and Koehn, P. (2006) Re-evaluating the Role of BLEU in Machine Translation Research. In *11th Conference of the European Chapter of the Association for Computational Linguistics: EACL 2006*. 249–256.

35. Carpenter, P., Jin, C., Wilson, D., Zhang, R., Bohus, D., and Rudnicky, A. (2001) Is This Conversation on Track? *Proceedings of EUROSPEECH*. Aalborg, Dinamarca, 3, 2121-2125.
36. Carrasco, R.D. and Oncina, J. (1994) Grammatical Inference and Applications, Lecture Notes in Artificial Intelligence (LNAI 862), Springer-Verlag.
37. Caskey, S.P., Story, E., and Pieraccini, R. (2003) Interactive Grammar Inference with Finite State Transducers. In *Proc. IEEE Workshop Automatic Speech Recognition and Understanding (ASRU 2003)*. 572-576.
38. Chant Inc. (2002) Chant Developer Workbench, <http://www.chant.net/Products/Developer%20Workbench/Default.aspx> .
39. Chelba, C., and Jelinek, F (2000) Structured Language Modeling. In *Computer Speech & Language*. 14(4), 283-332.
40. Chen, S. (1995) Bayesian Grammar Induction for Language Modeling. *Proceedings of the COLING/ACL 1995, 33<sup>rd</sup> Ann. Meeting of the Assoc. Computational Linguistics*, 228-235.
41. Churcher, G.E., Souter, C. and Atwell, E.S. (1996) Dialogues in Air Traffic Control. In *Proceedings of 11<sup>th</sup> Twente Workshop on Language Technology: Dialogue Management in Natural Languages Systems (TWLT11)*, Twente, Netherlands.
42. Corballis, L. P. (1994) Interactive Dynamic Grammar Constraint in Speech Recognition. *European Patent Application EP0618565*.
43. Cover, T.M. and Thomas, J.A. (2006). Elements of Information Theory. Second Edition. *John Wiley & Sons*, New York, NY. ISBN: 0-471-24195-4. 776.
44. Črepinšek, M., Mernik, M., and Žumer, V. (2004) Extracting Grammar from Programs: Brute Force Approach. *ACM SIGPLAN Notices*, 40(44), 29-38.
45. Črepinšek, M., Mernik, M., Javed, F., Bryant, B., and Sprague, A. (2005) Extracting Grammar from Programs- Evolutionary Approach. In *Proceedings of ACM SIGPLAN Notices*, 40(4), 39-46.
46. Dahl, D. A., Bates, M., Brown, M., Fisher, W., Hunicke-Smith, K., Pallett, D., Pao, C., Rudnicky, A., and Shriberg, E. (1994) Expanding the Scope of the

- ATIS Task: the ATIS-3 Corpus. *Proceedings of ARPA Human Language Technology Workshop*. 43-47.
47. Dellaert, F. (2002) The Expectation Maximization Algorithm. *Technical report GIT-GVU-02-20*, College of Computing, Georgia Institute of Technology.
48. Dempster, A.P., Laird, N.M., and Rubin, D.B. (1977) Maximum Likelihood from Incomplete Data via the EM Algorithm. *Journal of the Royal Statistical Society B*, 1977. 39: 1-38.
49. Doddington, G. (2002) Automatic Evaluation of Machine Translation Quality Using N-Gram Cooccurrence Statistics. In *Proceedings of the Human Language Technology Conference (HLT)*, San Diego, CA pp. 128–132
50. Dowding, J., Gawron, J.M., Appelt, D., Bear, J., Cherny, L., Moore, R., and Moran, D. (1993) Gemini: A Natural language System for Spoken-language Understanding. In *31<sup>st</sup> Annual meeting of the Association for Computational Linguistics*. Columbus, Ohio, 54-61.
51. Dupont, P. (1994) Regular Grammatical Inference from Positive and Negative Samples by Genetic Search: The GIG Method. *Lecture Notes in Artificial Intelligence, Proceedings of the 2<sup>nd</sup> International Colloquium on Grammatical Inference and Applications. ICGI'94*, 862, 236-245.
52. Edelkamp, S. and Korf, R. E. (1998) *The Branching Factor of Regular Search Spaces*. In *Proceedings of the 15th National Conference on Artificial Intelligence and the 10th Conference on Innovative Applications of Artificial Intelligence (AAAI-98/IAAI-98)*, Madison, Wisconsin, U.S., 299-304.
53. Eisenzopf, J. (2006) Top 10 Best Practices for Voice User Interface Design, *the Ferrum Group, LLC* .  
<http://www.developer.com/voice/article.php/1567051> .
54. Everett, S., Wauchope, K., and Perzanowski, D. (1993) Adding Speech Recognition to a Natural Language Interface, *Navy Center for Applied Research in Artificial Intelligence Naval Research Laboratory*,  
<http://www.aic.nrl.navy.mil/papers/1993/AIC-93-026.ps> .

55. Fenton, N.E. and Pfleeger, S.L (1996) Software Metrics: A Rigorous and Practical Approach. *Thomson Computer Press, First Edition*. 656.
56. Fraser, N.M. and Gilbert, G.N. (1991) Simulating Speech Systems. *Computer Speech and Language* 5, 81-99.
57. Frost, R.A and Chitte, S. (1999) A New Approach For Providing Natural-Language Speech Access to Large Knowledge Bases. *Proceedings of the Pacific Association of Computational Linguistics Conference PACLING '99*, University of Waterloo, 82-89.
58. Frost, R.A. (2002) Improving Speech-Recognition Accuracy by Coding Semantic Constraints in the Syntax of the Recognition Grammar. *Technical Note*. School of Computer Science, University of Windsor.
59. Frost, R.A., Abdullah, N., Bhatia, K., Chitte, S., Hanna, F. Roy, M., Shi, Y., and Su, L. (2004) LRRP SpeechWebs. In *Proceedings of CNSR Communication Networks and Services Research Conference (CNSR 2004)*. 91-98.
60. Frost, R.A. (2005) A Call for A Public-Domain SpeechWeb. *Communications of the ACM*. 48, 11, 45-49.
61. Frost, R.A., Ma, X., and Shi, Y. (2007) A Step Towards a Public-Domain SpeechWeb. *AAAI'07, the Twenty-Second AAAI Conference on Artificial Intelligence*. Vancouver, B.C., Canada, 1307-1308.
62. Frost, R. A., Karaki, A., Dufour, D. A., Greig, J., Hafiz, R., Shi, Y., Daichendt, S., Chandon, S., Barolak, J., and Fortier, R. J. (2008) MySpeechWeb: software to facilitate the construction and deployment of speech applications on the web. *ASSETS 2008*: 249-250
63. Fu., K.S. and Booth, T.L. (1975a) Grammatical inference: Introduction and survey, part 1. *IEEE Transactions on Systems, Man and Cybernetics*, 5, 1975.
64. Fu., K.S. and Booth, T.L. (1975b) Grammatical inference: Introduction and survey, part 2. *IEEE Transactions on Systems, Man and Cybernetics*, 5, 1975.

65. Gao, Y., Erdogan, H., Li, Y., Goel, V., and Picheny, M. (2001) Recent Advances in Speech Recognition System for IBM DARPA. *Proceedings of EUROSPEECH*. Aalborg, Denmark. 503-507.
66. Gavalda, M. and Waibel, A. (1998) Growing Semantic Grammars. In *Proceedings of the 17th International Conference on Computational Linguistics and 36th Annual Meeting of the Association for Computational Linguistics, COLING/ACL98*, Montreal, Canada. 451-456.
67. Gavalda, M. (2000) Growing Semantic Grammar. *Ph.D Dissertation*, Carnegie Mellon University.
68. Gini, C. (1921). "Measurement of Inequality of Incomes". *The Economic Journal* 31: 124–126.
69. Glass, J.R. (1999) Challenges for Spoken Dialogue Systems. In *Proceedings of the 1999 IEEE ASRU Workshop*. <http://www.sls.lcs.mit.edu>.
70. Glass, J. and Weinstein, E. (2001) SPEECHBUILDER: Facilitating Spoken Dialogue System Development. In *Eurospeech 2001*, Aalborg, Denmark.
71. Glass, J., Weinstein, E., Cyphers, S., Polifroni, J., Chung, G., and Nakano, N. (2004) A Framework for Developing Conversational User Interfaces. In *Proceedings of CADUI*, Funchal, Isle of Madeira, Portugal, 347-358.
72. Gorelov, Z. (2005) Interview with Zor Gorelov of Tell-Eureka. [http://www.telleureka.com/about\\_us/02news\\_11\\_25.htm](http://www.telleureka.com/about_us/02news_11_25.htm)
73. Gruenstein, A. (2002) Automatic Grammar Construction. Symbolic Systems, Stanford University, 115.
74. Han, B. (2000) Improving Spoken Dialog Systems, *Technical Report*, Language Technology Institute, Carnegie Mellon University.
75. Harper, M., White, C., Wang, W. and Johnson, M. (2000) The Effectiveness of Corpus-Induced Dependency Grammars for Post-processing Speech. In *Proceedings of the First Annual Conference of the North American Chapter of the Association for Computational Linguistics (NAACL)*. 102-109.
76. Harris, T.E. (1963) The Theory of Branching Processes. *Berlin-Gottin-gen-Heidelberg: Springer-Verlag*.

77. Hartzell, D. (2003) Simplifying Speech-Enabled Solutions or Deploying Speech-Enabled Services Should Be as Easy as Deploying Web Services. Invited talk at the Conference on Voice Enabled Services.
78. Hauptmann, A.G., Young, S.R., and Ward, W.H. (1988) Using Dialog-Level Knowledge Sources to Improve Speech Recognition. *Proceedings of the Seventh National Conference on Artificial Intelligence*, Morgan Kaufmann,
79. Hermannsdottir, S.H. (1996) A Dialogue Manager for A Spoken Dialogue System. *McGill University (Canada) MSC thesis*, MAI 34/04, 1616.
80. Hewlett Packard (HP) (2005) Developing Voice-Enabled Web Services. *Hewlett Packard (HP) white paper*.  
<http://whitepapers.techrepublic.com.com/whitepaper.aspx?docid=81819>
81. Honavar, V. and Slutzki, G. (1998) Grammatical Inference, *LNAI 143*, Springer-Verlag.
82. Hone, K. S. and Barber, C. (1995) Using A Simulation Method to Predict the Transaction Time Effects of Applying Alternative Levels of Constraint to User Utterances Within Speech Interactive Dialogues, *ESCA Workshop on Spoken Dialogue Systems*. 8. 1 209-212.
83. IBM (2005) WebSphere VoiceXML Programmer's Guide - Voice Server for Multiplatforms.  
<http://publib.boulder.ibm.com/infocenter/pvcvoice/51x/topic/com.ibm.voicetools.reldocs.doc/pgmguide.pdf> .
84. IBM Software Download (2006),  
[http://www14.software.ibm.com/webapp/download/preconfig.jsp?id=2006-02-23+09%3A08%3A30.953556R&cat=&fam=&s=z&S\\_TACT=104AH%20W42&S\\_CMP](http://www14.software.ibm.com/webapp/download/preconfig.jsp?id=2006-02-23+09%3A08%3A30.953556R&cat=&fam=&s=z&S_TACT=104AH%20W42&S_CMP)
85. IBM WebSphere Voice Toolkit (2010):  
[http://www-306.ibm.com/software/pervasive/voice\\_toolkit/about/#keyfeatures](http://www-306.ibm.com/software/pervasive/voice_toolkit/about/#keyfeatures) .
86. IBM WebSphere Voice Server:  
[http://www-01.ibm.com/software/pervasive/voice\\_server/](http://www-01.ibm.com/software/pervasive/voice_server/)
87. IEEE Industry Standards and Technology Organization (IEEE-ISTO) (2002) Some Thoughts on Speech Grammar.



- [http://www.voicexmlreview.org/Jan2002/columns/Jan2002\\_speak\\_listen.html](http://www.voicexmlreview.org/Jan2002/columns/Jan2002_speak_listen.html) .
88. Intervoice Inc. (2004) Intervoice Training Document – Voice User Interface Design – Speechworks 7.0 OSS/OSR and Naunce 8.0 – Speech Forms, 2004.
89. Jackson, P. (2004) Speaker and Speech Recognition Website: <http://www.ee.surrey.ac.uk/Personal/P.Jackson/eem.ssr/>
90. Jørgensen, F (2007) The Effects of Disfluency Detection in Parsing Spoken Language. *NODALIDA 2007 Conference Proceedings*, 240–244.
91. Joshi, A.K., Levy, L., and Takahashi, M. (1975) Tree Adjunct Grammars. *Journal of Computer and System Sciences*, 10(1), 136-163.
92. Ju, Y., Ollason, D.G., and Bhatia, S. (2009) Method and Apparatus for Automatic Grammar Generation from Data Entries. *United States Patent* 7636657. Primary Class: 704/10, International Classes: G06F17/21.
93. Justesen, J. and Larsen, K. (1975) On Probabilistic Context-Free Grammars that Achieve Capacity. *Inform. Contr.*, 29, 268 -285.
94. Kent, J.T. (1983) Information Gain and a General measure of Correlation. *Biometrika* 1983, 70(1), 163-173.
95. Kathrin Klamroth (2006) Measuring Distances. Springer New York. ISBN: 978-0-387-95498-1 (Print) 978-0-387-22707-8 (Online).
96. Knight, S., Gorrell, G., Rayner, M., Milward, D., Koeling, R. and Lewin, I. (2001) Comparing Grammar-Based and Robust Approaches to Speech Understanding: A Case Study. In *Eurospeech 2001*. 1779-1782.
97. Koza, J.R. (1992) Genetic Programming: On the Programming of Computers by Natural Selection. *MIT Press*.
98. Koza, J.R. (1994) Genetic Programming as a Means for Programming Computers by Natural Selection, in journal *Statistics and Computing*, 4(2):87—112.
99. Kuich, W. (1970) On the Entropy of Context-Free Languages. *Inform Contr.*, 16, 173-200.
100. Kullback, S. (1959), Information Theory and Statistics. *John Wiley and Sons*, NY.



101. Kupersstein, M. (2005) Building Professional Speech Applications for Mainstream IT Developers.  
<http://www.microsoft.com/speech/community/newsletter/articles/0905article.htm> .
102. Lamel, L., Bennacef, S., Rosset, S., Devillers, L., Foukia, S., Gangolf, J., and Gauvain, J. (1997) The LIMSI RailTel System: Field Trial of A Telephone Service for Rail Travel Information. *Speech Communication*, 23, 67-82.
103. Levin, E. and Mané, A. (2005) Voice User Interface Design for Automated Directory Assistance. In INTERSPEECH-2005. 2509-2512.
104. Lynch, P. and Brizee, A. (2010) Purdue Owl Online Writing Lab,  
<http://owl.english.purdue.edu/owl/resource/540/01>
105. MacFadyen, H. (2010)  
<http://www.uottawa.ca/academic/arts/writcent/hypergrammar/partsp.html>,  
University of Ottawa.
106. Mané, A. and Levin, E. (2005) Designing the Voice User Interface for Automated Directory Assistance, *Voice Advantage*, USA, City University of New York, USA.
107. Marcus, S., Brown, D., Goldberg, R., Schoeffler, M., Wetzell, W., and Rosinski, R. (1996) Prompt Constrained Natural Language-evolving the Next Generation of Telephony Services, *In Proceedings of Fourth International Conference on Spoken Language*, 2, 857-860.
108. McCabe T.J. (1976) A Complexity measure. *IEEE Transactions on Software Engineering*. 2(4):308-320.
109. McCandless and M., Glass, J. (1993) Empirical Acquisition of Word and Phrase Classes in the ATIS Domain. *The 3<sup>rd</sup> European Conference on Speech Communication and Technology*. 981-984.
110. McTear, M.F. (2002) Spoken Dialogue Technology: Enabling the Conversational User Interface. *ACM Press New York, NY, USA*, Periodical-Issue-Article, 2002, ISSN: 0360-0300, 90-169.
111. Meng, H. and Siu, K.C. (1999) Semi-Automatic Acquisition of Domain-Specific Semantic Structures. *Proceedings of the 6th European Conference on Speech Communication & Technology*, Budapest, Hungary, 2039-2042.

112. Meng, H. and Siu, K. C. (2002) Semiautomatic Acquisition of Semantic Structures for Understanding Domain-Specific Natural Language Queries (2002). *IEEE Transactions on Knowledge and Data Engineering*, 14(1), 172-181.
113. Mernik, M., Gerlić, G., Zumer, V., and Bryant, B. (2003) Can a parser be generated from Examples? *Proceedings of the ACM Symposium on Applied Computing*, Melbourne, 1063-1067.
114. Metz, B.D. (2008) Automatic Grammar Tuning Using Statistical Language Model Generation. *United States Patent* 20080052076, Kind Code: A1.
115. Miclet, L. and Higuera, C.D. (1996) Grammatical Inference: Learning Syntax from Sentences, *LNAI 1147*, Springer-Verlag.
116. Microsoft Corporation (2005). Microsoft Speech Application Software Development Kit (SASDK) Version 1.0. <http://www.microsoft.com/downloads/details.aspx?FamilyID=1194ed95-7a23-46a0-bbbc-06ef009c053a&DisplayLang=en> .
117. Microsoft Download Center (2010). <http://www.microsoft.com/downloads/details.aspx?FamilyId=5DAAE9C4-188C-4547-A9D6-1671132A39A1&displaylang=en&EventType=getsdk>
118. Miller, M.I. and O'Sullivan, J.A. (1992) Entropies and Combinatorics of Random Branching Processes and Context-free Languages. *IEEE Transactions on Information Theory*, 38(4), 1292-1310.
119. Moore, R.C. (1999) Using Natural-Language Knowledge Sources in Speech Recognition, *Research Institute for Advanced Computer Science NASA Ames Research Center*, Moffett Field, CA 94035.
120. Moore, R.C., Appelt, D., Dowding, J., Gawron, J.M., and Moran, D. (1995) Combining Linguistic and Statistical Knowledge Sources in Natural-Language Processing for ATIS. In *Spoken Language Systems Technology Workshop*, Austin, Texas, February 1995. Morgan Kaufmann Publishers, Inc., 261-264.

121. Morimoto, T. and Takahashi, S (2009) Automatic Construction of a FSA Language Model and Speech Recognition on it with Dynamic Alternative Path Search. *Proceedings of the International MultiConference of Engineers and Computer Scientists 2009 (IMECS 2009)*, March 18-20, 2009, Hongkong, 1, ISBN: 978-988-17012-2-0.
122. Morimoto, T. and Takahashi, S. (2008) Automatic Construction of FSA Language Model for Speech Recognition by FSA DP-Matching, *Trends in Intelligent Systems and Computer Engineering (Edtd. by O. Castillo et al.)*, Springer, 515-524.
123. National Research Council (U.S.) (1984) Automatic Speech Recognition In Severe Environments. *National Academy Press*, Washington, D.C., 1984.
124. Nuance (December 2003) Grammar Developer's Guide, Nuance Speech Recognition System Version 8.5.  
<http://community.voxeo.com/vxml/docs/nuance20/grammar.pdf> .
125. NÜ echo Inc. (2005) <http://www.nuecho.com/fr/services/grammar.shtml> .
126. Olsen, D.R. and Klemmer, S.R. (2005). The Future of User Interface Design Tools, *ACM CHI 2005 Workshop*, 2134 - 2135.
127. Omohundro, S. (1992) Best-first Model Merging for Dynamic Learning and Recognition. In *Advances in Neural Information Processing Systems 4*, ed. By John E. Moody, Steve J. Hanson, & Richard P. Lippman, San Mateo, CA: Morgan Kaufmann. 958-965.
128. Oncina, J. and Garcia, P. (1992) Inferring Regular Languages in Polynomial Update Time. *Series in Machine Perception and Artificial Intelligence*, World Scientific, 1, 49-61.
129. Opera Software ASA (2010) <http://www.opera.com/>.
130. Paineni, K., Roukos, S., Ward, T., and Zhu and W.J. (2002) BLEU: A Method for Automatic Evaluation of Machine Translation. In ACL-2002: 40<sup>th</sup> Annual meeting of the Association for Computational Linguistics. 311- 318.

131. Pargellis, A., Zhou, Q., Saad, A. and Lee, C. (1998) A Language for Creating Speech Applications, in *Proceedings ICSLP*, Sydney, Australia. Paper 0388.
132. Pargellis, A., Kuo, H., and Lee, C. (1999a) Automatic Dialogue Generator Creates User Defined Applications, In *Proceedings EUROSPEECH*, Budapest, Hungary.
133. Pargellis, A., Kuo, H., and Lee, C. (1999b) Automatic Application Generator Matches User Expectations to System Capabilities. *Proceedings ESCA Workshop Inteerac. Dialog. Multi-Modal System*, Kloster Irsee, Germany. 37-40.
134. Pargellis, A., Fosler-Lussier, E., Potamianos, A., and Lee, C.-H. (2001) Metrics for Measuring Domain Independence of Semantic Classes. In *Eurospeech 2001*. Aalborg, Denmark. 447-450.
135. Peissner, M. (2002) What the Relationship between Correct Recognition Rates and Usability, *Proceedings of the 6<sup>th</sup> International Scientific Conference on Work with Display Units (WWDU –World Wide Work)*, Berchtesgaden.  
[http://www.ergonomic.de/files/wwdu\\_2002\\_proceedings.pdf](http://www.ergonomic.de/files/wwdu_2002_proceedings.pdf) .
136. Pellom, B., Ward, W., and Pradhan, S. (2000) The CU Communicator: An Architecture for Dialogue Systems. *Proceedings of International Conference on Spoken Language Processing*. Beijing, China, 2,. 723-726.
137. Pereira, F. and Schabes, Y. (1992) Inside-outside Reestimation from Partially Bracketed Corpora. In *20<sup>th</sup> Meeting of the Association for Computation Linguistics (ACL'92)*. 128-135.
138. Pieraccini, R. (2004) Spoken Language Understanding, the Research/Industry Chasm. In *HLT/NAACL Workshop on Spoken Language Understanding for Conversational Systems*, Boston, 2004. 47-47.

139. Polifroni, J., Chung, G., and Seneff, S. (2003) Towards Automatic Generation of Mixed-Initiative Dialogue Systems from Web Content. *Proceedings of Eurospeech '03*, Geneva, Switzerland, 193-196.
140. Potamianos, A., Kuo, H. Lee, C., Pargellis, A., Saad, A., and Zhou, Q. (1999) Design Principles and Tools for Multimodal Dialog Systems. *In Proceedings of ESCA Workshop. Interact. Dialog. Multi-Modal System*, Kloster Irsee, Germany, June 1999, 167-170.
141. Power, J.F. and Malloy, B.A. (2000) Metric-based Analysis of Context-Free Grammars, *Proceedings Program Comprehension, IWPC 2000*, 8th International Workshop, 171-178.
142. Power, J.F. and Malloy, B.A. (2001) Exploiting Metrics to Facilitate Grammar Transformation into LALA Format, *Proceedings of the 2001 ACM Symposium on Applied Computing*. Las Vegas, Nevada, United States. 636-640.
143. Power, J.F. and Malloy, B.A. (2004) A Metrics Suite For Grammar-based Software. *Journal of Software Maintenance*. 16 (6): 405-426.
144. Rahmel, H. (2005) Strategies for Optimizing Alphanumeric Recognition Accuracy. PM Speech Components Group.  
<http://www.microsoft.com/speech/community/newsletter/articles/124004art/index.htm> .
145. Rayner, M., Hockey, B.A., James, F., Bratt, E.O., Goldwater, S., and Gawron, J.M. (2000) Compiling Language Models From A Linguistically Motivated Unification Grammar. In *COLING 2000*.
146. Riccardi, G. and Gorin, A. (2000) Stochastic Language Adaptation over Time And State in Natural Spoken Dialog Systems. *IEEE Transactions. On Speech and Audio Processing*, 8, 3-10.
147. Rosenfeld. R. (2000a) Two Decades of Statistical Language Modeling: Where Do We Go From Here? In *Proceedings of the IEEE, 2000*,. 88, 1270-1278.

148. Rosenfeld, R. (2000b) Incorporating Linguistic Structure into Statistical Language Models. In *Philosophical Transactions of the Royal Society of London A*, 358, 2000, 1311-1324.
149. Rudinsky, J., Mikula, T., Kencl, L., Jakub, D., and Garcia, X. (2009) Voice2Web: Architecture for Managing Voice-Application Access to Web Resources. *Proceedings of the 12<sup>th</sup> IFIP/IEEE International Conference on management of Multimedia and Mobile Networks and Services: Wired-Wireless Multimedia Networks and Services Management*. Venice, Italy 118-131.
150. Rudnicky, A., Bennet, C., Black, A., Chotomongcol, A., Lenzo, K., Oh, A., and Singh, R. (2000) Task And Domain Specific Modelling in the Carnegie Mellon Communicator System. *Proceedings of International Conference on Spoken Language Processing*. Beijing, China, 2, 130-133.
151. Rugelbak, J., Johnsen, F., and Knudsen, J. (2001) Experiences with a Norwegian Voice Controlled e-mail reader, Fornebu : Telenor Forskning og Utvikling. - 39 s. - (Telenor FoU ; N 38/2001)
152. Schramm, H., Rueber, B., and Kellner, A. (2000) Strategies for Name Recognition in Automatic Directory Assistance Systems. *Speech Communication*. 31(4), 329-338.
153. Seneff, S. (1992) TINA: A Natural Language System for Spoken Language Applications. *Computational Linguistics* 18(1), 61-86.
154. Seneff, S., McCandless, M., and Zue, V. (1995) Integrating Natural Language Into the Word Graph Search For Simultaneous Speech Recognition And Understanding. In *4th European Conference on Speech Communication and Technology (Eurospeech'95)*, 1781-1784.
155. Sereni, D. (2006) Termination Analysis for Higher-Order Functional Programs. *Doctoral thesis*. Oxford University Computing Laboratory Programming Research Group.
156. Shi, Y. (2003a) Using Natural Language Features to Improve Speech Recognition Accuracy. *60-510 Survey Report*. The School of Computer Science, University of Windsor, Canada.

157. Shi, Y. (2003b) An Investigation of Grammar Design in Natural-Language Speech-Recognition. *Master Thesis*. School of Computer Science, University of Windsor, ON, Canada.
158. Shi, Y. and Frost, R. (2004) An Investigation of Grammar Design in Natural-language Speech Recognition. *Proceedings of AI'2004, the Seventeenth Canadian Conference on Artificial Intelligence*. London, Ontario, Canada, 569-570.
159. Siu, K. and Meng, H. (1999) Semi-automatic Acquisition of Domain-specific Semantic Structures, *Proceedings of the 6th European Conference on Speech Communication & Technology*, Budapest, Hungary. 2039-2042.
160. Siu, K. C. and Meng, H. (2001) Semi-Automatic Grammar Induction for Bi-directional English-Chinese Machine Translation. *Proceedings of the 7th European Conference on Speech Communication & Technology (EUROSPEECH-2001)*, Scandinavia, 2749-2752.
161. Siu, K. C., Meng, H., and Wong, C. C. (2003) Example-based Bi-directional Chinese English Machine Translation with Semi-automatically Induced Grammars. *Proceedings of the 8th European Conference on Speech Communication and Technology (EUROSPEECH-2003)*, Geneva, Switzerland, 2801-2804.
162. Soule, S. (1974) Entropies of Probabilistic Grammars. *Inform. Contr.*,25, 57-74.
163. Srinivas, B., Sarkar, A., Doran, C. Hockey, B.A., and Joshi, A. (1996). An Approach to Robust Partial Parsing and Evaluation Metrics. In *Proceedings of the Workshop on Robust Parsing at European Summer School in Logic Language and Information*. 70-82.
164. Srinivas, B., Sarkar, A., Doran, C. and Hockey, B.A. (1998) Grammar and Parser Evaluation in the XTAG Project, *Workshop on the Evaluation of Parsing Systems*. Granada, Spain.
165. Stolcke, A. and Omohundro, S.M. (1994a) Inducing Probabilistic Grammars by Bayesian Model Merging. In *Proceedings of the Second*



- International Colloquium on Grammatical Inference and Applications (ICGI-94)*. 106-118.
166. Stolcke, A., and Omohundro, S.M. (1994b) Best-first Model Merging for Hidden Markov Model Induction. *TR-94-003, ICSI Technical Report*, Berkeley, CA.
  167. Sun Microsystems (1998a) Java Speech API Programmer's Guide. <http://java.sun.com/products/java-media/speech/forDevelopers/jsapi-guide/Preface.html> .
  168. Sun Microsystems (1998b) Java Speech Grammar Format Specification. <http://java.sun.com/products/java-media/speech/forDevelopers/JSGF/JSGF.html> .
  169. Sun Microsystems, Inc. (2000) Jspeech Grammar Format, *W3C Note* 05 June 2000.
  170. Takezawa, T., Kita, K., Hosaka, J., and Morimoto, T. (1991) Linguistic Constraints for Continuous Speech Recognition in Goal-Directed Dialogue. In *Proceedings of IEEE International Conference on Acoustics, Speech and Signal Processing*. 801-804.
  171. Tellme Networks Inc., <https://studio.tellme.com/> .
  172. Trias Sanz, R. and Marino, J. (2002) Bassurde [lite], a Machine-driven Dialogue System for Accessing Railway Timetable Information. *Proceedings of International Conference on Spoken Language Processing*. Denver Colorado, USA, 2685-2689.
  173. Tverra, K. (2004) Vox et praeterea nihil, *Master's Thesis*, Department of Informatics, University of Bergen.
  174. Turunen, M. (2004) Speech Interface Design, *Master's thesis*, University of Tampere, Finland.
  175. Vidal, E., Casacuberta, F., and Garcia, P. (1993) Grammatical Inference and Applications to Automatic Speech Recognition and Understanding. *Tech. Rep. DSIC II/41/93*, Departamento de Sistemas Informáticos y Computación, Universidad Politécnica de Valencia.



176. Vidal, E. (1994) Grammatical Inference: An Introductory Survey. *Grammatical Inference and Applications (ICGI-94)*. Springer, Berlin, Heidelberg, 1-4.
177. VoiceXML Forum (2004)  
<http://www.voicexml.org/specs/multimodal/x+v/12/>.
178. VOXEO corporation (2006) VoiceXML 2.1 Development Guide.  
[http://docs.voxeo.com/voicexml/n2.0/frame.jsp?page=t\\_16.htm](http://docs.voxeo.com/voicexml/n2.0/frame.jsp?page=t_16.htm)
179. Voxpilot company, <http://www.voxpilot.com/> .
180. W3C (World Wide Web Consortium) (2000). *JSpeech Grammar Format*. Sun Microsystems submission to W3C. <http://www.w3.org/TR/jsqf/>.
181. W3C (World Wide Web Consortium) (2001) Stochastic Language Models (N-Gram) Specification, <http://www.w3.org/TR/ngram-spec/> .
182. W3C (World Wide Web Consortium) (2004) Speech Recognition Grammar Specification, <http://www.w3.org/TR/speech-grammar/> .
183. W3C (World Wide Web Consortium) (2007a) Voice Extensible Markup Language (VoiceXML) 2.1, *W3C Recommendation 19 June 2007*, <http://www.w3.org/TR/voicexml21/>
184. W3C (World Wide Web Consortium) (2007b) XHTML™ 1.1 - Module-based XHTML - Second Edition, *W3C Working Draft 16 February 2007*, <http://www.w3.org/TR/xhtml11> .
185. W3C (World Wide Web Consortium) (2009) Extensible Markup Language (XML) <http://www.w3.org/xml>.
186. Waibel, A. (1996) Interactive Translation of Conversational Speech. *Computer*,.29.
187. Waibel, A. and Lee, K.F. (1990) Readings in Speech Recognition. *Morgan Kaufmann Publishers, Inc.*, ISBN 1-55860-124-4.
188. Walker, M. (2001) Darpa Communicator Dialogue Travel Planning Systems: the June 2000 Data Collection. *Proceedings of EUROSPEECH*. Aalborg, Denmark, 1371-1374.
189. Wang, Y.-Y. (1998) Grammar Inference and Statistical Machine Translation (1998). *Ph.D. thesis*, Camegie Melon University.

190. Wang, Y.-Y. and Waibel, A. (1998). Modeling with Structures in Statistical Machine Translation. In *Proceedings of the 36th Annual Meeting of the Association for Computational Linguistics and 17th International Conference on Computational Linguistics Montreal (COLING-ACL '98)*, Canada. 1357-1363.
191. Wang, Y., Mahajan, M., and Huang, X. (2000). A Unified Context-Free Grammar and N-Gram Model for Spoken Language Processing. In *Proceedings of the International Conference on Acoustics, Speech, and Signal Processing (ICASSP-2000)*. Istanbul, Turkey. 1639-1642.
192. Wang, Y.-Y. (2001) Robust Spoken Language Understanding in MiPad. In *Eurospeech 2001*, Aalborg, Denmark. 1555-1558.
193. Wang, Y.-Y. and Acero, A. (2001). Grammar Learning for Spoken Language Understanding. In *Proceedings of ASRU Workshop*. Madonna di Campiglio, Italy. 292-295.
194. Wang, Y.-Y. and Acero, A. (2002) Evaluation of Spoken Language Grammar Learning in the ATIS Domain. In *Proceedings of the International Conference on Acoustics, Speech, and Signal Processing (ICASSP'02)*. Orlando, Florida. 1, 41-44.
195. Wang, Y.-Y., Acero, A., Chelba, C., Frey, B., and Wong, L. (2002) Combination of Statistical and Rule-Based Approaches for Spoken Language Understanding. In *Proceedings of the International Conference on Speech and Language Processing (ICSLP)*. Denver, Colorado. 609-612.
196. Wang, Y.-Y., and Acero, A. (2003a) Concept Acquisition in Example-Based Grammar Authoring. In *Proceedings of the International Conference on Acoustics, Speech, and Signal Processing*. Hong Kong, China. 284-287.
197. Wang, Y.-Y., and Acero, A. (2003b) Combination of CFG and N-gram Modeling in Semantic Grammar Learning. In *Proceedings of the EUROSPEECH Conference (EUROSPEECH-2003)*. Geneva, Switzerland. 2809-2812.

198. Wang, Y.-Y. and Ju. Y-C. (2004) Creating Speech Recognition Grammars from Regular Expressions for Alphanumeric Concepts. *In Proceedings of the Proceedings of the International Conference on Speech and Language Processing*. Jeju Island, Korea. 2161-2164.
199. Wang, Y.-Y. and Acero, A. (2005): SGStudio: Rapid Semantic Grammar Development for Spoken Language Understanding. In *INTERSPEECH-2005*, 1869-1872.
200. Wang, Y.-Y. and Acero, A. (2006) Rapid Development of Speech Recognition Grammars. *Communication*, 48(3-4).
201. Ward, W. (1991) Understanding Spontaneous Speech: the Phoenix System. *ICASSP*, Toronto, Canada.
202. Ward, W. (1990) The CMU Air Travel Information Service: Understanding Spontaneous Speech. In *Proceedings of the DARPA Speech and Natural Language Workshop*, 127–129.
203. Ward, W. (1994) Recent Improvements in the CMU Spoken Language Understanding System. In *Human Language Technology Workshop*, Plainsboro, New Jersey. 213-216.
204. Wasinger, R. M. (2001) Dialog Based User Interfaces Featuring A Home Cooking Assistant. *Master's Thesis*, School of Electrical and Information Engineering, the University of Sydney.
205. Weber, H. and Görz, G. (1999) Symbolic Parsing and Probabilistic Decision Making. The Speech and language Experience with Hybrid Information Processing. In *Proceedings of HCI (2)*, April 6, 1999, 802-806.
206. Wong, C.-C. and Meng, H. (2001) Improvements on a Semi-automatic Grammar Induction Framework. In *IEEE Automatic Speech Recognition and Understanding Workshop (ASRU'01)*. Madonna di Campiglio, Italy. 288-291.
207. Wong, C.C., Meng, H., and Siu, K.C. (2001) Learning Strategies in a Grammar Induction Framework. *Proceedings of the Natural Language Processing Pacific Rim Symposium (NLPRS 2001)*, Tokyo, 153-157.

208. Xia, F (2001) Automatic Grammar Generation From Two Different Perspectives. *Doctoral Dissertation*, Computer and Information Science, the University of Pennsylvania. USA.
209. Yankelovich, N. (1997) Using Natural Dialogs as the Basis for Speech Interface Design, *Submitted to MIT Press as a chapter for the upcoming book, "Automated Spoken Dialog Systems," edited by Susann Luperfoy. Copyright 1997, Sun Microsystems, Inc.*
210. Young, S. R., (1990) Use of Dialog, Pragmatics and Semantics to Enhance Speech Recognition. *Speech Communication*, 9, 1990.
211. Young, S. R., Ward, W. H., and Hauptmann, A. G. (1989) Layering Predictions: Flexible Use of Dialog Expectation in Speech Recognition. *Proceedings of IJCAI-89*, Detroit. 1543-1549.
212. Young, S. and Matessa (1991) Using Pragmatic And Semantic Knowledge to Correct Parsing of Spoken Language Utterances. *Eurospeech-91*. 223-227.
213. Yu, D., Ju, Y.-C., Wang, Y., and Acero A. (2006). N-Gram Based Filler Model for Robust Grammar Authoring. In *Proceedings of the International Conference on Acoustics, Speech, and Signal Processing (ICASSP)*. 1, 565-568.
214. Zadrozny, W. and Kambhatla, N. (1999) Method and Apparatus for Creating Speech Recognition Grammars Constrained by Counter Examples. *United States Patent 59373*. Primary Class: 704/257, other classes: 704/255, 704/E15.022, 704/251, International Class: G10L15/18; G10L15/00; G10L5/06; G10L9/00
215. Zhang, J., Ward, W., Pellom, B., Yu, X., and Hacioglu, K. (2001) Improvements in Audio Processing and Language modeling in the CU communicator. *Proceedings of EUROSPEECH*. Aalborg. Denmark, 2209-2212.
216. Zue, V. Seneff, S. Glass, J.R. Polifroni, J. Pao, C. Hazen, T.J. and Hetherington, L. (2000) JUPITER: A telephone-Based

Conversational Interface for Weather Information. *IEEE Transactions on Speech and Audio Processing*. 85-96.

## **APPENDIX A: A SURVEY- DESIGN OF RECOGNITION GRAMMAR FOR VXML-LIKE APPLICATIONS**

### Table of Contents of the Survey

1. INTRODUCTION
  - 1.1 VoiceXML
  - 1.2 Grammars
  - 1.3 Spoken-Dialogue Systems
2. THE CHALLENGE OF GRAMMAR DESIGN
3. ISSUES IN GRAMMAR DESIGN
  - 3.1 Guidelines for Grammar Design
  - 3.2 Dialog Design
  - 3.3 Prompt Design
  - 3.4 Sub-Grammar Design
  - 3.5 Sub-Dialog Design
  - 3.6 Grammar Weights and Probabilities
  - 3.7 Error Handling
  - 3.8 Summary
4. VOICE USER INTERFACE (VUI) DESIGN
  - 4.1 Issues Related to VUI
  - 4.2 Summary
5. TESTING
  - 5.1 Testing Issues
  - 5.2 Summary
6. TOOLS AND ENVIRONMENTS
  - 6.1 Basic VoiceXML Development Environments
  - 6.2 NÜ Echo Grammar Environment
  - 6.3 IBM WebSphere Voice Toolkit
  - 6.4 Microsoft Speech Application SDK (SASDK)
  - 6.5 MCM Toolkit
  - 6.6 Philips Speech SDK
  - 6.7 Hewlett-Packard (HP) OCMP VXML Developer Toolkit
  - 6.8 CSLU Toolkit
  - 6.9 Chant Developer Workbench
  - 6.10 Fonelet's VoiceXML Toolkit
  - 6.11 Wizard of OZ (WOZ)
  - 6.12 BeVocal Café
  - 6.13 Tellme Studio

6.14 VoxPilot Open media Platform

6.15 Summary

## 7. CONCLUSION

## 1. INTRODUCTION

Over the past three decades, speech-recognition technologies have achieved significant developments. A large number of spoken-dialogue systems have been implemented. Aust *et al* (1995) present Philips system. Jupiter system (weather information system) (Zue *et al*, 1997 and Zue *et al*, 2000) and the AT&T's call redirection system (Riccardi and Gorin, 2000) are the pioneer systems. Other examples include the ARISE project (Lamel *et al*, 2000; Baggia *et al*, 2000) and Philips Directory-Assistance system (directory information service) (Schramm *et al*, 2000). Also, an important American project, the DARPA Communicator, has attracted the most important research organizations in USA, such as MIT, BBN, Carnegie and Mellon University (Rudnicky *et al*, 2000 and Carpenter *et al*, 2001), University of Colorado (Pellom *et al*, 2000; Zhang *et al*, 2001), AT&T (Walker, 2001), Bell Labs, SRI and IBM (Gao *et al*, 2001). Trias and Marino (2002) discuss BASURDE [LITE] system, the train travel information and ticket reservation services.

Rather than a graphic user interface, voice applications are applications with spoken input and/or output. There is overwhelming information flowing through the Internet nowadays, and many business transactions are conducted through the web. VoiceXML (VXML) makes it possible to access the Internet via voice (e.g. phone). More and more companies are recognizing speech as an integral part of their IT solutions (HP, 2005).

Currently, stochastic (statistical) language models and grammar-based language models are two mainstreams in Natural Language Understanding (NLU) research. Statistical language models have the advantages of simplicity, flexibility, adaptation, higher recognition accuracy, and robust performances. The primary disadvantage is the costly collection of huge amounts of training data and poor generalization with insufficient data. In addition, Statistical language models are not supported by commercial systems, such as VoiceXML browsers.



As an alternative solution to statistical techniques, grammar-based speech recognition is more expressive, more common and easier to use with reasonable recognition accuracy for small domains. An important advantage over statistical approaches is that grammar-based approaches do not require a large amount of training data which is difficult and expensive to collect.

Knight et al. (2001) declare that, statistical language models were popular around 1995, whereas grammar-based language models took the pre-eminent position in commercial products by 2001. By defining sets of rules, grammars define the utterances, phrases, and words that are accepted by the speech application. Effective grammars are a critical component of grammar-based speech applications (Nuance, 2003). Therefore, the need for guidelines for grammar design for VoiceXML-like applications is imperative. This survey aims to provide a comprehensive review of research and development in this area.

This survey is organized as follows. Section 1 briefly reviews the concepts of VoiceXML, recognition grammars, and spoken-dialog systems. Section 2 discusses the current challenges of grammar design. Section 3 focuses on the detailed principles and guidelines in grammar design. Section 4 considers the issues in Voice User Interface (VUI) design. The issues related to testing are discussed in section 5. Section 6 discusses tools and environments for speech-application development. Section 7 concludes the survey.

### **1.1 VoiceXML (VXML)**

The Voice eXtensible Markup Language (VoiceXML) is an industry standard defined by the World Wide Web Consortium (W3C, 2005). VoiceXML is an XML-based markup language for building distributed voice applications, much as HTML is a markup language for creating distributed visual applications. The structure of VoiceXML is similar to that of HTML, which allows web developers to write voice-enabled applications with ease.

VoiceXML provides features to collect spoken and DTMF (Dual Tone Multiple Frequency) input, generate synthesized audio prompts, control dialog flow and ECMA scripting, handle asynchronous events, record and play audio, and control basic telephone connections (HP, 2005). A VoiceXML *application* is built from one or more VoiceXML *documents* with the same application root document. Each document contains a variety of VoiceXML instructions for the application. The information in the root document is available to all of the documents in the application. The root document is loaded whenever one of the application's documents is loaded, and remains loaded as long as the application is active.

VoiceXML documents define applications as a set of *dialog states*. At any time, the user is either in a state or being transitioned to a state. A dialog may include several discrete dialog elements, called forms or menus. A *form* defines an interaction that collects information from the user, and makes the transition to a new state based on this information. A *menu* is essentially a form with only one piece of information to gather. For example, a menu presents the user with a set of choices. Based on the choice the user made, s/he is transitioned to another state of the application.

Therefore, a VoiceXML application or document constitutes a conversational finite state machine, moving the user from one state to the next. Each transition is determined by the dialog element at the time. The transitions are specified using Unified Resource Identifiers (URI), which can point to another form in the same document, another document, or to a document in a completely different application. Execution is terminated when a dialog does not specify a successor, or when all dialog elements in the current document have been visited, or if an explicit exit command ends the dialog. Events are thrown when certain conditions are detected.

## 1.2 Grammars

A grammar is a fundamental building block of speech technology (Mané and Levin, 2005). A *grammar* is a set of rules that define the possible words, phrases, or utterances which are accepted by the speech recognition engine. The main rule of a grammar is called the “root” rule, which acts as an entry point in the grammar. A rule can reference other rules, or list combinations of equivalent alternative wordings, repetitions or optional parts. A grammar may be trivial lists of possible words, or a set of rules defining complex sets of phrases.

Grammars may be incorporated into the application code as *inline grammars*, or be externally available as *external grammar* files. Inline grammars are typically small and uncomplicated. External grammars are usually larger and non-trivial. The advantages of using an external grammar are that, it is shareable among multi-applications, which eliminates the need to maintain several identical large grammars. Another advantage of external grammars is that they do not need to change with the changes of VoiceXML code.

A grammar can be defined statically, or dynamically using the technology to build dynamic HTML, such as CGI scripts, Java Beans, servlets, ASPs, and JSPs. In addition, some grammars are so common that they have been incorporated into the VoiceXML interpreter, such as those defining Boolean values and Dates.

Currently, there are several grammar formats available in grammar creation for voice applications. Speech Recognition Grammar Specification (SRGS) is the only standard for Automatic Speech Recognition (ASR) grammars (Baggia, 2006). The details of the W3C Speech Recognition Grammar Specification are available at (W3C, 2004). It was accepted as a W3C Recommendation in March 2004, which means that many companies demonstrated it to be easy to implement, and gave support to its development. The two grammar formats included in SRGS are:

- (1) XML format with an enforced syntax expressed both by a DTD and a schema;
- (2) ABNF (Augmented Backus-Naur Form) format, which is a textual and concise encoding of a grammar.

Both the ABNF Form and XML Form have the expressive power of a Context-Free Grammar (CFG) (W3C, 2004). ABNF format is suitable for quick hand coding, while XML is easily handled in automatic environments and is more suitable for integrating into XML-based Voice User Interface (VUI) design languages, i.e. VoiceXML 2.0.

SRGS is modeled on the Java Speech Grammar Format specification (JSGF), which is owned by Sun Microsystems, Inc., California, U.S.A. (W3C, 2005). JSGF is a platform-independent, vendor-independent textual representation of grammars for use in speech recognition. It adopts the style and conventions of the Java Programming Language in addition to use of traditional grammar notations. The textual representation is readable and editable by both developers and computers, and can be included in Java source code (Sun, 1998b).

Nuance (2003) extended the XML grammar as Grammar Specification Language (GSL). IBM Voice Toolkit supports XML and ABNF grammar formats (IBM, 2005). Bevoal Café, Voxpilot and Tellme support the XML and GSL grammar formats.

A grammar in a voice application can be in one of two modes: voice (the default mode) or DTMF (Dual Tone Multiple Frequency). DTMF can be used as an alternative to speech input, particularly when speech recognition is unreliable or problematic. In VoiceXML 2.0, DTMF is included as a value of the mode attribute in the <grammar> element. In a DTMF grammar, an automatic translation of phone buttons to DTMF tokens takes place. A DTMF grammar specifies a set of key presses that a user may use to perform an action or supply information, and for matching DTMF input, returns a corresponding semantic interpretation (W3C, 2005).

### 1.3 Spoken-Dialogue Systems

A complete spoken-dialogue system involves the integration of the following components: a speech recognition component, a language understanding component, a dialogue management component, a component for communication with an external system, a response generation component, and a speech output component (Glass, 1999) (Han, 2000) (McTear, 2002). These components work in a sequential stream, in which the first component receives the user's input, and the output from that component feeds into the next component as the input, and so forth, until the consequent voice output is synthesized for the user.

The construction of the spoken-dialog system usually consists of the following four steps (Pellom et al, 2000):

- 5) architectural design,
- 6) application design and data collection,
- 7) speech and natural language interface design, and
- 8) user feedback and evaluation.

Typically, spoken-dialogue systems can be categorized into: 1) transaction-based and 2) information-provision systems (which is called *queries-based systems* in IBM (2005)). In transaction-based systems, users can conduct transactions, such as buying or selling stocks. In information-provision systems (queries), users can obtain information on request. There are three ways of guiding users through these automated services: 1) system-driven (machine-directed), 2) mixed-initiative, and 3) user-driven (Rugelbak, et. al., 2001) (Wasinger, 2001) (Nuance, 2003) (Turunen, 2004) (Tverra, 2004) (IBM, 2005), and (Apache, 2005). In system-driven applications, the computer controls all interactions by sequentially executing each item a single time. The advantage is a reduction of the risk of errors. But the user may feel this to be too confining and controlling, which may

make them unwilling to use the application. In mixed-initiative dialogue system, the user is given a greater flexibility and allowed more natural responses. The advantage of the mixed-initiative system is that the user is more in charge. But the system has to infer information from the user's input, making it error-prone. The user-driven system is the extreme case, where the system opens the conversation with a question like "how may I help you?" The user is free to express his/ her goal, and the system faces the challenge of matching the user's responses. It has been reported that this technology is currently not considered mature for commercial applications (Rugelbak et al., 2001). Most successful applications are system-driven, directed dialogue systems.

## 2. THE CHALLENGE OF GRAMMAR DESIGN

Writing grammars is a daunting and expensive task, which forms a major bottleneck in the development of spoken language systems (Siu and Meng, 1999). Furthermore, there is no direct control that such a grammar will model the target language well when it is applied to realistic spoken queries.

Although speech-recognition technology has achieved significant progress over decades, it is not yet perfect. Speech recognition is not an exact, analytical science, but a probabilistic art and incorporates elements of sophisticated guessing (Abbott, 2001). There are still many limitations in voice applications. For example, the background noise or non-native speakers may cause poor speech-recognition performance. Also, a person, who is not familiar with the voice application, may have many out-of-grammar errors (i.e., words or sentences cannot be accepted by the current active grammar).

An effective and efficient comprehensive grammar should be able to handle a variety of user inputs. However, each user is different. It is almost impossible to design a grammar covering all possible answers to a particular question. Even for a YES/NO answer, the user may respond with “Yes”, “Yup”, “Yeah”, “Correct”, or “No”, “Nope”, “No way”. This inevitably adds difficulties in defining grammars to cover every user’s responses.

In addition, speech applications are getting broader and more sophisticated, which usually means grammars have significantly increased complexity (Nü echo, 2005). To write an effective grammar for a voice application, many factors need to be considered (see section 3.1). Writing a grammar, especially a complex grammar, is a tedious task requiring expertise. Also a grammar needs to be used and tuned iteratively, and a non-trivial grammar is very difficult to maintain.

The features of speech communication also implies difficulties in dialog design as follows:

- (1) Speech is transient and invisible, and human beings have limited short-term memory (Bouzid, 2006e). Psychologists have found that, in general, people can only memorize five to nine chunks of information at a time (Apache, 2005). People may quickly forget what they have just heard, which is different from traditional web pages where the information is always present and visible.
- (2) The conversation is linear. The communication is slower than visiting graphic web pages.
- (3) Users may not know the navigation words, and may not know how to respond to a prompt from the voice system.
- (4) Users may get lost, because of their short-term memory to know where they are in a conversation.



### **3. ISSUES IN GRAMMAR DESIGN**

#### **3.1 Guidelines for Grammar Design**

Designing good grammars is as much art as science (IBM, 2005). A well-defined grammar can not only improve speech recognition accuracy, but also provide the user with great flexibility and comfort in voice services. Good grammars are essential for the usability of a speech application. Shi (2003a) provided a survey on the techniques of using natural language features to improve speech recognition accuracy, such as constraining the grammar by integrating semantics, using probabilities (weights), combining stochastic and grammar-based techniques, large-vocabulary related techniques, and SpeechWeb techniques. From first-hand experience in writing grammars for real-world voice applications, many guidelines in VoiceXML application grammar design from the developer's point of view have been created. They are summarized and presented as follows.

Grammars for VoiceXML applications can be defined in an external file or inline. The developer can weigh such factors as follows (IBM, 2005):

- (1) Grammar size and its effect on speech recognition accuracy and document access time. Generally, the smaller grammar has the better recognition accuracy (Wasinger, 2001) (Shi, 2003b) (Mané and Levin, 2005) (IBM, 2005) and shorter access time (Nuance, 2003).
- (2) The importance of instantaneous response and the corresponding need to load the grammars up front. For example, if the grammar is large and in a menu or form that is unlikely to be executed, it can be defined as an external file. Conversely, if it needs to be instantaneously ready, it can be defined inline, rather than having to be downloaded from the web application server, when the user accesses the menu or form.

Abbott (2001) identifies the following techniques to improve speech recognition performance:

- (1) Use short phrases (Boyce, 2000) or multi-syllabic words for links (e.g., “start over”).
- (2) Reserve the shortest, commonest responses for field-level responses, which will be matched with high priority by the speech recognizer. The links with broader scope should be longer phrases that can be recognized in a variety of contexts.
- (3) Allow the use of DTMF where precise input of numbers is important or the system has difficulty in recognizing the user’s input (Apache, 2005). Limit spoken digits to 4 or less (Eisenzopf, 2006).
- (4) Do not share recognition errors with the user. For example, instead of “Did you say Austin or Boston”, the computer should respond with the prompt like: “I did not get that. What city”, instead of repeating the question.
- (5) Do not make the grammars too broad, or include too many synonyms.

It was reported that the complexity of a grammar greatly affects the speed and accuracy of the recognizer (Nuance, 2003) (IBM, 2005). The grammar designer should predict how users will respond. However, it is impossible to include all of the responses that can occur in the application because one cannot control how people speak. In practice, one can guess the most common ways that people will respond and include them in the grammar, instead of trying to include every alternative. After collecting some data, one can refine the grammar, collect more data, and refine the grammar further, and so on. Therefore, grammar writing is actually an iterative process.

In addition, Nuance (2003) observes that there are two types of responses from the user: 1) the information item by itself, and 2) the literal response to the questioning wording. For example, if the system asks “what is your departure city?” most responses will be just a city name like “Toronto”. A smaller group of

responses may be “My departure city is Toronto.” These observations indicate that prompts need to be worded carefully, and the grammars and the prompts should correspond closely with each other.

Nuance (2003) identifies the following guidelines for building a robust application.

- (1) Specific prompts will lead to a high recognition accuracy and robustness.
- (2) Smaller sub-grammars may result in a more robust system (Wasinger, 2001).
- (3) Cover as many words as possible in a sentence since the robust Natural Language engine ranks interpretations according to the number of words of a sentence.
- (4) Use as few grammar rules as possible in the application.
- (5) Use grammar weights/ probabilities to maximize the probability of the phrase fragments.

Nuance (2003) indicates that a 5% out-of-grammar rate is acceptable, even 10-20% out-of-grammar rates are not uncommon for certain types of recognition tasks. If the out-of-grammar input is a problem for a voice application, a Statistical Language Model (SLM) could be considered. A detailed discussion of SLM can be found at (W3C, 2001).

Jackson (2004) points out that:

- (1) A good grammar should cover effectively the range of responses that can be encountered in the application (IEEE, 2002). This can include the essential input as well as extraneous words and phrases.
- (2) A grammar that is too large will hinder speech processing and potentially lead to more mis-recognitions (Abbott, 2001) (IEEE, 2002) (IBM, 2005).
- (3) Grammars should not overlap (i.e., pay attention to scope);
- (4) Excessive use of global grammars (defined in the root document) can increase the possibility of overlapping.

In general, recognition accuracy can be improved by the following measures (IBM, 2005):

- (1) Simplifying the grammars to minimize the possibility of confusion between words (Abbott, 2001) (IEEE, 2002) (Jackson, 2004).
- (2) Presenting fewer choices (Apache, 2005).
- (3) Having fewer active grammars.
- (4) Ensuring that the grammar can accept user responses that mirror key phrases from preceding prompt. For example, if the system is asking “Are you a student or a teacher?” the grammar should be able to accept the phrase such as “a student”, “I am a student”, “teacher”, and “I am a teacher”.

Apache (2005) provides the following suggestions for grammar design:

- (1) Use form-level, mixed-initiative grammars whenever possible. People feel more comfortable if they think they are in control of the system, not the other way. One technique could be barge-in. Eisenzopf (2006) suggests that if natural dialogs fail, fall back to directed prompts.
- (2) Take advantage of grammars allowing global commands.
- (3) If the grammars cannot be determined at the time the VoiceXML document is written, dynamic grammars should be used.
- (4) Use the user’s terminology in the grammars, instead of the developer’s jargon.
- (5) Allow the user to phrase their input in multiple ways to increase the flexibility of the interface.
- (6) It’s important to include non-verbal vocalizations in grammars, such as “err” and “um”, which are common in human to human communication.
- (7) Avoid including words that have different meanings but similar pronunciations in the same grammar. Try to use only phonetically distinct words (Nuance, 2003).

To decide what words and phrases to be included in grammars, and when to make each grammar active, IBM (2005) mentions the following trade-offs which need to be taken into account:

- (1) *The length of words and phrases.* The longer the words and phrases, the better recognition accuracy, because of greater differentiation among valid utterances. However, the longer words and phrases could make the dialogs slower. Also, it is more difficult for user to remember longer phrases. On the other hand, shorter words and phrases increase the chances to be mis-recognized. Monosyllabic words and short words with unstressed vowels are especially prone to be recognized as each other, even though they may look and sound different to a human ear. Therefore, if a grammar has to include many short user utterances, it is important to minimize acoustic confusability by making them as acoustically distinct as possible. The advantages of using shorter words and phrases include faster dialogs progress and easy to remember for users.
- (2) *Vocabulary robustness and grammar complexity.* A robust grammar with great complexity may include many synonyms and alternative phrases to offer users with greater freedom of word choice. Consequently, users may assume that they can say anything, which would lead to large number of out-of-grammar errors. Also, the complex grammar files are larger and need longer time to load. A simple grammar with less robustness may constrain the users more with narrow lists of valid utterances. Also such grammar files are smaller, they can be loaded more quickly. Simple grammars usually have better recognition accuracy (Wasinger, 2001) (Mané and Levin, 2005).
- (3) *Number of active grammars.* If you activate more grammars at the same time, you are improving the usability of the application, such as by allowing anytime access to items on main menu. Meanwhile, you are increasing the chances of recognition conflicts, and the performance is degrading. The fewer active grammars may constrain the user more, but

provide better performance with less mis-recognitions due to recognition conflicts.

A special case which we need to discuss further in speech recognition, is numbers and letters (alphanumeric strings), which are used very widely in a variety of applications. Recognizing alphanumeric strings is one of the most challenging aspects of speech recognition because they are short and many of them sound very similar, even for human listeners (IBM, 2005). For example, “six” (the shortest spoken digit in English) is commonly inserted (recognized but not spoken) and falsely deleted (spoken but not recognized) by speech recognizers (Abbott, 2001). Also many of the letters are easily confused with other letters: “N” with “M”, “B” with “D” etc. (Rahmel, 2005). Furthermore, each letter in a string presents a new chance for error. Rahmel (2005) presents a formula to calculate the accuracy of a string taking into account the accuracy of each single character of the string and the length of the string:

$$\text{string accuracy} = (\text{accuracy of a single character recognition})^{\text{length of the string}}$$

To overcome the difficulty of recognizing alphanumeric strings, Abbott (2001) and Apache (2005) suggest a possible solution of allowing DTMF input for numbers and International Communications alphabet for letters (e.g., `alpha`, `bravo`, `charlie`, and `delta` represent `a`, `b`, `c`, and `d`).

In most cases, there is a pattern to the alphanumeric string. Rahmel (2005) provides the simplest and best way to solve the alphanumeric problem by explicitly spelling out each individual string as a separate phrase element. Since it's easy to get one character out of ten wrong, and it's more unlikely to get two or more characters wrong in just the right way so that they turn one valid string into another valid string. Rahmel (2005) states that this approach should work well for static lists up to 100K entries in size.

Furthermore, rather than constrain the individual characters, Rahmel (2005) identify groups of characters that occur together. Typically, the chance is low to recognize the multiple characters all in the right way, if they are all wrong. This approach could be combined with subsequently validating the top N n-best choices against a database containing the valid alphanumeric strings (Rahmel, 2005).

Rahmel (2005) talks about some tips for writing alphanumeric grammars:

- (1) Force the recognizer to use the specially trained letter recognition models by specifying the letter in the grammar as a letter followed by a dot to distinguish letters (e.g. A.) from words (e.g. “A person”).
- (2) Write numbers as words, i.e., use “one” instead of “1” and so on, so that the recognizer does not have to use text normalization to translate the digits into words.
- (3) Separate characters to avoid mis-recognizing the letter string as a word.
- (4) Clearly prompt the user (Eisenzopf, 2006).

### **3.2 Dialog Design**

Dialogs are the main components of a voice application. Recognition performance will be reduced if the speaker is unsure what to say in dialogs (Nuance, 2003). Each dialog has one or more speech and/or DTMF grammars associated with it. Dialogs determine the grammars. Therefore, it is very important for the designer to understand the dialogs well before writing the grammars. Due to the specific features that a conversation has, there exist more challenges for designing an effective dialog for a voice application than designing a traditional web page. Many efforts have been made to come up with good guidelines for dialog design.

There are two kinds of dialogs: forms and menus. *Forms* define an interaction that collects values for a set of form item variables. A *menu* presents the user with a choice of options and then transitions to another dialog based on that choice. Nuance (2003) suggests that, to design dialogs, one should be clear about the information required to complete, the information will be requested, and the type of the system (i.e., directed dialog or mixed-initiative dialog). This information will help determine the shape and content of the grammars for the application. Mané and Levin (2005) assert that a good dialog design relies on an understanding and consideration of the business demands, the technology constraints, and the user needs.

Tverra (2004) summarizes the principles in dialog design as follows:

- (1) Minimize the cognitive load for the users (Apache, 2005). The fact is that human beings can only remember short information in dialogues. Therefore, it is advisable to keep menu choices and information short.
- (2) Balance efficiency and clarity. Though short information helps user memorization, the prompts also need to be as clear as possible (Eisenzopf, 2006).
- (3) Ensure high accuracy (Eisenzopf, 2006). This means that the user should be able to obtain help any time. For example, tapered prompting could be a choice (Bouzid, 2006f).
- (4) Avoid using “I”. The user should keep in mind that s/he is communicating with a non-person, which means that s/he must comply with the rules of the system. However, this is not the view of Eisenzopf (2006) and Bouzid (2006b) who suggest the use of anthropomorphism (but only in natural dialogs) to construct a more naturally verbal conversation. Eisenzopf (2006) states that an AT&T study shows that callers are more satisfied with applications that used first person in conversations even though callers know that it is a computer. This is an arguable assertion. The decision of whether to use the first person is up to the developer.



- (5) Recover from errors gracefully (Angel Voice Site). Errors and mis-recognitions are unavoidable in voice applications. It is the best if the user does not notice the error, while the error occurs and recovers. Otherwise, the system needs to recover errors quickly and can not let the user feel it is his/ her fault.

IBM (2005) identifies the following issues when deciding how to group dialogs:

- (1) Logical grouping of menus or forms.
- (2) Resources they require.
- (3) Functions they perform.
- (4) Expected frequency of use.
- (5) Number of pages you want the VoiceXML browser to request from the Web application server. For example, a form or menu that is infrequently used and contains a large grammar or references, large grammar, or audio file, could be defined in a separate VoiceXML document, so that the large files are downloaded only when needed.

Apache (2005) mentions that a person usually can hold five to nine chunks of information in memory, therefore there should be no more than five options in a menu for choice. The available commands should be listed after the function description. Always put frequently-used items first in the menu, and let the user know the end of the menu if possible (Biber and Kozminski, 2005). In addition, to satisfy the user, the conversation should be designed to be as short as possible. One opinion is that, instead of counting the turns in the dialogue, the number of confirmations that were rejected is a more important factor when determining the user's level of content (Mané and Levin, 2005).

Menu is an important element in dialog design. Bouzid (2006g) provides the following suggestions for voice menu design.

- (1) Avoid the For ... Say or To ... Say Construct. For example, instead of using “to find out your balance, say BALANCE”, you can say “balance”.
- (2) Use landmarks for navigational feedback. For example, you can say “balance”, or “operator”.
- (3) If the user is an expert, let him go first.
- (4) Present menu choices when the caller doesn't speak or when what was said is out of context.

### 3.3 Prompt Design

*Prompts* are the short audio files that are played to the user (Biber and Kozminski, 2005). Prompts indicate that it's time for user's input. They provide important navigation clues within the spoken dialogues. The system is prompting the user for some information and waiting for user's input at that specific point in the application. Prompts can be prerecorded or dynamically generated by using TTS (Text-To-Speech) technology. Effective prompts can reduce the recognition errors, increase user satisfaction, and enhance system productivity (Apache, 2005).

Prompts should be defined before writing grammars because prompt wording can greatly affect the wording of the user's response, which needs to be captured by the grammar. The prompts for directed dialogs are specific, such as “what is your departure city?” The prompts for the mixed-initiative applications are open, such as “Where would you like to travel?” The open prompts add difficulties to grammar design, but are closer to human interactions. The specific prompts may provide the user with a robust system with high recognition accuracy (Yankelovich, 1997) (Angel Voice Site). Eisenzopf (2006), Bouzid (2006d), and Yankelovich (1997) suggest not using open prompts. Clear and unambiguous wording of a prompt is a key contributor to application success (Biber and Kozminski, 2005).

Apache (2005) discusses several techniques on prompting:

- (1) *Tapered prompts* mean that the system starts with a detailed prompt and tapers it to the prompt for the missing information or many pieces of information. It is a good choice if some information needs to repeat again and again (Yankelovich, 1997) (Eisenzopf, 2006) (Bouزيد, 2006d) (Bouزيد, 2006f). Typically, it is used in a mixed-initiative dialog. Hone and Baber (1995) point that the longer, more constraining prompts may result in more appropriate user response and less need for re-prompting. However, they increase the total transaction time.
- (2) Opposite to tapered prompts are *incremental prompts* (Marcus et al, 1996) (Biber and Kozminski, 2005) (Eisenzopf, 2006) (Bouزيد, 2006d) (Bouزيد, 2006f). Sometimes, the system provides the short prompt information first, then a more detailed prompt if required, and so on. In this way, the experts can move fast, and the novices can also get the required information.
- (3) *Leading prompts* are used to narrow the user's responses to a question, and specify the question for a specific answer (Biber and Kozminski, 2005). The designer can include important words, especially the words that are expected to be answered, at the end of the prompt. For example, the prompt could be "The current price is 45 dollars per share. Would you like to buy, sell, or quit?"
- (4) People will feel comfortable if they know they are understood properly. In many cases, the system needs to be sure it is proceeding correctly based on the user's purpose. Especially, when the next action could result in irrevocable consequences, confirmation and feedback are needed to assure the user that the communication is proceeding correctly (Yankelovich, 1997) (Biber and Kozminski, 2005). Eisenzopf (2006) suggests always confirming the recognition. Explicit confirmations are necessary for the actions that can cause severe and permanent results. However, too many unnecessary confirmations make the user interface too verbose and annoying the user, which is actually impacting on

- system's usability. An implicit confirmation can be used to inform the user about the results of a particular task and avoid confusion without the extra cost of asking for the user's consent.
- (5) The prompts need to be brief and deliver only the necessary information to the user due to people's limited memory (Biber and Kozminski, 2005).
  - (6) Design polite prompts (Boyce, 2000). No matter what the reason is causing the errors, never blame the user! Always let the system take the blame. Never make the user feel it is his/ her fault.
  - (7) Use of barge-in (Biber and Kozminski, 2005) (Bouzid, 2006a). Usually, an experienced user would like to speed up the communication by providing information quickly. *Barge-in* means that the user can interrupt prompts to input information, rather than waiting for the prompt to complete. This technique can make the system more productive. However, Boyce (2000) suggest not using barge-in unless it is sure that the majority users are frequent users. The final decision is up to the designer according to the specific application.

Apache (2005) has some additional suggestions as follows:

- (1) People have trouble in remembering synthesized speech for long and complex message. Therefore, prompts recorded in human speech should be used as much as possible.
- (2) The terminology in the prompts should be understood by the potential users.
- (3) Avoid compound questions and questions allowing multiple answers (Biber and Kozminski, 2005), because they are too verbose and confusing for users.
- (4) Use tones to let the user know it's their turn for input (Biber and Kozminski, 2005). Biber and Kozminski (2005) also mentioned a preceding set of instructions should be included.
- (5) Keeping the interface simple is more important than trying to offer all

things to all people.

The following tips in prompt design are summarized by Bouzid (2006b):

- (1) Drop the "Welcome to..." and "Thank you for calling..." to shorten the length of prompts.
- (2) Use an audio icon.
- (3) Drop the "For English..." and "You can interrupt me at any time".
- (4) Do not mention the web site upfront.
- (5) Establish that they can use speech.
- (6) Postpone the call-recording disclaimer because such disclaimers not only lengthen the opening prompt but may frustrate the user as a cue that the call is going to be transferred to a live agent.

Bouzid (2006d) suggests the following tips:

- (1) Ensure that all of the behavior avoids endless loops.
- (2) Do not mix voice and text to speech.
- (3) Do not put into your prompt something that your grammar can't handle.
- (4) Do not switch modes on the caller between tone and voice.
- (5) Do not go quiet for more than 3 seconds.
- (6) Instruct the user saying longer phrase instead of the hard-to-recognize short words. For example, direct the user say "help me" instead of "help".

### **3.4 Use of Sub-Dialog**

A *sub-dialog* is a mechanism for decomposing complex sequences of dialogs to better structure them, or to create reusable components (W3C, 2005). A sub-dialog is also a VoiceXML document, like a function call. Using sub-dialogs allows documents to call each other and exchange data, without using CGI or other server-side mechanisms (VOXEO, 2006). A sub-dialog provides a mechanism for invoking a new interaction, and returning to the original form.

The usages of sub-dialogs include the following (W3C, 2005):

- (1) Creating a confirmation sequence that may require a database query.
- (2) Creating a set of components that may be shared among documents in a single application.
- (3) Creating a reusable library of dialogs shared among many applications.

The advantages of using sub-dialogs are as follows (VOXEO, 2006):

- (1) Sub-dialogs are easier to maintain and faster to load and execute than the large document.
- (2) Using sub-dialogs is helpful for clean code.
- (3) Using sub-dialogs can eliminate redundant code.
- (4) Using sub-dialogs makes some common voice recognition dialogs reusable.
- (5) The use of sub-dialogs results in a much leaner, more modular code architecture.

### **3.5 Use of Sub-Grammar**

A grammar is either a top-level grammar or a sub-grammar. Top-level grammars are the only ones that can be referenced by an application at runtime. All the other grammars are sub-grammars that can be reference by only other grammars. However, the distinction between top-level grammars and sub-grammars does not apply to grammars used dynamically, including just-in-time, VoiceXML, and Speech Objects grammars.

Flat-file grammars are usually adequate for simple voice application, however, multi-level complex grammars are more powerful and flexible (VOXEO, 2006). The advantages of using sub-grammars include the following (Nuance, 2003):

- (1) Sub-grammars are reusable by multiple grammars or applications.
- (2) The use of sub-grammars simplifies grammar creation and revision.
- (3) Using sub-grammars helps focus the grammar development to the task at hand.
- (4) Defining sub-grammars hides unnecessary details and promotes modularity.
- (5) Sub-grammars can eliminate redundant code.
- (6) A hierarchy of grammars using sub-grammars can improve the robustness of speech application (VOXEO, 2006).

### 3.6 Grammar Weights and Probabilities

A *weight* is a multiplying factor assigned to the rule to influence the likelihood of a phrase in the grammar (Nuance, 2003) (W3C, 2004). A weight is a non-negative floating point value without exponential. Optionally, a weight can be assigned to any alternative in an alternative expansion. The items with higher weights are favoured more over others by speech recognition engine, if the acoustic processing results in similar scores.

The phrases that are expected to be spoken more frequently (more common) should be assigned higher weights, and the less likely to be spoken utterances are less likely to be matched with a lower weight by the recognizer. Therefore, the speech-recognition accuracy is improved. On the other hand, if the user's input does not match the rule with higher weight, the rules with lower weight are searched until matched or the search reaches the end of the grammar. In this sense, the weighted grammar is robust. If the summation of the weights of all the alternatives of a grammar rule is 1.0, these weights are considered *probabilities*. Probabilities are useful to reflect the frequency of items in a construct (Nuance, 2003).

The appropriate uses of weights/ probabilities can improve the recognition accuracy, robustness and speed (Nuance, 2003) (VOXEO, 2006). However, the bad assignments of weights/ probabilities can actually hurt the recognition performance. The weight/ probability assignment should be based on at least ten samples (on average) for each list element (Nuance, 2003) (Eisenzopf, 2006).

It is valuable to note that the default value for a non-labeled rule is 1.0 in a weighted grammar. A possible unintended result might be that the rule without weights may have stronger likelihood than the rule with weights (less than 1). Therefore, it is important to be consistent in the usage of weights and probabilities throughout all the grammars in the application.

W3C (2004) identifies the following limitations on the definition and application of weights:

- (1) No normative or informative algorithms can be used to assign weights.  
The usage of weights is under the internal control of the recognizer.
- (2) It is quite difficult to determine the appropriate weights for any specific grammar and recognizer. However, the assigned weights by guessing do not guarantee improvement of speech recognition performance.
- (3) Studying real speech input to a grammar is the best way to obtain the effective weights. A reasonable example for developing weights is to use weights that are correlated with the occurrence counts of a set of alternatives.
- (4) The appropriate weights for a particular recognizer do not mean the improvement of recognition performance on other speech recognizer.



### 3.7 Error Handling

Errors are inevitable in voice applications given current speech technologies. If the errors cannot be handled properly, the user will be frustrated and even refuse to use the voice application. The prevention and proper handling of errors are crucial to a successful speech application.

Apache (2005) discusses the following typical kinds of errors:

- (1) The user's input does not match the grammar.
- (2) Background noise causes the recognition failure.
- (3) The user's pronunciation (e.g. accent) may be the reason that the system fails to recognize it.
- (4) The user starts to speak too early, or too late, or not at all.

The goal of error prevention is to avoid putting the user in the situations that are error-prone. Apache (2005) suggests the following considerations.

- (1) Do not overload the user's memory.
- (2) Allow DTMF for digit string input, especially if the system has already failed to recognize this item.
- (3) Use comprehensive grammars to overcome the grammar mismatch errors.
- (4) Minimize background noise.
- (5) Well inform the user for help.

VoiceXML has a built-in mechanism for handling nomatch and noinput errors. Since an error can occur anywhere in a dialog, it is important to catch and handle both nomatch and noinput errors for each field (Bouzid, 2006f). The number of nomatch and noinput occurrences can be specified, so the system can mention the DTMF input or directly be transferred to a human operator after the specified number (Yankelovich, 1997).

IBM (2005) mentions several strategies toward error recovery and confirming user input, under different situations.

- (1) If the user input is invalid, the system can state the problem and re-prompt. Also, the user can be directed to keypad input.
- (2) If the recognition error occurs while the user is making choices along a menu path or completing items in a form, one can feed the recognized input forward into the next prompt. The “Go Back” command should be included in the first level help (Bouزيد, 2006c).
- (3) If the user distracts from the communication and does not hear all the information presented, the always active command “Repeat” can solve this problem.

Biber and Kozminski (2005) provide two approaches to handle recognition errors.

- (1) Use prompt escalation, which means that the prompts change every time the application queries the user for the same data. If all fail, the user might switch to Touch-Tone input. This not only avoids user frustration, but also increases the number of completed interactions using the automated system.
- (2) When the speech-recognition engine recognizes an input or utterance, it returns a value between 0 and 100 to indicate how confident it is of the match. The two confidence thresholds (rejection and confirmation) should be set (Everett et al, 1993) (Eisenzopf, 2006). An utterance with a confidence, below the “rejection threshold”, is rejected as a not-recognized utterance; above the “confidence threshold”, does not require confirmation; between the two thresholds, requires confirmation.

### 3.8 Summary

This section has discussed the issues related to grammar design and some proven guidelines for designing high-quality grammars from the developer's point of view. The topics covered include dialog design, prompt design, sub-grammar design, sub-dialog design, grammar weights and probabilities, and error handling. We summarize this section in the following.

Principles for grammar design are summarized as follows:

- (1) The complexity of a grammar greatly affects the speed and accuracy of the recognizer.
- (2) A smaller grammar may result in better speech recognition accuracy and fast access. A larger grammar will hinder speech processing and potentially lead to more mis-recognitions, also need more time to load.
- (3) Use short phrases or multi-syllabic words for links, and reserve the shortest, commonest responses for field-level response.
- (4) Specific prompts will lead to a high recognition accuracy and robustness. Presenting fewer choices in menus.
- (5) Simplify the grammar, do not make the grammars too broad, or include too many synonyms. Use as few as possible grammar rules in the application. Have as few as possible grammars active concurrently.
- (6) Ensure the grammar can accept the user responses that mirror key phrases from preceding prompt. Allow the user to phrase their input in multiple ways to increase the flexibility of the interface.
- (7) Include the most common ways that people will respond, instead of trying to include every alternative. Use the user's terminology in the grammars, instead of the developer's jargons.
- (8) include non-verbal vocalizations in grammars, such as "err" and "um"
- (9) Try to use only phonetically distinct words. Avoid including words that have different meanings but similar pronunciations in the same grammar.

- (10) Use grammar weights / probabilities to maximize the probability of the phrase fragments.

In addition, there are some trade-offs need to be taken into account in grammar design, such as the length of words and phrases, vocabulary robustness and grammar complexity, and number of active grammars. Furthermore, the special case for alphanumeric strings has been put forward and some possible solutions have been discussed.

Main guidelines for dialog design are listed as follows:

- (1) Due to the short memory of human beings, there should be no more than five options in a menu for choice.
- (2) Always put the frequently-used items first in the menu, and notify the user of the end if possible.
- (3) The conversation should be designed as short as possible
- (4) Present menu choices when the caller doesn't speak or when what was said is out of context.
- (5) Users should be able to obtain help any time.
- (6) Recover from errors gracefully.
- (7) Balance efficiency and clarity.

Since the wording of prompts greatly affects the wording of grammars, it should be defined before writing the grammars. The techniques to design prompts include tapered prompts, incremental prompts, leading prompts, barge-in, and confirmation and feedbacks. Also, some suggestions have been made such as, using recorded prompts, avoiding compound questions and questions allowing multiple answers, using tones to inform user's turn, and remaining simple interface not trying to offer all things to users.

The use of sub-grammars has the following advantages:

- (1) Sub-grammars are reusable and promote modularity. They hide unnecessary details and eliminate redundant code.
- (2) The use of sub-grammars simplifies grammar creation and revision.
- (3) Using sub-grammars helps focus the grammar development to the task at hand.
- (4) A hierarchy of grammars using sub-grammars can improve the robustness of speech application.

The advantages of using sub-dialogs include the following:

- (1) Sub-dialogs are easier to maintain and faster to load and execute than the large document.
- (2) Using sub-dialogs can eliminate redundant code, and is helpful for clean code.
- (3) Using sub-dialogs makes some common voice recognition dialogs reusable, and results in much leaner, more modular code architecture.

Using weights/ probabilities properly can improve recognition accuracy, robustness and speed. However, improper assignment of weights/ probabilities can actually hurt recognition performance.

We discuss the difficulties while defining and applying weights as follows:

- (1) No normative or informative algorithms can be used to assign weights.
- (2) It is quite difficult to determine the appropriate weights for any specific grammar and recognizer.
- (3) Studying real speech input to a grammar is the best way to obtain the effective weights.
- (4) The appropriate weights for a particular recognizer do not mean the improvement of recognition performance on other speech recognizer.

Errors are inevitable in voice applications. The prevention and proper handling of errors are crucial to a successful speech application. Some suggestions are as follows:

- (1) Do not overload the user's memory.
- (2) Allow DTMF for digit string input.
- (3) Use comprehensive grammars to overcome the grammar mismatch errors.
- (4) Minimize background noise.
- (5) Well inform the user for help.

The following are some strategies toward error recovery and confirming user input.

- (1) If the user input is invalid, the system can state the problem and re-prompt.
- (2) Use "Go Back" or "Repeat" commands.
- (3) Use prompt escalation.
- (4) Set confidence thresholds for rejection and confirmation.

## 4. Voice User Interface (VUI) Design

Voice User Interface (VUI) usability is the key to the success of a VoiceXML application. A good VUI design is not an option, but a requirement. A poor VUI does not only frustrate users, but is also insulating and provocative. A good VUI has a natural and human-like quality. Many technical limitations can be compensated with properly designed speech interface (Turunen, 2004) (Everett et al, 1993).

Commonly, people take it for granted that the usability of a speech application will increase with the improved ASR (Automatic Speech Recognition) performance. However, the usability of a speech application is determined by a tight and highly complex interplay between the ASR and the components of the VUI design (Peissner, 2002). Also, it is critical to strike the right balance between the simplicity of touch-tone Interactive Voice Response (IVR) systems and the complexity of AI-like speech applications in Voice User Interface (VUI) design (Gorelov, 2005).

The main reason of the difficulty in VUI design relies on the fact that speech has a temporary existence, and the users must remember what they have heard. One VUI design objective is to avoid making users hear more (or less) than they need to hear or to say (IBM, 2005). Also, it is important to make the user feel that they are moving forward with every interaction (Yankelovich, 1997). Dialog design and speech-recognition accuracy are the main factors that affect VUI usability. Both issues must be addressed to provide an application that people want to use. This requires iterations of usability testing and fine-tuning of the UVI.

#### 4.1 Issues Related to VUI

IBM (2005) points out the main concerns in designing an effective VUI as follows:

- (1) What to present.
- (2) How much to present.
- (3) How to present it.
- (4) When to present it.

To design an effective VUI, one should understand customer profiles, meet realistic expectations, and follow a design methodology that uses proven techniques. The details of VUI design methodology can be found at (IBM, 2005).

Abbott (2001) provides the following VUI design principles and techniques:

- (1) Keep it simple, and do it well. Do not compare the capacity of the VUI with that of GUI. Use the 80/20 rule. It means that, aim to simply and effectively handle the easiest 80% of the load, and leave the other 20% to other means (such as human operators) (Eisenzopf, 2006).
- (2) Accommodate Errors. Since errors are unavoidable, the VUI design should not try to eliminate errors, but rather to contain them and tolerate them. A good VUI is actually deceptively simple. This means that, the basic structure of dialogs is simple and easy, but it should be able to handle a multitude of errors. In developing a VUI, the minority of effort should be spent on the basic dialogs. The majority of effort should be spent on detecting errors, recovering from them, and getting the conversation back on track.
- (3) Design for everyone, everywhere. Each user is an independent individual. There are a wide variety of voices, speech skills, and vocabularies among users. Therefore, the response vocabulary should be simple and generic. The ideally VUI design can handle all kinds of voices in all kinds of environments.



Also, Abbott (2001) mentions the following issues in VUI design:

- (1) *Modeling*. In conversations, people tend to model their speech on the other party's. Modeling is useful for directing users to acceptable forms of speech. When use modeling in VUI design, use the prompts that are brief and to the point (Boyce, 2000). If providing help, it's more important to provide examples first than explain what's going on (Apache, 2005). Do not use long, wordy prompts (Bouzid, 2006d).
- (2) *Disfluency* is one of the biggest problems for continuous speech recognition. The longer utterance, the more disfluencies. Therefore, the application design should limit the length of utterances. However, people like to make long utterances while they are familiar with the system. To design a VUI to minimize the effects of disfluency, the designer should use mixed-initiative combined dialogs with directed forms. In addition, it's not wise to address disfluency through grammar design, because this will increase the grammar complexity and slow down recognition, with few chances of ultimate success.
- (3) *Synthesized speech*. More concentration and effort is required for people to listen to the synthesized speech than to listen to human speech. Therefore, try to use recorded prompts as much as possible. Pay attention to prosodic features when using synthesized speech. Synthesized speech is not appropriate to read long lists to the user.
- (4) *Turn-taking and error amplification*. A good VUI will make the user feel oriented, in control, and be able to anticipate what will happen next. It's very common for the two parties to lose track of whose turn it is in the human-computer interaction. A major goal of VUI design is to construct the interface to direct the user to a safe point where s/he is oriented, in control, and knows what's coming if something goes astray.
- (5) *Lost in space*. Since speech is transient, invisible, and asymmetric (Bouzid, 2006e), it is easy for people to feel "lost in space" and do not know "where they are" in a conversation. A good technique for maintaining orientation is

to provide auditory cues along the way, for example, use different voices for different parts of the application. However, do not use too many tunes, tones, or other non-speech audios, which make the user tired to listen repeatedly. Barge-in technique can enable experienced users to move fast and the inexperienced user get contextual feedback. Furthermore, the orientation tips in prompts are useful when the person is silent or cannot be understood. Do not force a lot of contextual information on the user unless s/he requests it. Make sure to incorporate error-handling to avoid ran-away error amplification.

- (6) *The wide range of users and environment* is a big challenge for VUI design. The following tips need to be considered to accommodate different experience levels and environments.
- (a) Shortcut should be available for expert users.
  - (b) Use mixed-initiative dialogs backed up with directed prompting for filling out forms.
  - (c) Incorporate yes/no exchanges as the fallback when more complicated dialogs are not working.
  - (d) Do not clutter up basic prompts with a lot of tutorial material aimed at expert users.
  - (e) If a user encounters a lot of errors, do not assume it means s/he is “slow”, it may be because an expert user is in a tough environment.

Apache (2005) and Sun (1998a) suggests the following tips to build user-friendly interface:

- (1) Use recorded audio for all prompt messages (Abbott, 2001) (Eisenzopf, 2006) (Bouzid, 2006d). Meanwhile, text is included for TTS as a backup in case the audio file is not available.
- (2) <help>, <noinput>, and <nomatch> event handlers are used widely to make sure the users are always guided through the dialogs.
- (3) <reprompt> comes with prompt counts to make messages more detailed

if the user gets stuck on one field.

- (4) Involve real-world users, not trained engineers, to thoroughly test all the dialogs.
- (5) Provide adequate customer support. Users should be able to contact a real person easily when they have problems with the computers.

Also, Apache (2005) suggests the *consistent* interface. Terminology consistency means the use of the same words, rather than synonyms to refer to an object or event. To convey a consistent personality to the user, use the same wording, attitude, and style in all dialogs in the application. Use the same key to the same word or action, while using DTMF.

## 4.2 Summary

A good VUI, with a natural and human-like quality, is crucial to the success of a VoiceXML application. A good VUI design is not an option, but a requirement. Many technical limitations can be compensated with properly designed speech interface.

Some principles and techniques for VUI design are summarized as follows:

- (1) Keep it simple, do it well.
- (2) Accommodate Errors.
- (3) Design for everyone, everywhere.
- (4) Use recorded audio for all prompt messages.
- (5) Use <help>, <noinput>, and <nomatch> event handlers to make sure the users are always guided through the dialogs.
- (6) Involve real-world users, not trained engineers, to thoroughly test all the dialogs.
- (7) Provide adequate customer support. The users should be able to contact a real person easily when they have problems with the computers.
- (8) Keep the consistent interface.

In addition, there are some issues related to VUI design that need to be considered.

- (1) *Modeling* is useful for directing users in speech applications.
- (2) *Disfluency* is a big challenge for continuous speech recognition. Limit the length of utterances in VUI design. Use mixed-initiative combined dialogs with directed forms to minimize the effects of disfluency.
- (3) Use recorded prompts as much as possible instead of synthesized speech.
- (4) A major goal of VUI design is to construct the interface to direct the user to a safe point where s/he is oriented, in control, and knows what's coming if something goes astray.
- (5) *Lost in space*. Provide auditory cues along the way to help users from feeling "lost in space".
- (6) *The wide range of users and environment* is a big challenge for VUI design.

## 5. TESTING

Because of the complexity and ambiguity, extensive testing and tuning are indispensable for speech-enabled applications. Tuning is an iterative process of analyzing system performance based on system logs and recorded user interactions, then applying the best design practices to achieve the most satisfying customer experience and to work around technology imperfections (Biber and Kozminski, 2005) (Eisenzopf, 2006). The tuning process should be based on actual user data, so that one can examine what users have really said to the system and update the grammars and dialogs accordingly.

Tuning is a complex task which can take a long time (sometimes, several months) and involve an interdisciplinary team of professionals, such as developers, testers, linguists, and psychologists.

### 5.1 Testing Issues

There are a few issues involved in grammar testing, such as:

- (1) Coverage test. One should test words and phrases that are in the grammar to verify that the grammar has the ability to parse a prescribed set of utterances (Nuance, 2003) (IBM, 2005) (Biber and Kozminski, 2005).
- (2) Over-generation test. It is to test the words and phrase that are out of grammars to make sure the grammars will not accept the unwanted sentences (Nuance, 2003)(IBM, 2005).
- (3) Interpretation test. It verifies that the grammar delivers expected natural language interpretation for a prescribed collection of phrases (Nuance, 2003).
- (4) Ambiguity test. It exposes phrases parsed by the grammar that have multiple interpretations (Nuance, 2003).
- (5) Pronunciation test is to detect words with unknown pronunciations and misspellings in the grammars (Nuance, 2003).

- (6) Regression test (Nuance, 2003). Whenever a grammar changes, it needs to be thoroughly tested to ensure that no errors have been introduced. The results from the new version will be compared with that of the old version.
- (7) Identify the consistent mis-recognitions (IBM, 2005). The grammar will be tested with a group of test subjects that are representatives of the demographics and environments of the users. One can vary the ambient noise level, gender, age, accent, and level of fluency during desktop testing. If a consistent mis-recognition is found, the developer needs to rephrase some entries or add multiple pronunciations.
- (8) If more than one grammar is active concurrently, each grammar needs to be tested separately, then they will be tested together (IBM, 2005).
- (9) If weights or probabilities are included in the grammar, the recognition performance should be tested with and without them (Nuance, 2003).

Biber and Kozminski (2005) mentions that the following aspects also should be analyzed and tuned:

- (1) Prompts should be unambiguous to prevent unexpected caller responses.
- (2) Dialogs. Usually users have their own expectations about the dialogs. If these anticipations cannot be met, the mis-recognitions occur and the user might be taken down unexpected paths.
- (3) Confidence thresholds. Tune the proper thresholds to accept the correct and reject the unexpected utterances.

## 5.2 Summary

Extensive testing and tuning are indispensable for speech-enabled applications. Tuning is an iterative process of analyzing system performance. Some aspects are needed to consider in testing, which are summarized as follows:

- (1) Coverage test to test the words and phrase in the grammar.
- (2) Over-generation test to test the words and phrase that are out of

grammars.

- (3) Interpretation test to make sure the grammar delivers expected natural language interpretation.
- (4) Ambiguity test to expose phrases with multiple interpretations.
- (5) Pronunciation test to detect words with unknown pronunciations and misspellings in the grammars.
- (6) Regression test to ensure that no errors have been introduced if the grammar changes.
- (7) Identify the consistent misrecognitions. If a consistent misrecognition is found, it is needed to be rephrased or add multiple pronunciations.
- (8) If more than one grammar is active concurrently, each grammar needs to be tested separately, then they will be tested together.
- (9) If weights or probabilities are included in the grammar, the recognition performance should be tested with and without them.
- (10) Prompts should be unambiguous to prevent unexpected caller responses.
- (11) Tune the proper confidence thresholds to accept the correct and reject the unexpected utterances.

## **6. TOOLS AND ENVIRONMENTS**

Revolutions in the history of technology have shown that the development of technology is driven by basic technology and by tools for developing solutions on top of that technology (Olsen and Klemmer, 2005)(Brad et al, 2000). As speech applications are accepted and adopted widely and widely, the need for more sophisticated Voice User Interfaces (VUI) grows proportionately. Meanwhile, the more sophisticated applications usually mean more complicated grammars. Furthermore, many grammars often need to be dynamically generated based on data obtained at run-time. As speech-recognition grammars grow larger and more complicated, the effective grammar development tools are in urgent needs, such as grammar editors, visualization tools, and the tools for diagnosing and solving problems.

### **6.1 Basic VoiceXML Development Environments**

VoiceXML applications utilize speech technologies for understanding and creating spoken dialogs (HP, 2005). Applications also leverage the Web and server-side technologies (JSP, ASP, CGI) for creating the back-end business logic and generating dynamic data.

Basically, there are two types of development environments for building VoiceXML applications: 1) local Software Development Kits (SDK) and 2) Remote hosts.

The local VoiceXML SDKs provide a variety tools for creating VoiceXML documents and related resources, such as editors and syntax checkers, dialog design tools, grammar design tools, reusable components, and debugger. Some may have rehearsal tools to test dialog flow or other capabilities.

The remote hosts may be Web-based development portal or hosted portal. On the Web-based development portal, developers upload VoiceXML documents



and associated resource files to the portal, and test the application by dialing a pre-assigned phone number. The hosted portal provides developers with the closet deployment environment, where developers upload a VoiceXML application to a document server (application/ Web server), and test the application from a phone. It allows developers to test the full life-cycle of the application, including back-end database access, server-side dynamic data generation, and dialog interaction.

## **6.2 NÜ Echo Grammar Environment**

The NÜ echo grammar environment is to tackle the challenges in grammar design with effective tools for grammar design, debugging, and testing, which address the complete lifecycle of speech-enabled application (NÜ echo, 2005).

The NÜ echo grammar environment is featured with ABNF editor, coverage editor, sentence interpreter (utterance matcher), semantics single-stepper, interactive sentence explorer (phrase enumerator), and grammar converters. The NÜ echo grammar environment is a truly integrated environment, where grammars can be designed in the same environment as the rest of the speech application. All tools in the environment are easy for debugging and tuning grammars at all levels.

This development environment has been extensively field tested. The NÜ echo grammar environment is vendor independent, which can support grammar formats from multiple vendors. It comes as an Eclipse plug-in, which is an open, Java-based extensible integrated development environment, supported by a growing number of organizations.

## **6.3 IBM WebSphere Voice Toolkit**

IBM WebSphere Voice Toolkit is an Integrated Development Environment (IDE) for speech application development. Its runtime server and voice development

tools are based on industry-standard VoiceXML and Java. It supports VoiceXML and Interactive Voice Response (IVR) applications.

The WebSphere Voice Toolkit V6.0 is powered by Eclipse technology and makes it easy to develop VoiceXML applications without having to know the internals of voice technology. The WebSphere Voice Toolkit is full-featured with graphical call flow building, VoiceXML development and debugging, Grammar development and debugging, Pronunciation builder, and Call Control extensible Markup Language (CCXML) development environment.

The WebSphere Voice Toolkit provides the Graphical Grammar Builder for visual composition of a grammar file for speech recognition, and the Prompt Manager for organizing the Audio Files in a voice application. More details can be found at (IBM WebSphere Voice Toolkit, 2010). The free trial of IBM WebSphere Voice Toolkit is available at (IBM Software Download, 2006):

#### **6.4 Microsoft Speech Application SDK (SASDK)**

Microsoft Speech Application SDK (SASDK) is a set of development tools supporting Speech Application Language Tags (SALT) specification which will make it easier and faster for developers to incorporate speech functionality into Web applications (Microsoft, 2005).

The application-authoring tools are seamlessly integrated into Microsoft Visual Studio .NET 2003. Therefore, under a familiar and powerful development environment, developers can easily create, debug and deploy speech-enabled ASP .NET Web applications that can be deployed to a Microsoft Speech Server. In addition to these authoring tools, the SASDK provides a powerful set of ASP .NET Speech controls, a Speech Add-in for Microsoft Internet Explorer, debugging tools, a speech application deployment service, tools for speech application log analysis, sample and reference applications, a rich grammar library, and reference documentation. The Microsoft Speech Application SDK

(SASDK) can be used widely from telephones to Windows Mobile-based devices and desktop PCs.

SASDK version 1.1 can be freely downloaded from (Microsoft Download Center, 2010).

### **6.5 MCM toolkit**

The Metaphor Conversation Manager (MCM) toolkit is a VUI toolkit, which enables the developer to build professional speech applications for Microsoft speech server. MCM leverages the Microsoft Speech Application SDK (SASDK) and its speech Web controls. Using the only syntax C#, MCM provides an all-in-one programming environment to build complete speech applications, from dialogs with callers to back-end integration to communications with live service agents.

The MCM toolkit sits on top of speech Web controls and hides the low-level complexities of building speech applications, such as grammar creation, grammar binding, exception handling, and call event handling (Kuperstein, 2005). The technique of allowing complete control of advanced dialog features eases the user in building speech interfaces. All MCM projects can be exported to standard Visual Studio .NET projects to debug, extend, customize, and deploy, and can be deployed for any language that has a recognition engine

In addition to the development environment, MCM includes post-deployment application management tools. The Application Monitor is a Web interface to observe system performance and modify logging levels under real-time load conditions. The Application Editor enables a non-engineer (e.g. business analyst) to change prompts, adjust business variable, and perform other application revisions in real time, based on business requirements.

More technical documentation on the MCM toolkit is available at:  
[http://www.metaphorsol.com/MCM3\\_docs/MCM\\_3.htm](http://www.metaphorsol.com/MCM3_docs/MCM_3.htm).

## 6.6 Philips Speech SDK

Philips provides a full range of professional services designed to customize speech SDK for voice application development and integration. Philips speech SDK can be used to integrate the latest speech recognition technology from Philips Speech Processing into your applications. Philips speech SDK also includes an easy-to-use C/C++ API so that speech recognition can be integrated into the programming environment. The capabilities of Philips speech SDK include:

- (1) Dictation recognition (speech-to-text).
- (2) Command recognition.
- (3) Verification recognition.
- (4) Spelling recognition.
- (5) Correction functions.
- (6) Natural language understanding.
- (7) Natural dialog between man and machine.
- (8) Audio recording and playback.
- (9) User interface components.

More information is available at:  
<http://www.speechrecognition.philips.com/index.asp?id=521>.

A free trial of Philips Speech SDK can be downloaded from:  
<http://www.speechrecognition.philips.com/index.asp?id=641> .

### **6.7 Hewlett-Packard (HP) OCMP VXML Developer Toolkit**

Hewlett-Packard has developed an Eclipse plug-in, the HP OCMP VXML Developer Toolkit, to add VoiceXML application creation support to Eclipse. This plug-in and the Eclipse platform provide a unified, robust development environment for building VoiceXML applications.

It is easy to use the OCMP VXML Developer Toolkit to (HP, 2005):

- (1) Create VoiceXML 2.0 compliant voice applications.
- (2) Create voiceXML projects, VoiceXML documents, grammar files, ECMA Script files, JSP files, and Prompt text files using specialized wizard.
- (3) Import existing VoiceXML project documents into the VoiceXML application.
- (4) Validate VoiceXML documents, grammar files, and ECMA Script files.
- (5) Build and deploy VoiceXML projects to the OCMP execution platform.
- (6) Launch the OCMP testing environment.

The HP OCMP VXML Developer Toolkit supports VoiceXML 2.0 DTD (version 20021018) and SRGS XML Grammar Form DTD (version 20020820). The HP OCMP VXML Developer Toolkit, HP OCMP SDK, and the Eclipse platform are available at: <http://devresource.hp.com/drc/topics/vxml.jsp> for free of charge.

### **6.8 CSLU Toolkit**

The CSLU Toolkit is a platform for research and development of spoken-language systems. The CSLU Toolkit includes the tools of:

- (1) Audio and visual tools, speech recognition.
- (2) Text-to-speech (TTS), rapid application developer (RAD).
- (3) Language-training wizards, Baldi, SpeechView.
- (4) PSL tools, tutorials, robust parsing, etc.

CSLU Toolkit is easy to use, and powerful for research and development use. The systems work in real world also incorporate research advances. CSLU Toolkit is used for language training, education, corporate uses, research, and corpus development. It is free for research use, and customizable for corporate use.

Detail information and download are available at: <http://www.cslu.ogi.edu/toolkit/>.

## **6.9 Chant Developer Workbench**

Chant Developer Workbench is a tool set for speech application development. The advantages of using Chant Developer Workbench include maximizing speech recognition accuracy, tailoring and enhancing text-to-speech (TTS) playback quality, creating, and testing. In addition, using Chant Developer Workbench, the developer can deploy grammars, lexicons, profiles, and TTS markup with applications and build and enhance a workbench of reusable technologies for developing software that speaks and listens. The Chant Developer Workbench product family includes the following four kits:

- (1) Chant GrammarKit is speech recognition grammar management software that enables the developer to create, modify, and test context-free grammars before integrating and deploying them with the application.
- (2) Chant LexiconKit is lexicon management software that enables the developer to create, delete, modify, extend, backup, and restore user and application lexicon.
- (3) Chant ProfileKit is speech recognition profile management software that enables the developer to create, delete, modify, train, backup, and restore profiles.
- (4) Chant VoiceMarkupKit is text-to-speech (TTS) markup language management software that enables the developer to create, modify, and test TTS markup to enhance the playback quality when synthesizing.

More information is available at:

<http://www.chant.net/Products/Developer%20Workbench/Default.aspx> .

### **6.10 Fonelet's VoiceXML Toolkit**

Fonelet's VoiceXML Toolkit is for those who do not want to spend a lot of time learning VoicXML, but do want to create simple voice-enabled applications. Fonelet VoiceXML Toolkit is web-based and offers a GUI. Fonelet Studio helps build "fast prototype" applications and other quick, simple voice applications. With Fonelet VoiceXML Toolkit, the developer can design interactive dialogs, grammars, address books, Fonelet XML, and Mobile discussion board (to exchange voice and text messages on the phone or via web browsers). Another advantage of Fonelet VoiceXML Toolkit lies on that it does not require programming and VoiceXML knowledge to accomplish the above achievements.

More information is available at:

<http://journals2.iranscience.net:800/www.commweb.com/www.commweb.com/article/COM20011008S0008> .

### **6.11 Wizard of OZ (WOZ)**

The Wizard of OZ (WOZ) experimental prototyping method means that a person simulates the system to be designed (Wasinger, 2001) (Fraser and Gilbert, 1991) (Bernsen and Dybkjær 1995). Suede is a WOZ Prototyping tool for speech user interfaces. It's available at: <http://guir.berkeley.edu/projects/suede/> .

### **6.12 BeVocal Café**

BeVocal Café is a Web-based development platform and voice hosting service for anyone interested in building voice-enabled services in short time. It is a free, Java-based development environment with various valuable tools, documentation, and other resources, for building, debugging, and running voice applications.

BeVocal provides a reliable, secure, high-performance Hosting Network to run the VoiceXML applications. Once built, applications can be easily deployed to production and hosted on BeVocal's proven, carrier-grade VoiceXML Hosting Network.

The free demonstrations offered by Bevocal are available online at: <http://cafe.bevocal.com/> , or by dialing 1-800-BVOCAL.

### **6.13 Tellme Studio**

Tellme Studio is a hosted VoiceXML Platform which allows developers to develop, test, and publish the Internet-powered VoiceXML based applications on the Tellme Network. As a web-based VoiceXML development platform, Tellme Studio provides an external VoiceXML gateway and a configuration application to connect live telephony numbers/ extensions with the VoiceXML-based telephony applications. Tellme Studio features a number of online tools targeted for VoiceXML application development, including online scratchpads, syntax validator, VoiceXML terminal, grammar scratchpad, debug log, grammar phrase checker, phrase generator and DTMF generator. For more information, refer to <https://studio.tellme.com/> .

### **6.14 Voxpilot Open Media Platform**

Voxpilot Open Media Platform is a distributed call control and VoiceXML-based media processing platform designed to enable rapid delivery of next generation interactive DTMF, speech, and multimedia services on a single platform. The Voxpilot Open Media Platform architecture leverages W3C and IETF open standards, which revolutionize the way in which voice services are deployed. The Voxpilot Open Media Platform supports VoiceXML 2.0, SSML, SRGS, SISR and CCXML. The unique combination of VoIP and PSTN interfaces supported by the Voxpilot Open Media Platform makes it ideally positioned to support the migration



from traditional telephony networks to next generation “all-IP” network architectures such as the IP Multimedia Subsystem (IMS). More information is available at <http://www.voxpilot.com/> .

### 6.15 Summary

This section introduced the developing tools, environments, and developing platforms for developers to ease the developments of voice applications. This section covers the basic two types of development environments for building VoiceXML applications, i.e. local Software Development Kits (SDK) and Remote hosts.

The developing tools, environments, and platforms include Nü echo grammar environments, IBM WebSphere Voice Toolkit, Microsoft Speech Application SDK (SASDK), MCM toolkit, Philips speech SDK, CSLU Toolkit, Chant Developer Workbench, Fonelet’s VoiceXML Toolkit, BeVocal Café, Tellme Studio, and Voxpilot Open Media Platform. The summary of the developing tools and environments are in Table 1.

Table 1: Summary of developing tools and environments

No	Tool/ Environment / Platform	Feature	URL
1	local Software Developmen t Kits (SDK)	Including a variety tools for creating VoiceXML documents and related resources, such as editors and syntax checkers, dialog design tools, grammar design tools, reusable components, and debugger. Rehearsal tools to test dialog flow or other capabilities.	
2	Remote hosts (Web-based developmen t portal or hosted portal)	Developers upload VoiceXML documents and associated resource files to the portal, test the application by dialing a pre-assigned phone number. Test the full life-cycle of the application, including back-end database access, server-side dynamic data generation, and dialog interaction.	
3	NÜ echo grammar environment	Including ABNF editor, coverage editor, sentence interpreter (utterance matcher), semantics single-stepper, interactive sentence explorer (phrase enumerator), and grammar converters. An integrated environment, where grammars can be designed in the same environment as the rest of the speech	<a href="http://www.nuecho.com/fr/services/grammar.shtml">http://www.nuecho.com/fr/services/grammar.shtml</a>

		application. All tools are easy for debugging and tuning grammars at all level. Vendor independent. Comes as an Eclipse plug-in, which is an open, Java-based extensible integrated development environment.	
4	IBM WebSphere Voice Toolkit	Integrated Development Environment (IDE). Its runtime server and voice development tools are based on industry-standard VoiceXML and Java. Supports VoiceXML and Interactive Voice Response (IVR) applications. Powered by Eclipse technology, easy to develop VoiceXML applications without knowing the internals of voice technology. Full-featured with graphical call flow building, VoiceXML development and debugging, Grammar development and debugging, Pronunciation builder, and Call Control extensible Markup Language (CCXML) development environment. Graphical Grammar Builder for visual composition of a grammar file. Prompt Manager for organizing the Audio Files in a voice application.	<a href="http://www14.software.ibm.com/webapp/download/preconfig.jsp?id=2006-02-23+09%3A08%3A30.953556R&amp;cat=&amp;fam=&amp;s=z&amp;S_TACT=104AH%20W42&amp;S_CMP">http://www14.software.ibm.com/webapp/download/preconfig.jsp?id=2006-02-23+09%3A08%3A30.953556R&amp;cat=&amp;fam=&amp;s=z&amp;S_TACT=104AH%20W42&amp;S_CMP</a>
5	Microsoft Speech Application SDK (SASDK)	Set of development tools supporting the Speech Application Language Tags (SALT) specification. Application-authoring tools are seamlessly integrated into Microsoft Visual Studio .NET 2003. Set of ASP. NET Speech controls, a Speech Add-in for Microsoft Internet Explorer, debugging tools, a speech application deployment service, tools for speech application log analysis, sample and reference applications, a rich grammar library, and reference documentation. Can be used widely from telephones to Windows Mobile-based devices and	<a href="http://www.microsoft.com/downloads/details.aspx?FamilyId=5DAAE9C4-188C-4547-A9D6-1671132A39A1&amp;displaylang=en&amp;EventType=getsdk">http://www.microsoft.com/downloads/details.aspx?FamilyId=5DAAE9C4-188C-4547-A9D6-1671132A39A1&amp;displaylang=en&amp;EventType=getsdk</a>

		desktop PCs.	
6	Metaphor Conversatio n Manager (MCM) toolkit	A VUI toolkit, an all-in-one programming environment, which enables the developer to build professional speech applications for Microsoft speech server, from dialogs with callers to back-end integration to communications with live service agents. Also including post-deployment application management tools such as: Application Monitor, a Web interface, to observe system performance and modify logging levels under real-time load conditions, and Application Editor to change prompts, adjust business variable, and perform other application revisions in real time.	<a href="http://www.metaphorsol.com/MCM3_docs/MCM_3.htm">http://www.metaphorsol.com/MCM3_docs/MCM_3.htm</a>
7	Philips speech SDK	Integrate the latest speech recognition technology from Philips Speech Processing into your applications. Includes an easy-to-use C/C++ API so that speech recognition can be integrated into the programming environment. The capabilities include: dictation recognition (speech-to-text), command recognition, verification recognition, spelling recognition, correction functions, natural language understanding, natural dialog between man and machine, audio recording and playback, user interface components.	<a href="http://www.speechrecognition.philips.com/index.asp?id=641">http://www.speechrecognition.philips.com/index.asp?id=641</a>
8	Hewlett- Packard (HP) OCMP vXML	An Eclipse plug-in to add VoiceXML application creation support to Eclipse. Provide a unified, robust development environment for building VoiceXML applications. Supports VoiceXML 2.0 DTD (version 20021018) and SRGS XML Grammar Form DTD (version 20020820). Used for: creating VoiceXML 2.0	<a href="http://devresource.hp.com/drc/topics/vxml.jsp">http://devresource.hp.com/drc/topics/vxml.jsp</a>

	Developer Toolkit	compliant voice applications, voiceXML projects, VoiceXML documents, grammar files, ECMA Script files, JSP files, and Prompt text files using specialized wizard, importing existing VoiceXML project documents into the VoiceXML application, validating VoiceXML documents, grammar files, and ECMA Script files, building and deploying VoiceXML projects to the OCMP execution platform, launching the OCMP testing environment.	
9	CSLU Toolkit	A platform for research and development of spoken-language systems. Including the tools of: audio and visual tools, speech recognition, text-to-speech (TTS), rapid application developer (RAD), language-training wizards, Baldi, SpeechView, PSL tools, tutorials, robust parsing, etc. Used for language training, education, corporate uses, research, and corpus development. Free for research use, and customizable for corporate use.	<a href="http://www.cslu.ogi.edu/toolkit/">http://www.cslu.ogi.edu/toolkit/</a>
10	Chant Developer Workbench	The advantages include maximizing speech recognition accuracy, tailoring and enhancing text-to-speech (TTS) playback quality, creating, and testing. Developer can deploy grammars, lexicons, profiles, and TTS markup with applications and build and enhance a workbench of reusable technologies for developing software that speaks and listens. Including: (1) Chant GrammarKit, speech recognition grammar management software, (2) Chant LexiconKit, lexicon management software, (3) Chant ProfileKit, speech recognition profile management software, (4) Chant VoiceMarkupKit, text-to-speech (TTS) markup	<a href="http://www.chant.net/Products/Developer%20Workbench/Default.aspx">http://www.chant.net/Products/Developer%20Workbench/Default.aspx</a>

		language management software.	
11	Fonelet's VoiceXML Toolkit	Web-based with a GUI. It is for those who do not want to spend a lot of time learning VoicXML, but do want to create simple voice-enabled applications. It is used for designing interactive dialogs, grammars, address books, Fonelet XML, and Mobile discussion board (to exchange voice and text messages on the phone or via web browsers). It does not require programming and VoiceXML knowledge.	<a href="http://journals2.iranscience.net:800/www.commweb.com/www.commweb.com/article/COM20011008S0008">http://journals2.iranscience.net:800/www.commweb.com/www.commweb.com/article/COM20011008S0008</a>
12	Wizard of OZ (WOZ) experimental prototyping method	It means that a person simulates the system to be designed. Suede is a WOZ Prototyping tool for speech user interfaces.	<a href="http://guir.berkeley.edu/projects/suede/">http://guir.berkeley.edu/projects/suede/</a>
13	BeVocal Café	It is a Web-based development platform and voice hosting. It is a free, Java-based development environment with various valuable tools, documentation, and other resources, for building, debugging, and running voice applications. It provides a reliable, secure, high-performance Hosting Network to run the VoiceXML applications. Once built, applications can be easily deployed to production and hosted on BeVocal's proven, carrier-grade VoiceXML Hosting Network.	<a href="http://cafe.bevocal.com/">http://cafe.bevocal.com/</a>
14	Tellme	It's a hosted VoiceXML Platform allowing developers to develop, test, and	<a href="https://studio.tellme.com/">https://studio.tellme.com/</a>

	Studio	publish the Internet-powered VoiceXML based applications on the Tellme Network. It provides an external VoiceXML gateway and a configuration application to connect live telephony numbers/ extensions with the VoiceXML-based telephony applications. It includes online scratchpads, syntax validator, VoiceXML terminal, grammar scratchpad, debug log, grammar phrase checker, phrase generator and DTMF generator.	
15	Voxpilot Open Media Platform	A distributed call control and VoiceXML-based media processing platform designed to enable rapid delivery of next generation interactive DTMF, speech, and multimedia services on a single platform. It leverages W3C and IETF open standards. It supports VoiceXML 2.0, SSML, SRGS, SISR and CCXML. The unique combination of VoIP and PSTN interfaces supported by it makes it ideally positioned to support the migration from traditional telephony networks to next generation “all-IP” network architectures such as the IP Multimedia Subsystem (IMS).	<a href="http://www.voxpilot.com/">http://www.voxpilot.com/</a>

## 7. CONCLUSION

As speech technology has achieved significant development over the past three decades, the speech-enabled applications have emerged and been applied widely. It is a complicated task for voice application design, which involves a wide range of techniques. VoiceXML is one of the powerful tools in voice applications development. Grammar design is a critical component determining the performance of grammar-based speech applications. Voice User Interface (VUI) usability is the key to the success of a VoiceXML application. Iterative testing and tuning are indispensable for speech-enabled applications.

In the research reviewed in this survey, we have found 15 voice-application development environments available, 4 of which are freely downloadable, i.e., IBM WebSphere Voice Toolkit, Microsoft Speech Application SDK (SASDK), Philips speech SDK, and CSLU Toolkit. Over 20 articles have been found on grammar design, and 4 of which are refereed scientific papers, i.e., (Yankelovich, 1997) (Boyce, 2000), (Abbott, 2001), and (Mané and Levin, 2005). Over 10 articles are related to Voice User Interface (VUI) design, and 2 of which are refereed scientific papers, i.e., (Boyce, 2000) and (Peissner, 2002). 4 non-refereed articles talk about voice-application testing.

This survey aims to provide practical guidelines for tackling the challenges in grammar design and related problems in VoiceXML-like application development from developers' point of view. The guidelines presented in this survey are proven field-tested experience. Although many of them are generally adaptable, some may just fit to their original situations, not suitable for all applications. The developer needs to take into account the specific situation in voice-application development.



## **APPENDIX B: A SURVEY – AUTOMATIC GENERATION OF SPEECH-RECOGNITION GRAMMARS**

### Table of Contents

#### 1. INTRODUCTION

#### 2. DEFINITIONS

#### 3. MOTIVATION

#### 4. DYNAMIC GRAMMARS

#### 5. AUTOMATIC GRAMMAR GENERATION

##### 5.1 The Knowledge-Based Approaches

##### 5.2 The Data-Driven Approaches

###### 5.2.1 Introduction

###### 5.2.2 Bayesian Model Merging Framework

###### 5.2.3 Growing Semantic Grammar System

###### 5.2.4 Semi-Automatic Grammar Induction Approach

###### 5.2.5 Genetic Programming (GP)

###### 5.2.6 Robust Grammar Authoring Paradigm

###### 5.2.7 Statistical Language Model Generation

###### 5.2.8 Method of Using Counter Examples

###### 5.2.9 Summary

##### 5.3 The Combining Knowledge-Based and Data-Driven Approaches

#### 6. AUTOMATIC SPEECH APPLICATION GENERATION

#### 7. CONCLUSION

## 1. INTRODUCTION

With the growing interest and demand for human-machine interaction, much work on speech recognition has been carried out over the past three decades. A large number of spoken-dialogue systems have been implemented. Aust et al (1995) present Philips system. Jupiter system (weather information system) (Zue et al, 1997, 2000) and AT&T's call redirection system (Riccardi and Gorin, 2000) are pioneer systems. Other examples include the ARISE project (Lamel et al, 2000; Baggia et al, 2000) and Philips Directory-Assistance system (directory information service) (Schramm et al, 2000). Also, an important American project, the DARPA Communicator, has attracted attention from the most important research organizations in USA, such as MIT, BBN, Carnegie Mellon University (Rudnicky et al, 2000 and Carpenter et al, 2001), the University of Colorado (Pellom et al, 2000; Zhang et al, 2001), AT&T (Walker, 2001), Bell Labs, SRI and IBM (Gao et al, 2001). Trias and Marino (2002) discuss the BASURDE [LITE] system, the train travel information and ticket reservation service.

There are two main directions in the natural-language speech recognition research: 1) the grammar-based language model and the 2) statistical language model (SLM). In the grammar-based approach, domain-specific semantic grammars are developed to define the legal utterances in the spoken-dialogue application. A statistical language model adopts a data-driven, statistical modeling approach, which requires a large corpus of training data.

Statistical language models have the advantages of simplicity, flexibility, adaptation, high recognition accuracy, and robust performances. The primary disadvantage is the costly collection of huge amounts of training data and poor generalization with insufficient data. In addition, statistical language models are not supported by readily-available commercial systems, such as VoiceXML (VXML) browsers. Compared to statistical techniques, grammar-based speech recognition is more common and easier to use with reasonable recognition accuracy for small domains. An important advantage over statistical approaches

is that grammar-based approaches do not require the large amount of training data which is difficult and expensive to collect, especially in the early phase of system development (Caskey et al, 2003). However, grammar-based techniques require experts to write high-quality grammars, which can be difficult to maintain and extend. In addition, grammar-based models are not as robust as statistical models, as they cannot handle the out-of-grammar (OOG) utterances (Caskey et al, 2003).

Therefore, statistical approaches are often used for broad and shallow natural language understanding, and grammar-based approaches are frequently used for narrow and deep understanding in a specific domain (Ward and Issar, 1994) (Wang, 2001), where grammars can be crafted carefully to cover as many usages in the domain as possible (Wang, 2002).

Stochastic (statistical) language models were popular around 1995, while the grammar-based language models took the pre-eminent position in commercial products by 2001 (Barnard et al, 1999) (Knight et al, 2001) (Caskey et al, 2003). Also, there are some successful cases combining the two approaches by taking both of their advantages and overcoming each other's weakness with a good balance of speech-recognition accuracy and robustness (Moore et al, 1995) (Knight et al, 2001), (Rayner and Carter, 1997), (Geutner, 1996), (Jones et al, 1993) (Wang et al, 2000) .

While hundreds of spoken dialog systems have been deployed in many different sectors, it is still very costly and laborious to develop such systems due to the long development cycle required to get the application to an acceptable level. One of the main barriers in developing such applications is the development of grammars (Wang and Acero, 2006).

In this survey, we review around 90 scientific papers on automatic generation of speech-recognition grammars and related work. Roughly, the methodologies in automatic grammar generation/ grammar authoring/ grammar induction are

classified into three categories: 1) the knowledge-based approaches, 2) the data-driven approaches, and 3) the combining knowledge-based and data-driven approaches.

Seneff, Dowding et al, Ward, Akiba and Itou, Caskey et al, Wang and Ju are currently active research groups using knowledge-based approaches in automatic grammar generation.

Meanwhile, much work is using data-driven approaches because they are regarded as the approaches that can model real data closely. Meng, Siu, and Wong, at the Chinese University of Hong Kong, are the representative research group who adopt data-driven approaches in semi-automatic grammar induction. Besides, Stolcke and Omohundro, Wang and Waibel, Gavalda and Waibel, Koza, and Yu et al all have proposed various data-driven approaches in automatic grammar generation.

Attempting to take the advantages and avoiding or overcoming the disadvantages, there emerge the approaches combining the knowledge-based approaches and data-driven approaches in automatic grammar generation. The representative research group is Wang and Acero, from Speech Technology Group of Microsoft. In 2001, they proposed a machine-aided grammar authoring system. They claim that this system enables a developer, without knowing the linguistics, to rapidly develop a high-quality grammar for conversational systems. Later on, they applied this system in ATIS (Air Traffic Information System) task in 2002, and further improved the system in 2003, 2005, and 2006.

In addition, in this survey, we review the work on automatic speech application generation, which covers 8 scientific papers. Among this work, Pargellis et al have presented an Automatic Dialogue Generator (ADG), and further upgraded it as Application Generator (AG), which can automatically create and manage user-customized speech-enabled applications.

The remainder of this survey is structured as follows:

Section two introduces some definitions of the technical terms that are used in the survey to help the reader to better understand the contributions of each paper.

In section three, we discuss the motivation of investigating the techniques in automatic grammar generation.

Section four introduces the concept of dynamic grammars and their usage.

Section five presents the methodologies of automatic grammar generation / grammar authoring/ grammar induction in three main categories, knowledge-based approach, data-driven approach, and combining knowledge-based and data-driven approach.

In section six, the techniques in building automatic speech applications are presented.

Finally, section seven concludes with a summary of the survey.

## 2. DEFINITIONS

In this section, we briefly introduce some definitions of the technical terms that are used in the survey to help the reader to better understand the contributions of each paper.

*Grammar-Based Language Model* uses grammars to specify the utterances of the system. In grammar-based language models, domain-specific grammars are developed to define the legal utterances in the spoken-dialogue application.

*Statistical Language Model (SLM)* is a probability distribution  $P(s)$  over all possible sentences  $s$ , or spoken utterances, documents, or any other linguistic units (Rosenfeld, 2000).

*ATIS* is Air Travel Information Service, which is being used by several ARPA-funded sites to develop and evaluate speech-understand systems (Ward and Issar, 1994).

*Domain-Specific Grammar* is a set of rules, like syntactic grammar, defining the legal combination of individual words into constituents and constituents into sentences within the application domain. Also, non-terminals are semantic concepts and their relations in a specific domain (Ward, 1991) (Gavalda, 2000) (Wang and Acero, 2001).

*Dynamic Grammar* is a grammar that can be dynamically created and modified while an application is running (Nuance, 2003). A dynamic grammar can be a file that is referenced using external rules, or it can be created directly in a database using API functions.

*Frequently Requested Listings (FRL) approach* means that a grammar is built based on the information about the most-frequently-requested listings and voice recordings that users refer to. In such a grammar, each listing has a unique corresponding branch, which compiles all the linguistic representations of the

listing that have been collected, with a reference to the listing ID (Identification Number) in the directory.

*Inside-Outside (IO) algorithm* was first introduced by Baker (1979) to infer the parameters of Stochastic Context-Free Grammars (SCFGs) and generalize the parameter estimation methods for HMMs (Hidden Markov Model) to SCFGs. It uses the current rule probabilities and the training set to estimate the expected frequencies of certain types of derivation step, and then compute new rule probability estimates as appropriate ratios of those expected frequency estimates (Pereira and Schabes, 1992).

*Expectation-Maximization (EM) algorithm* is used in statistics for finding maximum likelihood estimates of parameters in probabilistic models, where the model depends on unobserved latent variables. EM alternates between performing an expectation (E) step, which computes an expectation of the likelihood by including the latent variables as if they were observed, and a maximum (M) step, which computes the maximum likelihood estimates of the parameters by maximizing the expected likelihood found on the E step. The parameters found on the M step are then used to begin another E step, and the process is repeated. More details are available in (Dempster et al, 1977), (Frank, 2002).

*Mutual Information (MI)*. In probability theory and information theory, the Mutual Information (or trans-information) of two random variables is a quantity that measures the mutual dependence of the two variables. The most common unit of measurement of mutual information is the bit, when logarithms to the base 2 are used. The value of the Mutual Information (MI) between two random variables indicates the level of the reduction in uncertainty. The higher of MI indicates the larger reduction in uncertainty. Zero MI means the variables are independent (Cover and Thomas, 2006).

*Information Gain (IG)* can be used to define a measure of correlation between two random quantities (Kent, 1983). It measures the difference between two probability distributions. In information theory and machine learning, Information Gain (IG) is an alternative synonym for *Kullback-Leibler divergence*.

*Kullback-Leibler (KL)* is a non-symmetric, non-commutative, measure of the difference between two probability distributions  $P$  and  $Q$ . KL measures the expected difference in the number of bits required to code samples from  $P$  when using a code based on  $P$ , and when using a code based on  $Q$ . Typically  $P$  represents the “true” distribution of data, observations, or a precise calculated theoretical distribution. The measure  $Q$  typically represents a theory, a model, a description or an approximation of  $P$  (Kullback, 1959).

*Manhattan-Norm*. In linear algebra, functional analysis and related areas of mathematics, a *Norm* is a function which assigns a strictly positive length or size to all vectors in a vector space, other than the zero vector. *Manhattan-Norm* is also known as Taxicab metric with corresponding variations in the name of the geometry. It alludes to the grid layout of most streets on the island of Manhattan, which causes the shortest path a car could take between two points in the city to have length equal to the points’ distance in taxicab geometry (Klamroth, 2006).

*Gini Index* (Gini, 1921) is a standard economic measure to see the degree of income inequality in a society. Algebraically, it is defined as “Expected value of the ration of the difference of two arbitrary specimens to the mean value of all specimens”. The *Gini Index* is the Gini Coefficient expressed as a percentage, and is equal to the Gini Coefficient multiplied by 100. The *Gini Coefficient* is a measure of statistical dispersion most prominently used as a measure of inequality of income distribution or inequality of wealth distribution. It is defined as a ratio with values between 0 and 1.

*BLEU (Bilingual Evaluation Understudy) metric* is a metric for evaluating the quality of machine translation output. Quality is considered to be the



correspondence between a machine's output and that of a human. "The closer a machine translation is to a professional human translation, the better it is". (Papineni et.al., 2002). BLEU was one of the first metrics to achieve a high correlation with human judgments of quality, and remains one of the most popular (Callison-Burch et.al., 2006) (Doddington, 2002).

*Temporal clustering*: words or multi-word entries that co-occur sequentially are clustered together based on the Mutual Information (MI) metric or the Information Gain (IG) metric.

*Spatial clustering*: words or multi-word entries with similar left and right linguistic contexts are clustered together based on the symmetric divergence that is applied to the left and right linguistic contexts of the entity pair

*Genetic programming (GP)* is an evolutionary approach that programs computers by natural selection (Koza, 1992) and (Dupont, 1994). In genetic programming, populations of computer programs are genetically bred using the Darwinian principle of survival of the fittest and using a genetic crossover (sexual recombination) operator appropriate for genetically mating computer programs (Koza, 1994).

*Regular Positive and Negative Inference (RPNI) algorithm*: is a framework for identifying any language consistent with a given sample in polynomial time (Oncina and Garcia, 1992).

### 3. MOTIVATION

It was reported that semantic-based robust understanding technology has been widely used in human-machine (Ward, 1990) (Zue et al, 2000) (Wang, 2001) and human-human conversational systems (Waibel, 1996). However, speech-enabled systems have not yet become the mainstream in the real world. Among the existing difficulties, the discrepancy between the lab research and the reality in industry hinders the development of such voice applications (Wang and Acero, 2006). Pieraccini (2004) lists the difficulties and the potential areas of improvement in spoken language understanding research as follows:

- (1) There are few data in the spoken language system design/ development phrase, which is difficult for creating the grammar using data-driven approaches.
- (2) There is a huge amount of data available after application deployment, which is extremely difficult to manually analyze.

In addition, Wang (2001) owes this limited success to the complexity of the following problem:

- (1) compared to the complexity of the target grammar, the available data is typically sparse, and a good generalization mechanism to correctly cover a large variety of language constructions is hard to obtain.

Due to the long development cycle, it is very costly and laborious to develop speech-enabled systems. Also, one of the main reasons that it is not practical for regular developers to implement a conversational system is that, such implementations rely on manual development of domain-specific grammars, a task that is time-consuming, error-prone, and requires extensive language expertise (Meng and Siu, 2001) (Wang and Acero, 2003a) (Wang and Acero, 2006). Siu and Meng (1999, 2002) state that writing grammars is a daunting and expensive task, which forms a major bottleneck in the development of spoken language systems. Furthermore, due to the disfluencies and non-grammatical

utterances of spoken language, a handcrafted grammar cannot guarantee good coverage of real data when deployed in real applications (Meng and Siu, 2001). Bangalore and Johnston (2003) point out that the heavy cost of authoring and maintenance of grammars and inevitable brittleness due to lack of coverage in the rule sets are the main reasons that the scalability of the conversational system is a bottleneck.

Based on discussions with developers, Wang and Acero (2006) summarize the main difficulties in writing a speech-recognition grammar as follows:

- (1) It is hard to anticipate the various alternatives for an expression. For example, “520” can be read as “five two oh”, “five two zero”, “five twenty”, “five hundred twenty”, etc.
- (2) It is hard to normalize speech inputs with Semantic Interpretation (SI) tags, due to the various alternatives.
- (3) It is hard to optimize grammar structures for best recognition performance, for example, with high recognition accuracy and speed.
- (4) The verbosity of XML, which is accepted by Speech Recognition Grammar Specification (SRGS) (W3C, 2004), may be a source of errors in manual grammar development.

Therefore, if conversational systems are to become a mainstream, it is apparent that writing domain-specific grammars must become easier for a typical application developer (Wang and Acero, 2001, 2005, 2006). Therefore, tools for fast grammar authoring/ language learning/ grammar induction/ grammar inference and tools for automatic or semi-automatic adaptation/ learning/ system tuning are important and useful to improve the spoken language system’s performance.

The issue of automatic grammar generation has attracted the attention of researchers for many years (Fu and Booth, 1975a, 1975b), (Carrasco and Oncina, 1994), (Miclet and Higuera, 1996), and (Honavar and Slutzki, 1998),

though most of that work focuses on toy applications. The approaches for natural language processing are not adequate for Natural Language Understanding (NLU) applications (Wang and Waibel, 1998) and (Stolcke and Omohundro, 1994a). Recently, researchers have been working on tools for rapid development of mixed-initiative systems (Glass, 2001) (Glass and Weinstein, 2001) and (Glass et al, 2004), but without addressing the problem of grammar authoring. However, other researchers have developed tools that let an end user refine an existing grammar (Gavaldà, 2000). The revised grammar still relies on the initial grammar. Also it assumes that the developer has a good knowledge of language structures.

With the above concerns, this survey aims to overview the techniques related to automatic/ semi-automatic grammar generation and related technology of speech-recognition grammars.

## 4. DYNAMIC GRAMMARS

A dynamic grammar is a grammar that can be dynamically created and modified while an application is running (Nuance, 2003). Since large grammars will result in significant real time delays while loading, the proper usage of smaller dynamic grammars appear to be a good choice to overcome the scaling problems (Wasinge, 2001) and (Levin and Mané, 2005). Also, a dynamic grammar can result in a more easily maintainable and fluid grammar design for VoiceXML applications (Voxeo, 2006).

A dynamic grammar can be a file that is referenced using external rules, or it can be created directly in a database using API functions. Nuance provides a dynamic grammar mechanism letting the developer create and update grammars at runtime and use them for recognition immediately without needing to recompile the recognition package. The “gate” technique in (Nuance, 2003) is a dynamic grammar that dynamically enables or disables various branches in a static grammar. Voxeo (2006) provides a technique to create a dynamic grammar from a data source, e.g. Microsoft Access. The utterances and return values of the grammar can be obtained from the data source using a server side language. Wang (2001) presents a robust chart parser to support dynamic grammars so that the parser is able to customize the grammar online for different user data.

Levin and Mané (2005) apply dynamic grammars in his project of designing a Voice User Interface (VUI) for Automated Directory (AD) assistance to overcome the scaling problem with the large size of the listing directory. The methods to deal with the large database while automatically generating the grammar in (Levin and Mané, 2005) are discussed as follows:

(1) Extension of the Frequently Requested Listings (FRL) approach to automated grammar generation. With the FRL approach, a grammar is built based on the information about the most-frequently-requested listings and voice recordings users refer to. In such a grammar, each listing has a unique corresponding

branch, which compiles all the linguistic representations of the listing that have been collected, with a reference to the listing ID (Identification Number) in the directory. In speech recognition, by looking up the information associated with the branch, which has a reference to a listing ID and the full listing information, the path in the grammar, which is best matching the user utterance, is found.

In the Extension of the FRL approach, each directory listing is associated with a unique corresponding branch that compiles all the linguistic representations of the listing automatically generated by the variation model from normalized listing name. Its advantage is simple because each branch in the grammar has a reference to the listing ID in the directory. The disadvantages include the fact that the size of the grammars scales with the size of the listing directory because every listing in the grammar is associated with a separate branch. Meanwhile, the daily changed listings cause the grammars to be recompiled and reloaded very frequently which increases the resources and the infrastructure necessary for deployment.

(2) To circumvent the above difficulties in the Extension of the FRL approach, Levin and Mané (2005) propose the approach of separating recognition from search and the use of over-generative grammars. They adopt a compact grammar, which defines an over-generating language, to recognize the listing names, without a separate branch for every listing in the directory. The advantages include the small size of the grammar and less frequently recompiling with the over-generative grammars. The disadvantages include the non-trivial search. Without the association between the recognized utterance and a listing ID, the directory needs to be searched after the recognition, outputting the listings with a high similarity with the recognized utterance. Also, the n-best results need to be confirmed in the VUI design.

A summary of the major work on dynamic grammars is in Table 4.

Table 4: summary of Dynamic Grammars

Date	Authors	Title of the Paper	Major Contribution
2001	Wang, Y.-Y.	Robust Spoken Language Understanding in MiPad	Robust chart parser to support dynamic grammars so that the parser is able to customize the grammar online for different user data.
2003	Nuance Communications, Inc.	Grammar Developer's Guide	"Gate" technique to dynamically enable or disable various branches in a static grammar
2005	Levin, E., Mané, A.	Designing the Voice User Interface for Automated Directory Assistance	1. Extension of the Frequently Requested Listings (FRL) approach to automated grammar generation. 2. Approach of separating recognition from search and the use of over-generative grammars
2006	Voxeo Corporation	<a href="http://www.vxml.org/">http://www.vxml.org/</a>	A technique to create a dynamic grammar from a data source, e.g. Microsoft Access.

## 5. AUTOMATIC GRAMMAR GENERATION

To facilitate the development of speech-enabled applications and services, researchers have been working on tools for rapid development of mixed-initiative systems (Glass and Weinstein, 2001) and (Gavaldà, 2000). Fu and Booth (1975a, 1975b) surveyed the early work on automatic learning of finite state automata (FSA) from training data.

In this survey, we overview the grammar generating techniques and classify them into the following three types: 1) the knowledge-based approaches, 2) the statistical approaches, and 3) the combining knowledge-based and statistical approaches.

### 5.1 The Knowledge-Based Approaches

Due to the fact that statistical language models require large amounts of data, which is costly in terms of time and effort, a way of developing language models without a corpus for a given task at a reasonable cost is needed. This problem has been recognized for many years and has been discussed more recently by (Akiba and Itou, 2000).

Assuming that syntactic structures do not vary across different domains and thus a high level syntactic CFG could be shared by different applications, Seneff (1992) reuses the domain-independent part of a grammar to alleviate the grammar authoring problem. In the natural language system, TINA (Seneff, 1992), an initial set of hand-crafted context-free rewrite rules was first converted to a network structure where the semantic categories were intermixed with syntactic ones. Seneff (1992) uses the domain specific knowledge by replacing the low-level syntactic non-terminals with semantic non-terminals. For example, they may replace the domain-specific concepts HOTEL\_NAME with the noun phrases (NPs).

Also, in (Seneff, 1992), new rules can be automatically generated for the rules



sharing the common elements on the right-hand sides. For example, with the rule  $X \Rightarrow A B C$  and  $X \Rightarrow B C D$ , the system would automatically generate two new rules,  $X \Rightarrow B C$  and  $X \Rightarrow A B C D$ . This ability to automatically generate new rules permits the system to generalize more quickly to unseen structures. Also, it greatly simplifies the implementation, because rules do not have to be explicitly monitored during the parse. However, it can potentially lead to overgeneralization to some extent.

A similar idea of including domain specific semantic features in the typed unification grammar is adopted by Dowding et al (1993) in a natural-language understanding system *Gemini*. Gemini includes a mid-sized constituent grammar of English, a small utterance grammar for assembling constituents into utterances, and a lexicon. All three are written in the typed unification formalism, a variant of unification formalism, used in the Core Language English (Alshawi, 1992). The typed unification provides a facility for grammar development in grammar error analysis and warning of the improperly assigned feature values. In addition, this type analysis is performed statistically at compiling time. There is no run-time overhead for adding types to the grammar.

In Phoenix, the spoken language understanding system, Ward and Issar (1994) model semantics by using Recursive Transition Networks (RTN) to extract information relevant to a task. This limits the grammar rules that can be shared. However, developers can fine tune a grammar without any limitations imposed by a background syntactic grammar. The lexicon to the recognition grammar can be augmented using completion techniques, such as adding the words from the parsing grammar, synonyms and other words related to the words in the training corpus. For example, if the word “Monday” appears in the training corpus, the words “Mondays”, as well as other days like “Tuesday” will be added.

Akiba and Itou (2000) introduce a knowledge-based semi-automatic method of acquiring a language model. This method uses all kinds of knowledge resources to construct the language model. For example, it considers both novice users’

word sequences that are or are not sentences, and expert users' constituents that can make sentences. Also, a corpus is considered as one of knowledge resources. To integrate information from such a range of knowledge resources, a specific class of attribute grammars is used as a uniform representation.

Using Perl code, Gruenstein (2002) implements the automation of the recognition grammar from a list of strings.

Based on the fact that speech grammars need to be used and tuned iteratively, Caskey et al (2003) propose an algorithm to augment the coverage of an existing CFG based on a set of new sentence examples that are not covered by the existing grammar. Using a Finite State Transducer (FST) representation of CFGs, this algorithm attempts to find the minimal set of modifications to the grammar to increase its coverage while preserving its original structure. Also, this proposed approach includes an interactive component to allow developers to control the generalization of the new grammar.

Wang and Ju (2004) provide a way to construct high-performance speech-grammars for alphanumeric concepts, which are common in practice. Using this approach, a developer only needs to write down a regular expression for a concept, the algorithm automatically generates a W3C grammar with appropriate semantic interpretation tags. However, the quality of the grammar is highly dependent on the way the regular expression is written. Preliminary experimental results in (Wang and Ju, 2004) have shown that the generated grammar consistently outperforms the general alphanumeric rules in the grammar library. In some cases the semantic error rates were cut by more than 50%.

The main difficulty with the knowledge-based approach is that, to create high-quality grammars, the grammar developer must have in-depth knowledge of both linguistics and the domain (Wang and Acero, 2006).

A summary of the major work on the Knowledge-Based approaches is listed in Table 5.1.

Table 5.1: summary of Knowledge-Based Approaches

Date	Authors	Title of the Paper	Major Contribution
1992	Seneff, S.	TINA: A Natural Language System for Spoken Language Applications	Reuses the domain-independent part of a grammar to alleviate the grammar authoring problem
1993	Dowding, J., et. al.	Gemini: A Natural language System for Spoken-language Understanding.	A typed unification formalism for grammar development in grammar error analysis and warning of the improperly assigned feature values
1994	Ward, W., Issar, S.	Recent Improvements in the CMU Spoken Language Understanding System.	Model semantics by using Recursive Transition Networks (RTN) to extract information relevant to a task, and allow developers to fine tune a grammar without any limitations imposed by a background syntactic grammar.
2000	Akiba, T., Itou, K	Semi-Automatic language Model Acquisition without Large Corpora	A method of acquiring a language model using all kinds of knowledge resources to construct the language model.
2002	Gruenstein, A.	Automatic Grammar Construction	Implement the grammar automation from a list of strings, using Perl code.
2003	Caskey, S.P., Story, E., Pieraccini, R.	Interactive Grammar Inference with Finite State Transducers.	An algorithm to augment the coverage of an existing CFG based on a set of new sentence examples that were not covered by the existing grammar.
2004	Wang, Y.-Y. and Ju. Y-C.	Creating Speech Recognition Grammars from Regular Expressions for Alphanumeric Concepts.	Construct high-performance speech-grammars for the common-used alphanumeric concepts

## 5.2 The Data-Driven Approaches

### 5.2.1 Introduction

As an alternative to knowledge-based approaches, data-driven approaches are applied because the grammar can model real data closely (Meng and Siu, 2001). Many grammar induction approaches are typically corpus-based, data-driven approaches (Siu and Meng, 1999) (Caskey et al, 2003). The corpus may be annotated with some domain-dependent semantic tags, or domain-independent syntactic tags. The grammar induction algorithms will automatically capture patterns from the data (Meng and Siu, 2001).

Zhou and Ren (1999) classify the statistical grammar-inference into the following two types: 1) supervised learning method - to directly obtain useful syntactic statistics from corpus; 2) unsupervised training method – to automatically acquire syntactic knowledge from raw texts by an iterative algorithm, such as the commonly-used EM (Expectation-Maximization) algorithm (Dempster et al, 1977).

Vidal et al (1993) and Vidal (1994) introduce an Error Correcting Grammar Inference (ECGI) algorithm to infer infinite state grammars that are able to generalize over a set of examples. Wang (1998) presents a statistical word-based grammar-inference approach by ignoring the language structures. Wang (1998) claims that this approach can achieve the good performance comparable to the best commercial systems. Different from the word-based approach of (Wang, 1998), Arai et al (1998) propose a phrase-based approach to automatically generate a collection of grammar fragments each representing a set of syntactically and semantically similar phrases.

Wang and Waibel (1998) adopt a similar approach by using iterative clustering and sequence building operations to find the common structures in a statistical spoken language translation system, which achieves over 10% error reduction for spoken language translation task. Similarly, Pargellis et al (2001) present an

approach to semi-automatically find language structures from training using statistical processing techniques.

### *5.2.2 Bayesian Model Merging Framework*

Stolcke and Omohundro (1994a) present a Bayesian model merging framework to induce probabilistic grammars from corpora of samples. By adding ad-hoc rules, the samples are incorporated to a working grammar, and elements of the models (e.g., states or non-terminals) are merged to achieve generalization and a more compact representation. What to merge and when to stop is governed by the Bayesian posterior probability of the grammar. The framework is so general that it can be instantiated for a variety of probabilistic models, such as Hidden Markov Model (HMM), n-gram, and Stochastic Context-free Grammars (SCFG).

Based on the general “model merging” strategy (Omohundro, 1992), Stolcke and Omohundro (1994b) describe a technique for inducing the structure of Hidden Markov Models (HMMs) from data. By directly encoding the training data, a maximum likelihood HMM is first constructed. Then, more general models are generated by merging the HMM states using a Bayesian posterior probability criterion to determine the states to merge and stop generalizing. This procedure is a heuristic search for the HMM structure with highest posterior probability. With three evaluating applications, Stolcke and Omohundro (1994b) claim that this merging procedure is more robust and accurate.

### *5.2.3 Growing Semantic Grammar (GSG) System*

Gavaldà and Waibel (1998), and Gavaldà (2000) propose the Growing Semantic Grammar (GSG) system, which can aid the end-users who do not have expertise in language processing to rapidly deploy the Natural Language Understanding System (NLU) front-ends and dynamically customize the system. With the collected data, GSG includes the following stages to develop a grammar:

- (1) With a simple editor, GSG designs and analyzes the Domain Model (DM) by building a hierarchical structure of the relevant concepts in the domain.
- (2) A kernel grammar covering a small subset of the collected data is semi-automatically constructed.
- (3) The grammar is dynamically expanded over the collected data and beyond in an interactive environment.
- (4) The grammar is ready to deploy.

#### 5.2.4 Semi-Automatic Grammar Induction Approach

Siu and Meng (1999) propose a statistical approach for semi-automatic grammar induction from un-annotated corpora within a restricted domain. The generated grammar contained both semantic and syntactic structures, which are useful for language understanding. They adopt an iterative procedure to cluster the words while constructing a grammar from an un-annotated corpus of sentences in a restricted domain. *Semi-automatic grammar induction* means that the generated grammar can be further hand-revised to improve quality. The authors have claimed that the algorithm also shows promise in portability across languages. Experiments with the ATIS (Air Travel Information Service) corpus show positive results in semantic parsing, when compared to an entirely handcrafted grammar.

Inspired by McCandless and Glass (1993), and with the similar motivation to Akiba and Itou (2000), Wong and Meng (2001) extend their work described in (Siu and Meng, 1999). The proposed grammar induction is based on agglomerative clustering of words in a corpus of un-annotated sentences from the ATIS domain. Clustering was implemented both spatially and temporally. In spatial clustering, words or multi-word entries with similar left and right linguistic contexts are clustered together. In temporal clustering, words or multi-word entries that co-occur sequentially are clustered together.

Observing that SQL expressions provide information for natural language

structures, Wong and Meng (2001) use the information in the SQL queries in the induced grammar. Instead of Mutual Information (MI), Information Gain (IG) is used to capture phrasal structures, and to determine an automatic stopping criterion for agglomerative clustering. In addition, this approach can be seeded with pre-specified semantic categories to expedite the learning process, and reduce the demand for large training corpora (Wong and Meng, 2001).

In (Wong et al, 2001), the semi-automatic grammar induction approach of (Siu and Meng, 1999) is extended by investigating the use of Information Gain (IG) in place of Mutual Information (MI) for grammar induction based on an unannotated training corpus. The experiments using the ATIS-3 training corpus indicate better precision and recall of desired semantic categories at earlier stages in the grammar induction process while using IG rather than MI. Since grammar induction is an iterative process, Wong et al. (2001) propose an approach to automatically terminate the iteration with a stopping criterion. In (Wong et al, 2001), the coverage of a grammar is measured in terms of the percentage of words/ terminals in the training corpus that are captured in the grammar. If the stopping criterion is defined to be the point where the relative growth in grammar coverage falls below 1%, the grammar induction is terminated at iteration 100. Experiments using the ATIS-3 test sets show promising results compared with the handcrafted and semi-automatic grammars from (Siu and Meng, 1999), based on NLU performance.

Siu and Meng (2001) explore the portability of the semi-automatic grammar induction approach in (Siu and Meng, 1999) to the Chinese language, based on a corpus of translated ATIS-3 queries. To assess grammar quality, Siu and Meng (2001) develop a framework for bi-directional English-Chinese example-based machine translation, where the English and Chinese grammars were obtained by running the semi-automatic grammar induction procedure on the English and Chinese corpora separately. The induced English (/Chinese) grammars are used to parse the input English (/Chinese) queries, and the parsed concepts are then used to generate the Chinese (/English) translation. Siu and Meng (2001) adopt

the translation-by-analogy (also known as Example-Based Machine Translation, or EBMT) approach, which has the advantage that the translation quality can be improved with more available training. Instead of handcrafted grammars, semi-automatically induced grammars reinforce this advantage of EBMT. Experiments on the ATIS-3 test sets show a high percentage (76% to 91%) of user-accepted translations.

Meng and Siu (2002) further describe the semi-automatic grammar induction methodology from un-annotated corpora of information-seeking queries in a restricted domain. The resultant grammars contain language structures that tightly couple semantics with syntax, which are conducive to spoken natural language understanding. Based on the work of (Siu and Meng, 1999) and (Wong and Meng, 2001), Meng and Siu (2002) adopt agglomerative clustering in the grammar induction, which includes temporal clustering and spatial clustering. The induced grammar is amenable to hand-editing for refinement, hence it is *semi-automatic* in nature. Also, it is easily portable across different restricted domains, as well as across languages. While comparing the semi-automatically-induced grammar with a handcrafted grammar in the experiments using ATIS corpus, the handcrafted grammar gave concept error rates of 7% and 11.3% in language understanding, and the semi-automatically induced grammar gave 11% and 12% respectively on the corresponding two test corpora. However, the hand-crafted grammar took two months to develop and the semi-automatically-induced grammar took only three days to produce. These results show a desirable trade-off between language understanding performance and grammar development effort.

Siu et al (2003) report three extensions to the bi-directional English-Chinese Example-Based Machine Translation (EBMT) paradigm (Meng and Siu, 2001) as follows:

- (1) The comparative merits of three distance metrics (i.e., Kullback-Leibler, Manhattan-Norm, and Gini Index) are investigated for agglomerative



clustering procedure, which is discussed in (Siu and Meng, 2001) to generate context-free grammar rules from un-annotated sentences in grammar induction.

- (2) An automatic evaluation method is proposed to generate multiple translation outputs for a single input utterance based on the BLEU metric.
- (3) Siu et al (2003) present a selection strategy that leverages information from the example parse trees in the Example-Based Machine Translation paradigm to improve the performance in Chinese-English translation.

Ju et al. (2009) invent an approach of generating speech-recognition grammars from a data set or big list of items. This method firstly uses a processor to automatically generate a simulated recognition search tree representing items in a data set. Next the processor generates the speech recognition grammar automatically using the simulated recognition search tree and stores the speech recognition grammar for future use in speech recognition.

#### *5.2.5 Genetic Programming (GP)*

Genetic programming (GP) is an evolutionary approach that programs computers by natural selection (Koza, 1992) and (Dupont, 1994). In genetic programming, populations of computer programs are genetically bred using the Darwinian principle of survival of the fittest and using a genetic crossover (sexual recombination) operator appropriate for genetically mating computer programs (Koza, 1994). Hierarchical automatic function definition enables genetic programming to define potentially useful functions automatically and dynamically during a run (Koza, 1994). Genetic programming is a successful technique for getting computers to automatically solve problems, which has been successfully used in a wide variety of problems where solutions can be expressed with modestly short programs (Banzhaf et al, 1998). Dupont (1994) uses a genetic approach to infer grammars for regular languages and compares it with the RPN

(Regular Positive and Negative Inference) algorithm (Oncina and Garcia, 1992) which can identify any regular language in the size limit of samples.

Mernik et al (2003) propose the grammar-specific genetic operators for crossover and mutation to grammar induction. Črepinšek et al.(2004) discuss the search space of CFG induction and propose a Brute Force approach to grammar induction. Owing to the large search space, the exhaustive (brute-force) approach can only be applied to small samples. Therefore, Črepinšek and Mernik (2005) propose a more efficient approach by using genetic-programming with application to inducing grammars from programs written in simple domain-specific languages. The authors claim that experiments show that the genetic approach is comparable to other grammatical inference approaches.

#### *5.2.6 Robust Grammar Authoring Paradigm*

Since it is very common that users' speech does not conform to a rigid CFG, it is important to build robust grammars for voice systems. Yu et al (2006) propose a robust grammar authoring paradigm to transfer rigid Context-Free Grammars (CFGs) into more robust semantic CFGs. This system takes a simple CFG as input, using n-gram based Filler Models (FMs) to model the garbage words between slots, and generates a hybrid n-gram/ CFG in W3C SRGS (Speech Recognition Grammar Specification) format which can run in many standard ASR (Automatic Speech Recognition) engines.

The authoring paradigm in (Yu et al, 2006) is described as follows: a basic grammar is constructed with pre-ambles, post-ambles, and slots. Pre-ambles and post-ambles are fillers modeled with word n-grams, and slots carry semantic information such as numbers, a list of commands, date, time, currency, and credit card number, etc. Thus, to build such a grammar, developers only need to provide a slot grammar (e.g., a name list, cardinal or ordinary number, and date time, etc) and plug it into the above structure. The slot grammar can be from a reusable library grammar or created with grammar controls.

The authors claim that the generated robust CFG does not lose accuracy for test utterances that are covered by the rigid CFG, also greatly improves the robustness of the speech applications by covering more cases than the rigid CFG. Especially, the paradigm is good at rejecting the out-of-grammar (OOG) utterances and recognizing utterances with pre-ambles or post-ambles. They also claim that the robust CFG can achieve recognition accuracy close to the class-based n-gram LM customized for the application, with a few example sentences for adaptation. In addition, the authors demonstrate that the proposed paradigm is superior in many aspects to other filler models.

### *5.2.7 Statistical Language Model Generation*

Metz (2008) invents a speech processing solution to automatically tune the grammar using statistical language model generation. Firstly, one or more speech recognition grammars are applied to the speech-recognition system for multiple recognition instances by performing a plurality of speech-to-text operations. Then, based on the obtained recognition instances, the system automatically creates a set of words and phrases and automatically weighs the words and phrases based upon the recorded recognition data. So, a replacement grammar can be automatically generated from the set of words and phrases. The replacement grammar is a statistical language model grammar, though the original speech-recognition grammar can be either a grammar-based language model grammar or a statistical language model grammar. And the original speech recognition grammar can be written in various grammar format specification languages, such as a NUANCE Grammar Specification Language (GSL), a Speech Recognition Grammar Specification (SRGS) compliant language and a JAVA Speech Grammar Format (JSGF) compliant language.

A performance analyzer is configured to compare the performance of original speech recognition grammar with that of the replacement grammar. The decision of whether to replace the original grammar depends on the test for the speech-recognition performance. Metz (2008) presents an administrator of the speech-

recognition system with an option to replace the original speech recognition grammar.

#### *5.2.8 Method of Using Counter Examples*

Zadrozny and Kambhatla (1999) invent a method of taking counter examples to generate and revise speech-recognition grammars to reduce errors in the overall system. In this method, an initial grammar is given in Backus-Naur Form (BNF) notation. The author uses a sentence generator to generate a list of all sentences that are accepted by the grammar and identify the inappropriate or irrelevant sentences that are accepted by the grammar (counter-examples) from the list. With the assistance of the list of counter examples and the original grammar, a grammar reviser program prunes the list and generates a revised grammar. The revision process is iterated several times until is deemed satisfactory in that it accepts only relevant sentences.

#### *5.2.9 Summary*

One advantage of the corpus-based grammar induction approaches lies in the fact that the grammars produced model real data closely. One disadvantage of such approaches is the requirement of the large amount of data, and the annotation of the large corpus (which is adopted by some approaches) is time-consuming and costly. Also, the data-driven approaches suffer from the data sparseness problem, and are generally very slow (Caskey et al, 2003). In addition, the quality of the inducted grammars, using the purely bottom-up, data-driven grammar inference algorithms, cannot be guaranteed (Wang and Acero, 2006).

A summary of the major work on the Data-Driven approaches is in Table 5.2.9.

Table 5.2.9: summary of Data-Driven Approaches

Date	Authors	Title of the Paper	Major Contribution
1979	Baker, J.	Trainable Grammar for Speech Recognition	Inside-Outside (IO) algorithm to automatically infer probabilistic context-free grammars (PCFGs)
1992	Pereira, F., Schabes, Y.	Inside-outside Reestimation from Partially Bracketed Corpora	Combining structured and stochastic models in grammar induction
1992 1994	(1) Omohundro, S. (2) Stolcke and Omohundro	(1) Best-first Model Merging for Dynamic Learning and Recognition (2) Inducing Probabilistic Grammars by Bayesian Model Merging	“model merging” strategy to induce the structure of Hidden Markov Models (HMMs) from data
1993, 1994	(1) Vidal, E., Casacuberta, F., Garcia, P. (2) Vidal, E.	(1) Grammatical Inference and Applications to Automatic Speech Recognition and Understanding (2) Grammatical Inference: An Introductory Survey	An Error Correcting Grammar Inference (ECGI) algorithm to infer infinite state grammars that are able to generalize over a set of examples
1994	Stolcke, A., Omohundro, S.M.	Best-first Model Merging for Hidden Markov Model Induction	Bayesian model merging framework to induce probabilistic grammars from corpora of samples
1994	Dupont, P.	Regular Grammatical Inference From Positive and Negative Samples by Genetic Search: The	Uses a genetic approach to infer grammars for regular languages and compares it with the RPNI (Regular Positive and Negative Inference) algorithm (Oncina and

		GIG Method	Garcia, 1992)
1998	Wang, Y. –Y	Grammar Inference and Statistical Machine Translation	A statistical word-based grammar-inference approach by ignoring the language structures
1998	Arai, K., Wright, J., Riccardi, G., Gorin, A.	Grammar Fragment Acquisition Using Syntactic and Semantic Clustering	A phrase-based approach to automatically generate a collection of grammar fragments each representing a set of syntactically and semantically similar phrases
1998	Wang, Y.-Y., Waibel, A.	Modeling with Structures in Statistical Machine Translation	Use iterative clustering and sequence building operations to find the common structures in a statistical spoken language translation system
1998 2000	(1) Gavalda, M., Waibel, A. (2) Gavalda, M.	(1) Growing Semantic Grammars (2) Growing Semantic Grammars (Ph. D Thesis)	Growing Semantic Grammar (GSG) system to aid the non-expert end-users to rapidly deploy the Natural Language Understanding System (NLU) front-ends and dynamically customize the system
1999	Meng, H., Siu, K.C.	Semi-Automatic Acquisition of Domain-Specific Semantic Structures	semi-automatic grammar induction from un-annotated corpora within a restricted domain
1999	Zadrozny, W. and Kambhatla, N.	Method and Apparatus for Creating Speech Recognition Grammars Constrained by Counter Examples.	Invent a method of taking counter examples to generate and revise speech-recognition grammars to reduce errors in the overall system.
2000	Chelba, C., Jelinek, F	Structured Language Modeling	Infer grammars using the methods on the combination of

			structured and stochastic models
2001	Pargellis, A., Fosler-Lussier, E., Potamianos, A., Lee, C.-H.	Metrics for Measuring Domain Independence of Semantic Classes	An approach to semi-automatically find language structures from training using statistical processing techniques
2001	Wong, C.-C., Meng, H.	Improvements on a Semi-automatic Grammar Induction Framework.	Grammar induction approach, based on agglomerative clustering of words in a corpus of un-annotated sentences from the ATIS domain. Information in the SQL queries is used in the induced grammar.
2001	Wong, C.C., Meng, H., and Siu, K.C.	Learning Strategies in a Grammar Induction Framework	Use Information Gain (IG) in place of Mutual Information (MI) for grammar induction based on an un-annotated training corpus
2001	Siu, K. C. and Meng, H.	Semi-Automatic Grammar Induction for Bi-directional English-Chinese Machine Translation	Explore the portability of the semi-automatic grammar induction approach to the Chinese language, based on a corpus of translated ATIS-3 queries
2002	Meng, H., Siu, K. C.	Semiautomatic Acquisition of Semantic Structures for Understanding Domain-Specific Natural Language Queries.	Agglomerative clustering in the grammar induction. Resultant grammars contain language structures that tightly couple semantics with syntax, which are conducive to spoken natural language understanding.
2003	Siu, K. C., Meng, H.,	Example-based Bi-directional	Three extensions to the bi-directional English-Chinese

	Wong, C. C.	Chinese English Machine Translation with Semi-automatically Induced Grammars.	Example-Based Machine Translation (EBMT) paradigm in (Meng and, 2002).
2003	Mernik, M., Gerlić, G., Zumer, V., Bryant, B.	Can a parser be generated from Examples?	Grammar-specific genetic operators for crossover and mutation to grammar induction.
2006	Yu, D., Ju, Y.-C., Wang, Y., Acero A.	N-Gram Based Filler Model for Robust Grammar Authoring.	Robust grammar authoring paradigm to transfer rigid Context-Free Grammars (CFGs) into more robust semantic CFGs
2008	Metz, B.D.	Automatic Grammar Tuning Using Statistical Language Model Generation	Automatically tunes the grammar using statistical language model generation. Presents an administrator of the speech recognition system with an option to replace the original speech recognition grammar
2009	Ju, Y., Ollason, D., Bhatia, S.	Method and apparatus for automatic grammar generation from data entries	Invent an approach of generating speech-recognition grammars from a data set or big list of items, involving simulated recognition search tree.



### 5.3 The Combining Knowledge-Based and Data-Driven Approaches

While knowledge-based approaches and data-driven approaches both have advantages and disadvantages in natural language processing, much work emerges to combine these two approaches attempting to take their advantages and overcome the disadvantages. Also, the interest of combining the knowledge-based and data-driven models has recently increased in grammar authoring research.

Based on bracket matching schemes, Zhou and Ren (1999) propose an approach for automatically generating Chinese Probabilistic Context-Free Grammars (PCFGs). They annotate the training texts with constituent boundary information and use the bracket matching schemes upon boundary predicted texts to implement the EM (Expectation-Maximization) algorithm (Dempster et al, 1977). Different knowledge resources, such as the automatically generated grammar, and a set of special rules summarized by linguists or extracted from tree banks, are integrated to obtain a better initialization for the learning process. The authors claim that the experimental results show good learning efficiency of this algorithm and high reliability of the generated grammar. This proposed method guarantees an automatically generated grammar with a broad-coverage and a good bootstrapping for the learning process.

Combining a domain-specific semantics, a library grammar, syntactic constraints and a small amount of example sentences that have been semantically annotated, Wang and Acero (2001) propose a machine-aided grammar authoring system. The authors claim that it enables a programmer, without knowing the linguistics, to rapidly develop a high-quality grammar for conversational systems.

The grammar is generated from the following three inputs: 1) a semantic schemas defining the domain semantics, 2) a grammar library that contains CFG rules for domain-independent concepts (e.g., Date and Time) or domain-specific semantic terminals (e.g., city names and airlines), and 3) semantically annotated

training data. Wang and Acero (2001) inherit semantic constraints from schema, use annotation to reduce the search space, and use syntactic clues to align the remaining words of the sentence.

Wang and Acero (2001) ascribe the consistently better understanding accuracies with much less authoring effort than the manually authored grammar to the following three reasons: 1) data driven learning, 2) the template grammar, and 3) the use of multiple information source. The authors claim that the grammar authoring tool greatly eases semantic grammar development by integrating different information sources and learning from annotated examples to induct CFG rules.

To study the general applicability of the algorithm as well as to provide the research community with more informative results, Wang and Acero (2002) have applied the algorithm in (Wang and Acero, 2001) to the well studied Airline Travel Information System (ATIS) task (Dahl et al., 1994) and compare the performance of the learned grammar with one of the best performers in ATIS evaluations. The results show that the semi-automatically learned grammar achieves comparable performance to the manually authored grammar. In addition, the smaller size and the common paradigm of the learned grammar may make the system work faster and be easier to maintain.

While the example-based grammar authoring tool (Wang and Acero, 2001) has some basic learning capabilities, they often require users' intervention to solve the ambiguities to induct grammar rules, which is very intrusive and greatly slows down the grammar development. Wang and Acero (2003a) present an Expectation-Maximization (EM) algorithm (Dempster *et al.*, 1977) to automatically resolve the segmentation ambiguities. Their preliminary experiment results show that this algorithm not only eliminates the human involvement in ambiguity resolution, but also improves the overall spoken language understanding accuracy.

Although the grammar authoring tool of (Wang and Acero, 2001) can significantly reduce the effort in grammar development with promising understanding accuracy across different domains, it has the following limitations (Wang and Acero, 2003b):

- It only works well for slot-rich high resolution understanding tasks.
- The generated grammars only work well with robust understanding technology.

With these concerns, Wang and Acero (2003b) propose a composite model of HMM and CFG, a modification to the model in (Wang and Acero, 2001), by combining semantic CFG and n-gram statistical model. The HMM models the template rules and the n-gram pre-terminals; the CFG models the library grammar. This combined CFG/n-gram model overcomes the robustness and the scalability problem existing in the semantic grammar model described in (Wang and Acero, 2001). The authors claim that the preliminary results show the 32% error reduction in high resolution understanding of the new model.

To further tackle the problem when little data is available at initial states of data-driven grammar-learning system development, Wang and Acero (2005, 2006) present SGStudio, which significantly reduces the requirement for large amount of training data. *SGStudio* is an example-based grammar authoring tool. The authors claim that it enables software developers with little speech/linguistic background to rapidly create quality semantic grammars for speech-driven applications. SGStudio includes the following components: the knowledge-assisted example-based grammar learning, grammar control, and configurable grammar structure. SGStudio adopts the HMM/ CFG composite model which integrates the domain knowledge in the data-driven grammar learning framework. The HMM/ CFG composite model uses CFGs as the lexicalization models for slot fillers, which generally model a specific concept. The concept can be domain-independent, like date and time, or domain-dependent, such as insurance policy number and auto part numbers. Grammar controls and the control operations

provide the tools to generate various grammars for the concepts that can be used in a system-initiated dialog or as the filler of a slot in a mixed-initiative system. The combination of the knowledge-based and data-driven approaches achieves the balance between robustness and accuracy better than or comparable to the best manually developed grammars.

A summary of the major work on the approaches of combining Knowledge-Based and Data-Driven is listed in Table 5.3.

Table 5.3: summary of the Combining Knowledge-Based and Data-Driven Approaches

Date	Authors	Title of the Paper	Major Contribution
1999	Zhou, Q., Ren, F.	Automatic Inference for Chinese Probabilistic Context-Free Grammar	An approach for automatically generating Chinese Probabilistic Context-Free Grammars (PCFGs)
2001	Wang, Y.-Y., Acero, A.	Grammar Learning for Spoken Language Understanding.	A machine-aided grammar authoring system, to rapidly develop a high quality grammar for conversational systems, by combining a domain-specific semantics, a library grammar, syntactic constraints and a small amount of example sentences that have been semantically annotated
2002	Wang, Y.-Y., Acero, A.	Evaluation of Spoken Language Grammar Learning in the ATIS Domain	Apply the algorithm in (Wang and Acero, 2001) to the well studied Airline Travel Information System (ATIS) task and compare the performance of the learned grammar with one of the best performers in ATIS evaluations
2003	Wang, Y.-Y., Acero, A.	Concept Acquisition in Example-Based Grammar Authoring	An Expectation-Maximization (EM) algorithm (Dempster <i>et al.</i> , 1977) to automatically resolve the segmentation ambiguities
2003	Wang, Y.-Y., Acero, A.	Combination of CFG and N-gram Modeling in Semantic Grammar Learning	A composite model of HMM and CFG, a modification to the model in (Wang and Acero, 2001), by combining semantic CFG and n-gram statistical model
2005	Wang, Y.-Y.,	(1) SGStudio: Rapid Semantic	SGStudio significantly reduces the requirement for large

2006	Acero, A.	Grammar Development for Spoken Language Understanding. (2) Rapid Development of Speech Recognition Grammars	amount of training data
------	-----------	----------------------------------------------------------------------------------------------------------------	-------------------------

## 6. AUTOMATIC SPEECH APPLICATION GENERATION

A spoken dialogue system consists of the following components: a speech recognition component, a language understanding component, a dialogue management component, a component for communication with an external system, a response generation component, and a speech output component (McTear, 2002), (Han, 2000), and (Glass, 1999). These components work in a sequential stream, in which the first component receives the user's input, the output from that component feeds into the next component as the input, and so forth, until the consequent voice output is synthesized for the user. It is a great challenge to build each component of the spoken-dialogue system. To facilitate the creation of speech-enabled systems, research work to automatically generate the various components has been carried out.

Pargellis et al (1999a) present an Automatic Dialogue Generator (ADG), which is a software engine with associated library files to simplify the generation of new voice applications. Given any task description in tables, the ADG can automatically generate a finite-state dialogue for that task in a uniform and consistent fashion. The advantages of using an ADG to generate dialogues include:

- (1) prompts and grammars are generated in a consistent manner,
- (2) prompts and grammars are generated dynamically, and
- (3) user-specified applications can be quickly generated.

Given the advantages of the Automatic Dialogue Generator (ADG) (Pargellis et al, 1999a), Pargellis et al (1999b) propose the Application Generator (AG), a system that automatically creates, and then manages, user-customized speech-enabled applications. The AG consists of four modular components: the Automatic Dialogue Generator (ADG), the Profile Manager (PM), the Information and Services Manager (ISM), and the Dialogue Manager (DM):

- (1) The Profile Manager (PM) uses a Q&A session to obtain knowledge about the user's preferences, and stored this in a user profile.
- (2) The Information and Services Manager (ISM) accesses the available databases and services, including distributed sources such as the World Wide Web or corporate file systems.
- (3) The Automatic Dialogue Generator (ADG) combines these data with the profile and builds a speech interface by generating a series of dialogue states, with associated grammars and system prompts.
- (4) Finally, the Dialogue Manager (DM) interfaces between the user and the dialogue specification are generated by the ADG.

AG is a platform that automatically generates a dialogue model by matching a user's expectations with the system's capability and available resources (Pargellis et al, 1999b). One advantage of AG is that it enables each user to define his/her own dialogue session. Therefore, dialogue interactions are more accurate, faster, and rewarding.

To facilitate the creation of mixed-initiative spoken-dialogue systems for both novice and experienced developers, Glass (2001), Glass and Weinstein (2001b) present SPEECH-BUILDER, a suite of tools, which employs intuitive methods of specification to allow developers to specify domain-dependent linguistic information and create spoken dialogue interfaces. Using SPEECH-BUILDER, instead of defining the language grammars, developers specifies the basic semantic concepts (keys) and provides examples of user utterances to trigger different system behaviors (actions). Based on the inputs, the system automatically configures the speech recognition, language understanding, language generation, and discourse components. Also, a hierarchical grammar can be generated if the developer uses bracketing to label portions of the example sentences as being subject to a particular structure.

Several spoken-dialogue systems in different domains have been created using SPEECH-BUILDER, such as a directory of the people working at the MIT



Laboratory for Computer Science, an application to control the various physical items in a typical office environment, and a system for real-time weather information access. Since November 2000, SPEECH-BUILDER has been accessible from within MIT and limited other locations for beta-testing.

Motivated by a desire to minimize the need for a pre-determined dialogue flow, Polifroni et al (2003) propose an approach that automatically builds a mixed-initiative dialogue system from online knowledge resource. In this approach, decisions on dialogue flow are made dynamically based on analyses of data, either prior to user interaction or during the dialogue itself. Polifroni et al (2003) also introduce a simulation server to examine the operation of the overall dialogue system, particularly the interaction between the dialogue flow and the response generation outputs. Overall, these techniques aim towards the goal of creating new domains automatically with little or no human input. Furthermore, automatic methods are more adaptable and robust against frequent online changes (Polifroni et al, 2003).

Glass et al (2004) further discuss the framework in (Glass, 2001), and (Glass and Weinstein, 2001), SPEECH-BUILDER, which facilitates the creation of mixed-initiative conversational interfaces for novice and expert developers of human language technology. SPEECH-BUILDER has a web-based interface, where developers can specify the information about the interactions between a human and a spoken dialogue system. SPEECH-BUILDER uses XML to store information that is human-readable. With the configuration, the developer can use a web-interface to compile it. Using the specified information and example sentences provided by the developer, this compilation process takes usually one or two minutes. After the compilation, the developer can examine the resulting grammar, deploy the system, talk to it and subsequently iteratively refine aspects of the understanding, generation, dialogue, etc.

A summary of the major work on automatic speech application generation is listed in Table 6.

Table 6: summary of automatic speech application generation

Date	Authors	Title of the Paper	Major Contribution
1999	Pargellis, A., Kuo, H., Lee, C.	Automatic Dialogue Generator Creates User Defined Applications	Automatic Dialogue Generator (ADG), a software engine with associated library files to simplify the generation of new voice applications
1999	Pargellis, A., Kuo, H., Lee, C.	Automatic Application Generator Matches User Expectations to System Capabilities	Application Generator (AG), a system that automatically creates, and then manages, user-customized speech-enabled applications
2001	(1) Glass, J., Weinstein, E. (2) Glass, J.	(1) SPEECH-BUILDER: Facilitating Spoken Dialogue System Development  (2) SPEECH-BUILDER: Facilitating Spoken Dialogue System Development, (MIT M.Eng. Thesis)	SPEECH-BUILDER, a suite of tools, employs intuitive methods of specification to allow developers to specify domain-dependent linguistic information and create spoken dialogue interfaces
2003	Polifroni, J., Chung, G., Seneff, S.	Towards Automatic Generation of Mixed-Initiative Dialogue Systems from Web Content	An approach automatically builds a mixed-initiative dialogue system from online knowledge resource
2004	Glass, J., Weinstein, E., Cyphers, S., Polifroni, J., Chung, G., Nakano, N.	A Framework for Developing Conversational User Interfaces	Further improve the framework in (Glass, 2001), and (Glass and Weinstein, 2001), SPEECH-BUILDER, to facilitate the creation of mixed-initiative conversational interfaces for novice and expert developers of human language technology

## 7. CONCLUSION

Over the past three decades, much work on speech recognition technology has been carried out, and a large number of spoken-dialogue systems have been developed. However, speech-enabled systems have not become the mainstream yet in the real world. Among the existing barriers, writing a domain-specific grammar, which is time-consuming, error-prone, and requires intensive language expertise, forms a major bottleneck in the development of spoken-language systems (Wang and Acero, 2001, 2005, 2006), (Meng and Siu, 2001), (Wang and Acero, 2003a), (Bangalore and Johnston, 2003), and (Wang and Acero, 2006).

While conducting this survey, we have noticed that there is not much work on automatic grammar authoring, language learning/ grammar induction/ grammar inference or the tools for automatic or semi-automatic adaptation/ learning/ system tuning. The issue of automatic grammar generation attracted the attention of researchers as early as 1975 (Fu and Booth, 1975a, 1975b), however most of the work focused on toy problems (Carrasco and Oncina, 1994), (Miclet and Higuera, 1996), and (Honavar and Slutzki, 1998). Gavalda (2000) develops tools to let an end user refine an existing grammar, but it requires the developer a good knowledge of language structures. Recently, Glass (2001), Glass and Weinstein (2001), and Glass et al (2004) have presented tools (i.e., SPEECH-BUILDER) for rapid development of mixed-initiative systems, but they did not address the problem of grammar authoring.

In summary, the methodology in automatic speech-recognition-grammar generation is classified into the following three types: 1) knowledge-based approaches, 2) data-driven approaches, and 3) combining knowledge-based and data-driven approaches. 6 refereed scientific papers use knowledge-based paradigm, around 30 scientific papers use data-driven approaches, and around 10 scientific papers combine knowledge-based and data-driven approaches. Also, we have found 6 refereed scientific papers discussing the tools for automatic

speech application generation, and 5 scientific papers talking about dynamic grammars.

Also, we have observed some research groups working actively in automatic generation of speech-recognition grammars and speech applications with continuing work, such as Meng, Siu, and Wong, at the Chinese University of Hong Kong ( <http://www.cuhk.edu.hk/v6/en/> ), using statistical approaches, Wang and Acero, at Speech Technology Group of Microsoft (<http://research.microsoft.com/research/srg/>), using combined knowledge-based and data-driven approaches, Glass, and Weinstein, at Spoken Language Systems Group MIT Laboratory for Computer Science ( <http://www.sls.lcs.mit.edu> ), working on the tools (i.e., SPEECH-BUILDER) for rapid development of mixed-initiative systems.

**APPENDIX C: GRAMMAR – READ A BOOK**

```
/* read_a_book.gram */
grammar read_a_book;

public <s> = <command>;

<command> = <read_command> <place> [(to | until) <place>]
 | <question>
 | <greeting>
 | <help>;

<read_command> = [please] (read from
 | start reading from
 | go to);

<place> = the beginning
 | <order_number> (page | chapter)
 | page <number>
 | chapter <under_hundred>
 | <word>;

/* order number is used limited to tenth */
<order_number> = first
 | second
 | third
 | fourth
 | fifth
 | sixth
 | seventh
 | eighth
 | ninth
 | tenth;

<number> = <under_hundred>
 | <hundreds>
 | <thousands>;

<under_hundred> = <digit>
 | <teens>
 | <tens> [<digit>];

<hundreds> = <digit> hundred [<under_hundred>];

<thousands> = <digit> thousand [<hundreds>];
```

```

<digit> = one
 | two
 | three
 | four
 | five
 | six
 | seven
 | eight
 | nine;

<teens> = ten
 | eleven
 | twelve
 | thirteen
 | fourteen
 | fifteen
 | sixteen
 | seventeen
 | eighteen
 | nineteen;

<tens> = twenty
 | thirty
 | forty
 | fifty
 | sixty
 | seventy
 | eighty
 | ninety;

<question> = what is the book title
 | who wrote the book;

<greeting> = hello
 | hello there
 | hi, how are you
 | goodbye
 | bye-bye;

<help> = help
 | help please
 | please help
 | I need help
 | give me a help;

/* <word> will include all the words from the story
 distinctly (without duplicate) */

```

<word>= king| queen| said| day| had| child| one| happened|  
 bathing| frog| crept| water| land| wish| fulfilled|  
 year| gone| have| daughter| came| contain| joy|  
 ordered| feast| invited| kindred| friends|  
 acquaintances| women| thirteen| kingdom| twelve|  
 plates| eat| left| home| held| manner| splendor|  
 end| bestowed| gifts| baby| gave| virtue| beauty|  
 riches| world| eleven| made| promises| thirteenth|  
 wished| avenge| greeting| looking| cried| voice|  
 fifteenth| prick| spindle| fall| dead| saying|  
 word| turned| room| shocked| twelfth| remained|  
 unspoken| undo| sentence| soften| death| sleep|  
 hundred| years| princess| fain| keep| misfortune|  
 orders| burnt| saw| love| maiden| palace| went|  
 sorts| places| looked| rooms| bed-chambers| liked|  
 tower| climbed| winding-staircase| reached| door|  
 key| lock| sprang| sat| woman| spinning| flax|  
 mother| doing| nodded| head| rattles| took| wanted|  
 spin| touched| decree| pricked| finger| moment|  
 felt| fell| bed| stood| lay| extended| come|  
 entered| hall| began| go| court| horses| stable|  
 dogs| yard| pigeons| roof| flies| wall| fire|  
 flaming| hearth| became| meat| frizzling| cook|  
 going| pull| hair| boy| forgotten| let| wind|  
 trees| castle| leaf| moved| grow| hedge| thorns|  
 grew| seen| flag| story| briar-rose| named|  
 country| time| sons| tried| get| found| hands|  
 youths| caught| died| heard| man| thorn-hedge|  
 stand| grandfather| kings| youth| see| dissuade|  
 listen| words| passed| awake| son| parted| accord|  
 pass| closed| hounds| lying| wings| house| kitchen|  
 holding| hand| seize| maid| hen| pluck| throne|  
 breath| opened| sleeping| turn| eyes| stooped|  
 kiss| kissed| awoke| astonishment| shook| jumped|  
 wagged| tails| pulled| heads| flew| flickered|  
 cooked| joint| sizzle| box| ear| screamed|  
 plucking| fowl| marriage| celebrated| lived|  
 contented| days;

## APPENDIX D: GRAMMAR – READ A BOOK (5-word sequence)

```

/* read_a_book_5_word_sequence.gram */

grammar read_a_book_5_word_sequence;

public <s> = <word> <word> <word> <word> <word> ;

<word>= king| queen| said| day| had| child| one| happened|
 bathing| frog| crept| water| land| wish| fulfilled|
 year| gone| have| daughter| came| contain| joy|
 ordered| feast| invited| kindred| friends|
 acquaintances| women| thirteen| kingdom| twelve|
 plates| eat| left| home| held| manner| splendor|
 end| bestowed| gifts| baby| gave| virtue| beauty|
 riches| world| eleven| made| promises| thirteenth|
 wished| avenge| greeting| looking| cried| voice|
 fifteenth| prick| spindle| fall| dead| saying|
 word| turned| room| shocked| twelfth| remained|
 unspoken| undo| sentence| soften| death| sleep|
 hundred| years| princess| fain| keep| misfortune|
 orders| burnt| saw| love| maiden| palace| went|
 sorts| places| looked| rooms| bed-chambers| liked|
 tower| climbed| winding-staircase| reached| door|
 key| lock| sprang| sat| woman| spinning| flax|
 mother| doing| nodded| head| rattles| took| wanted|
 spin| touched| decree| pricked| finger| moment|
 felt| fell| bed| stood| lay| extended| come|
 entered| hall| began| go| court| horses| stable|
 dogs| yard| pigeons| roof| flies| wall| fire|
 flaming| hearth| became| meat| frizzling| cook|
 going| pull| hair| boy| forgotten| let| wind|
 trees| castle| leaf| moved| grow| hedge| thorns|
 grew| seen| flag| story| briar-rose| named|
 country| time| sons| tried| get| found| hands|
 youths| caught| died| heard| man| thorn-hedge|
 stand| grandfather| kings| youth| see| dissuade|
 listen| words| passed| awake| son| parted| accord|
 pass| closed| hounds| lying| wings| house| kitchen|
 holding| hand| seize| maid| hen| pluck| throne|
 breath| opened| sleeping| turn| eyes| stooped|
 kiss| kissed| awoke| astonishment| shook| jumped|
 wagged| tails| pulled| heads| flew| flickered|
 cooked| joint| sizzle| box| ear| screamed|

```



## Appendix D: Grammar – Read A Book (5-Word Sequence)

---

plucking| fowl| marriage| celebrated| lived|  
contented| days;

## APPENDIX E: GRAMMAR – WORD SEQUENCE (SOLAR SYSETM)

```

/* 10-word word-sequence grammar
 Allows 1 to 10 words sequence
*/

grammar wordSequence_solar;

public <s> = <word>
 | <word> <word>
 | <word> <word> <word>
 | <word> <word> <word> <word>
 | <word> <word><word> <word><word>
 | <word> <word><word> <word><word> <word>
 | <word> <word><word> <word><word> <word><word>
 | <word> <word><word> <word><word> <word><word>
 | <word> <word><word> <word><word> <word><word>
 | <word> <word>
 | <word> <word><word> <word><word> <word><word>
 | <word> <word> <word>
 | <simple>;

<word> = <intransvb>
 | <transvb>
 | <cnoun>
 | <pnoun>
 | <det>
 | <adj>
 | <linkingvb>
 | <quest>
 | <other>;

<intransvb>
 = spin | spins | spun | exist | exists | existed;

<transvb>
 = orbit | orbits | orbited | discover | discovers
 | discovered | find | finds | found;

<cnoun>
 = people | planet | moon;

<pnoun>
 = bernard | bond | cassini | dollfus | fountain
 | galileo | hall | herschel | huygens | kowal

```

## Appendix E: Grammar – Word sequence (Solar System)

---

| kuiper | arsen | lassell | melotte | nicholson  
| perrine | pickering | earth | jupiter | mars  
| mercury | neptune | pluto | saturn | uranus  
| venus | almathea | ariel | callisto | charon  
| deimos | dione | enceladus | europa | ganymede  
| hyperion | iapetus | io | janus | jupitereighth  
| jupitereleventh | jupiterfourteenth | jupiterninth  
| jupiterseventh | jupitersixth | jupitertenth  
| jupiterthirteenth | jupitertwelfth | luna | mimas  
| miranda | nereid | oberon | phobos | phoebe | rhea  
| saturnfirst | tethys | titan | titania | triton  
| umbriel;

<det> = a | an | every | one | two | three | four;

<adj> = red | atmospheric;

<linkingvb>  
= is | was | are | were;

<quest>  
= did | do | does;

<simple>  
= hello | hi, there | how are you | good, thanks  
| fine, thanks | have a good day | goodbye | bye-bye ;

<other> = and | or | by;

## APPENDIX F: EXAMPLE DATABASE CONNECTIONS

The following is the source code in Java programming language for example database connections.

```
/* connection to Access Database */
public static Connection getAccessConnection() throws
Exception {
 String url = "jdbc:odbc:solar_system";
 String username = "Administrator";
 String password = "123";

 //Class.forName(driver);
 DriverManager.registerDriver (new
 sun.jdbc.odbc.JdbcOdbcDriver());
 Connection conn = DriverManager.getConnection(url,
 username, password);
 return conn;
}

/* connection to Oracle Database */
public static Connection getOracleConnection() throws
Exception {
 String driver = "oracle.jdbc.driver.OracleDriver";
 String url =
"jdbc:oracle:thin:@localhost:1521:solar_system";
 String username = "mp";
 String password = "mp2";

 Class.forName(driver); // load Oracle driver
```

```
 Connection conn = DriverManager.getConnection(url,
 username, password);
 return conn;
 }

 /* connection to MySQL Database */
 public static Connection getMySQLConnection() throws
 Exception {
 String driver = "org.gjt.mm.mysql.Driver";
 String url = "jdbc:mysql://localhost/solar_system";
 String username = "oost";
 String password = "oost";

 Class.forName(driver);
 Connection conn = DriverManager.getConnection(url,
 username, password);
 return conn;
 }
```

## APPENDIX G: AUTOMATICALLY GENERATED CFG GRAMMAR FOR THE SOLAR SYSTEM

```

/* CFG_new.jsgf */
grammar CFG_new ;

public <s> = <linkingvb> <termph> [<transvb> by] <termph>
 | <linkingvb> <termph> <termph>
 | <quest> <sent>
 | (who | what) <verbph>
 | (which | how many) <nouncla><verbph>
 | <greetings>;

<sent> = <termph> <verbph>;

<termph> = <stermph>
 | <stermph> (and | or) <stermph>;

<stermph> = <pnoun> | <detph>;

<verbph> = <transvbph> | <intransvb>;

<transvbph> = (<transvb> | <linkingvb> <transvb> by)
 <termph>;

<detph>= <det> <nouncla>;

<nouncla> = <adj> <cnoun> | <cnoun>;

<pnoun> = bernard | bond | cassini | dollfus | fountain |
 galileo | hall | herschel | huygens | kowal |
 kuiper | larsen | lassell | melotte | nicholson |
 perrine | pickering | earth | jupiter | mars |
 mercury | neptune | pluto | saturn | uranus |
 venus | almathea | ariel | callisto | charon |
 deimos | dione | enceladus | europa | ganymede |
 hyperion | iapetus | io | janus | jupitereighth |
 jupitereleventh | jupiterfourteenth | jupiterninth
 | jupiterseventh | jupitersixth | jupitertenth |
 jupiterthirteenth | jupitertwelfth | luna | mimas

```

```
 | miranda | nereid | oberon | phobos | phoebe |
 rhea | saturnfirst | tethys | titan | titania |
 triton | umbriel ;
<cnoun> = people | planet | moon ;
<transvb> = orbit | orbits | orbited | discover | discovers
 | discovered | find | finds | found ;
<intransvb> = spin | spins | spun | exist | exists | existed ;
<det> = a | an | every | one | two | three | four ;
<adj> = red | blue ;
<linkingvb> = is | was | are | were ;
<quest> = did | do | does ;
<greetings> = hello | hi there | how are you | fine, thanks
 | goodbye | bye- bye ;
```

## APPENDIX H: AUTOMATICALLY GENERATED SCG GRAMMAR FOR THE SOLAR SYSTEM

```

/* SCG_new.jsgf */
/* Assuming we have n semantic types, we use type_k (1<=k<=n)
to denote each semantic type.
Also, in the SCG grammar template, we just use type_i to
list all the semantic types type_1, type_2, ..., type_n.
meanwhile, we use type_k to specify some specific type */

grammar scg_new ;

public <s> = <linkingvb> <termph_verbph>
 | <quest> <sent>
 | is <pnoun> <pnoun>
 | is <pnoun> (a|an) <nouncla>
 | is <pnoun> (a|an) <nouncla> or (a|an) <nouncla>
 | (who) <verbph_type_1>
 | (what) <verbph_type_2>
 | (what) <verbph_type_3>
 | (which) <nouncla_verbph_type_1>
 | (which) <nouncla_verbph_type_2>
 | (which) <nouncla_verbph_type_3>
 | (how many) <nouncla_verbph_type_1>
 | (how many) <nouncla_verbph_type_2>
 | (how many) <nouncla_verbph_type_3>
 | <greetings>;

<termph_verbph> =
 <termph_type_2> <transvb_type_1> by
 <termph_type_1>

```



```

 | <termph_type_3> <transvb_type_1> by
 <termph_type_1>
 | <termph_type_2> <transvb_type_3> by
 <termph_type_3>;

<sent> =
 <termph_type_1> <verbph_type_1>
 | <termph_type_2> <verbph_type_2>
 | <termph_type_3> <verbph_type_3>;

<termph_type_1> = <stermph_type_1> | <stermph_type_1>
 (and|or) <stermph_type_1>;
<termph_type_2> = <stermph_type_2> | <stermph_type_2>
 (and|or) <stermph_type_2>;
<termph_type_3> = <stermph_type_3> | <stermph_type_3>
 (and|or) <stermph_type_3>;

<stermph_type_1> = <pnoun_type_1> | <detph_type_1>;
<stermph_type_2> = <pnoun_type_2> | <detph_type_2>;
<stermph_type_3> = <pnoun_type_3> | <detph_type_3>;
<detph_type_1> = <det> <nouncla_type_1>;
<detph_type_2> = <det> <nouncla_type_2>;
<detph_type_3> = <det> <nouncla_type_3>;

<nouncla> =
 <nouncla_type_1>
 | <nouncla_type_2>
 | <nouncla_type_3>;

<nouncla_type_1> = <cnoun_type_1> ;
<nouncla_type_2> = <cnoun_type_2>
 | <adj_type_2> <cnoun_type_2>;
<nouncla_type_3> = <cnoun_type_3>
 | <adj_type_3> <cnoun_type_3>;

<verbph_type_1> = <transvb_type_1>
 (<termph_type_2> | <termph_type_3>);
<verbph_type_2> = <intransvb_type_2>;

```

```

<verbph_type_3> = <transvb_type_3> <termph_type_2>
 | <intransvb_type_3>;
<nouncla_verbph_type_1> =
 <nouncla_type_1> <verbph_type_1>
 | <nouncla_type_2> <verbph_passive_type_1>
 | <nouncla_type_3> <verbph_passive_type_1>;
<nouncla_verbph_type_2> = <nouncla_type_2> <verbph_type_2>;
<nouncla_verbph_type_3> =
 <nouncla_type_3> <verbph_type_3>
 | <nouncla_type_2> <verbph_passive_type_3>;
<verbph_passive_type_1> =
 <linkingvb> <transvb_type_1> [by
 <termph_type_1>];
<verbph_passive_type_3> =
 <linkingvb> <transvb_type_3> [by
 <termph_type_3>];
<pnoun> = <pnoun_type_1>
 | <pnoun_type_2>
 | <pnoun_type_3> ;
<pnoun_type_1> = bernard | bond | cassini | dollfus |
 fountain | galileo | hall | herschel | huygens |
 kowal | kuiper | larsen | lassell | melotte |
 nicholson | perrine | pickering ;
<cnoun_type_1> = people;
<transvb_type_1> = discover | discovered | discovers | find
 | finds | found;
<pnoun_type_2> = earth | jupiter | mars | mercury | neptune
 | pluto | saturn | uranus | venus ;
<cnoun_type_2> = planet;
<intransvb_type_2> = spin | spins | spun | exist | existed |
 exists;
<adj_type_2> = red | blue;

```

```
<pnoun_type_3> = almathea | ariel | callisto | charon |
 deimos | dione | enceladus | europa | ganymede |
 hyperion | iapetus | io | janus | jupitereighth |
 jupitereleventh | jupiterfourteenth | jupiterninth
 | jupiterseventh | jupitersixth | jupitertenth
 | jupiterthirteenth | jupitertwelfth | luna |
 mimas | miranda | nereid | oberon | phobos |
 phoebe | rhea | saturnfirst | tethys | titan |
 titania | triton | umbriel ;

<cnoun_type_3> = moon ;

<transvb_type_3> = orbit | orbits | orbited ;

<intransvb_type_3> = spin | spins | spun | exist | existed |
 exists ;

<adj_type_3> = red | blue ;

<det> = a | an | one | two | three | four | the ;

<linkingvb> = is | was | are | were ;

<quest> = did | do | does ;

<greetings> = hello | hi, there | how are you | good, thanks
 | fine, thanks | have a good day | goodbye | bye-
 bye ;
```

## APPENDIX I: THE XML FILE FOR SPEECH APPLICATION

### Read-A-Book

The URL for the XML files of the speech application Read-A-Book is at:

[http://cs.uwindsor.ca/~speechweb/p\\_d\\_speechweb/read\\_a\\_book/read\\_a\\_book.xml](http://cs.uwindsor.ca/~speechweb/p_d_speechweb/read_a_book/read_a_book.xml).

Note that this file can only be downloaded using a browser that can execute X+V files.

The source code of the XML file for speech application Read-A-Book is as follows:

```
<?xml version="1.0"?>
<!DOCTYPE html PUBLIC "-//VoiceXML Forum//DTD XHTML+Voice
1.2//EN"
"http://www.voicexml.org/specs/multimodal/x+v/12/dtd/xhtml+voice1
2.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
 xmlns:vxml="http://www.w3.org/2001/vxml"
 xmlns:ev="http://www.w3.org/2001/xml-events"
 xmlns:xv="http://www.voicexml.org/2002/xhtml+voice"
 xml:lang="en-US">

<!--

Date: March. 2006
Developer: Ma, Xiaoli(William)
Architecture: LRRP (Dr.Frost, University of Windsor, Canada)

*****-->
```

```

<head>

<!-- Please modify the value of variable sv_greeting, and cgiLink
to fit your application. -->
<script type="text/javascript">
/** The greeting message that will say to the user, only at the
first time the user visits this page. */
var sv_greeting ="hello, I can read a book to you.";
/** The link to your CGI interpreter location.
Notice, you have to place the CGI interpreter program with this
page in the same domain to prevent a cross-domain security
error.**/
var
cgiLink="http://cs.uwindsor.ca/~speechweb/p_d_speechweb/read_a_bo
ok/read_a_book.cgi";
</script>

<!-- VoiceXML form. -->
<vxml:form id="vxml_form">
<vxml:field name="st_field" xv:id="voice_input" modal="true">
 <!-- NOTICE!!! PLEASE MODIFY THE VALUE OF 'src' ATTRIBUTE IN
THE NEXT LINE <grammar> ELEMENT TO YOUR GRAMMAR FILE LOCATION.-->
 <vxml:grammar type="application/x-jsgf"
src="http://cs.uwindsor.ca/~speechweb/p_d_speechweb/read_a_book/r
ead_a_book.jsgf" />
 <!-- The following greeting will only speak out when user
connects to a new interpreter. -->
 <vxml:prompt cond="sayGreetings==true"><vxml:break
time="500ms"/><vxml:value expr="sv_greeting"/>
 <vxml:value expr="updateShowFrame('GREETING:
'+sv_greeting);"/></vxml:prompt>
 <vxml:filled>
 <!--*****

```

This "filled" element will be run after user speech input has recognized.

Inside this element, first step, i have assign the user input to the variable 'question',

because VoiceXML code can access a JavaScript defined variable, but JavaScript can not see a VoiceXML defined variable.

Then, in the next step, i call a JavaScript function "runCode()" to proceed AJAX submit process.

\*\*\*\*\*-->

```
<vxml:assign name="question" expr="st_field"/>
```

```
<!--*****
```

Calls to javascript mainControl() function to do the logical process based on user voice input.

\*\*\*\*\*-->

```
<vxml:assign name="javacode" expr="mainControl();"/>
```

```
<vxml:prompt><vxml:break time="300ms"/><vxml:value
expr="answer"/></vxml:prompt>
```

```
<!-- If the answer is not a link to next interpreter, then repeat
the voice dialog. -->
```

```
<vxml:if cond="isLink==false">
```

```
<vxml:throw event="repeat.st_field"/>
```

```
</vxml:if>
```

```
</vxml:filled>
```

```
<vxml:catch event="nomatch noinput">
```

```
<vxml:prompt>Sorry, I don't understand, can you say it
again?</vxml:prompt>
```

```
<vxml:reprompt/>
```

```
</vxml:catch>
```

```
<vxml:catch event="help">
```

```
No help is available! Restart the dialog!
```

```
<vxml:clear namelist="st_field"/>
```

```
<vxml:reprompt/>
```

```
</vxml:catch>
```

```
</vxml:field>
```

```

<!-- Catch the 'repeat.st_field' event. -->
<vxml:catch event="repeat.st_field">
 <vxml:clear namelist="st_field"/>
<!-- Restart the voice form without change the speech grammar. --
>
 <vxml:reprompt/>
</vxml:catch>
</vxml:form>

<script type="text/javascript">
/***** Declare global variables shared by JavaScript and
VoiceXML *****/
var sayGreetings=true;
var defaultGreetingMsg="Hi, i'm ready to talk now.";
/** The location of next remote speech-application/CGI-
application interpreter. */
var nextPage="";
/** Question query recognized from user's speech (request). */
var question="";
/** Answer query returned from remote CGI interpreter (response).
***/
var answer="";
var answerRecieved=false;
/** Answer query contains a link to next CGI interpreter. */
var isLink=false;
var gotoNext= false;
/** This variable needed for VXML to call JavaScript code. */
var javacode="";
/** menu page of the demo public-domain speechweb. */
var
startPage="http://luna.cs.uwindsor.ca/~speechweb/p_d_speechweb/me
nu/demo_menu.xml";

```

```

if(sv_greeting=="")
 sv_greeting=defaultGreetingMsg;

/*****

This is the main control function to the whole question submit
and answer retrived procedures.
It will call submitReq() method to send the question to the CGI
program.
then it will check the answer whether it is a link to new CGI
program or a simply answer string.
if it is a link to another interpreter, then retrieve the data
from there,
and call the 'changeData' function to change the neccessary
information for the next round dialog.
*****/
*****/
function mainControl()
{
 updateShowFrame("QUESTION: "+question+"
");

 answer="";
 answerRecieved=false;
 isLink=false;
 sayGreetings=false;

 /* call submitReq() method to send the question to the CGI
program. */
 submitReq("POST", cgiLink);
 /** Cannot receive data from CGI interpreter. Network problem.
**/
 if(answerRecieved==false)

```



```

 return "-1";
 answer = getAnswer(xmlhttp.responseText);

 /***** Check whether the received answer is a link or not.
 And, assign the result to the global variable isLink. *****/
 checkAnswer(xmlhttp.responseText);

 /***** if the answer is not a link, then show the answer to
 the user and return. *****/
 if(!isLink)
 {
 gotoNext=false;
 updateShowFrame("RESPONSE: "+answer+"
");
 return "1";
 }

 nextPage=getNextInterpreter(xmlhttp.responseText);

 updateShowFrame("RESPONSE: "+answer+"

");

 if(gotoNext==true)
 window.location=nextPage;

 return "1";
}

/*****

This function returns the substring that has to be spoken as a
result of the user's question.
Same procedure is applied for extracting the content to be spoken
out.

```

```

*****/
function getAnswer(answer)
{
 var ex=answer;
 var index;
 if((ex.indexOf('LINK=',0)) == -1)
 return ex;
 ex= ex.slice(5);
 index = ex.indexOf(";",0);
 ex = ex.substring(0,index);
 return ex;
}

/*****

This function uses AJAX, it will submit the question to the given
URI if it use a 'POST' method.
Or, it will retrieve data from the given URI if it use a 'GET'
method.

*****/

function submitReq(method, url)
{
 /***** Initialize AJAX XMLHttpRequest object. *****/
 xmlhttp=new XMLHttpRequest();
 /*****

 Assign a event listener to the 'onreadystatechange' event.
 Different listerner assigned depends on a 'GET' or a 'POST'
method.
 *****/

 if(method=="GET")
 xmlhttp.onreadystatechange=stateChange_GET;
```

```
else
 xmlhttp.onreadystatechange=stateChange_POST;

 /** Check whether the url involves a cross-domain security
error before send the request. */
 if(isCrossDomain(url)==true)
 {
 /** if method is 'GET', it means this function is called from
loadPage() function to validate a user input URL. */
 if(method=="GET")
 alert("Cannot validate input URL since it involves a cross-
domain security issue. Load URL immediately.");
 /**
 if method is not 'GET', which means 'POST' method,
 it means this method is called from main control to submit a
question query to the interpreter.
 */
 else
 updateShowFrame("SYSTEM ERROR: An error which against the
web browser cross-domain security issue."
 +" Your CGI interpreter has to be placed in
the same domain with this voice page."+
 "Please contact to your application
provider to fix this problem. \n"
 +"Your CGI interpreter location: "+ url+"
Current voice page host domain: "+window.location.host);
 answerRecieved=true;
 answer="An error which against the web browser cross-domain
security issue has occured. Please check the error message to
continue.";
 return;
 }
}
```

```
/** Open the connect, sychronized. ***/
xmlhttp.open(method,url,false);

if(method=="GET")
 xmlhttp.send();
else
{
 xmlhttp.setRequestHeader("Content-Type", "application/x-www-
form-urlencoded");
 xmlhttp.send("question="+question);
}
}

function stateChange_POST()
{
 /***** if xmlhttp shows loaded *****/
 if (xmlhttp.readyState==4)
 {
 if (xmlhttp.status==200 || xmlhttp.status==304)
 {
 answerRecieved = true;
 }
 else
 {
 answerRecieved = false;
 xmlhttp.responseText = "";
 }
 }
}

function stateChange_GET()
{
 /***** if xmlhttp shows loaded *****/
 if (xmlhttp.readyState==4)
```

```

{
 if (xmlhttp.status==200 || xmlhttp.status==304)
 {
 answerRecieved = true;
 }
 else
 {
 answerRecieved = false;
 xmlhttp.responseText = "";
 }
}
}

/** Validate the given url with the current page domain(hostname),
to see whether they are in the same domain or cross-
domain(different domain). */
function isCrossDomain(url)
{
 var domain = url;
 var i = domain.indexOf("//");
 if(i==-1)
 return false;

 domain = domain.slice(i+2);

 var k = domain.indexOf("/");
 if(k!=-1)
 domain = domain.slice(0, k);
 else
 return true;

 var host = window.location.hostname;

 if(host==domain)

```

```

 return false;
else
 return true;

}

/*****

Check whether there is a occurrence of '=' character in the
answer, which means a link existed in it.
And, assign the result to the global variable 'isLink'.

*****/

function checkAnswer(answer)
{
 if((answer.indexOf('LINK=',0))== -1)
 isLink=false;
 else
 isLink=true;

 return isLink;
}

/*****

if the answer is a link, this function will return the next
interpreter's URI as a string. Otherwise, return "-1".

*****/

function getNextInterpreter(answer)
{
 var loc;
 var ex = answer;
 var index;

```

```

/*****
Check if the answer is a link to next speech application
interpreter.
It should never be evaluated as true, otherwise error.
*****/

 if(!isLink)
 return "-1";

/*****

If the answer is a link, then its formation should be:
"LINK=_answer;SIHLO=_location;".
e.g. Question send to judy.cgi: "can i talk to solar man".
 Answer received from judy.cgi: "LINK=yes. here he
is;SIHLO=http://luna.cs.uwindsor.ca/~speechweb/p_d_speechweb/judy
/judy.xml"

*****/

/***** extracts the LINK= substring from the string and assigns
it to the variable ex *****/
 ex = ex.slice(5);
/***** gets the index position of ';' *****/
 index = ex.indexOf(";",0);
 index = index+1;

/*****

The string after the '=' and upto ';' are eliminated because this
is the content which is the answer-query of the user's input.

*****/

 loc = ex.substr(index);
/*****


```

eliminating 'SIHLO=' from the loc variable.

SIHLO contains the server address starting right after '=' and ended by the delimiter ';'.

```

```

```
*****/
```

```
 ex = loc.slice(6);
 index = ex.indexOf(";",0);
 loc = ex.substring(0,index);

 return loc;
}
```

```
/** Update the text area in the HTML and show message on it. */
```

```
function updateShowFrame(message)
```

```
{
 var objTable = document.getElementById("logFrame");

 objTable.insertRow(0);
 objTable.rows[0].insertCell(0);
 objTable.rows[0].insertCell(1);
 var cell0 = objTable.rows[0].cells[0];

 var cell1 = objTable.rows[0].cells[1];
 cell1.align="left";
 cell0.align="left";
 cell0.width="105";
 if(message.indexOf("SYSTEM ERROR: ")!=-1)
 {
 var objFont = document.createElement("font");
 objFont.color="red";
 objFont.size="-1";
 objFont.appendChild(document.createElement("b"));
 objFont.firstChild.innerHTML = message.slice(0,
message.indexOf(":"+1);
```



```

cell0.appendChild(objFont);

var objFont2 = document.createElement("font");
var objIta = document.createElement("i");
objFont2.color="black";
objFont2.size="-1";
objFont2.appendChild(objIta);
cell11.appendChild(objFont2);
objIta.innerHTML=message.slice(message.indexOf(":")+1);
}else
{
 var index = message.indexOf(":");
 var ex=message.slice(0,index+1);
 var objFont = document.createElement("font");
 if(ex.indexOf("QUESTION:")!=-1)
 objFont.color= "blue";
 else if(ex.indexOf("RESPONSE:")!=-1)
 objFont.color="green";
 else
 objFont.color="purple";

 objFont.appendChild(document.createElement("b"));
 cell0.appendChild(objFont);
 objFont.firstChild.innerHTML = ex;
 cell11.appendChild(document.createElement("font"));
 cell11.firstChild.innerHTML = message.slice(index+1);
}

/***** Insert a table row as an empty line after a response
and greeting message. *****/
if(message.indexOf("QUESTION")==-1)
{
 objTable.insertRow(0);
 objTable.rows[0].insertCell(0);
 objTable.rows[0].colspan="2";

```

```
 objTable.rows[0].cells[0].innerHTML = "
 ";
 }

 return "";
}

/** Load user's application. */
function loadPage(checkInput)
{
 /** Get user's input. */
 var loc = document.getElementById("id_nextPage").value;

 /** if user's input is empty, then return a error message. */
 if(loc=="")
 {
 updateShowFrame("SYSTEM ERROR: Please input the URL to your
voice page in the above text field. It can not be empty!");
 }
 /** if user input is not empty, and user asked to validate URL
before go. */
 else if(checkInput==true)
 {
 submitReq("GET", loc);
 /** if the valicating process return a false as result, which
means invalid URL. */
 if(answerRecieved==false)
 {
 if(xmlhttp.status==404)
 updateShowFrame("SYSTEM ERROR: Unable to load your voice
page. File does not exist at: "+ loc);
 else
 updateShowFrame("SYSTEM ERROR: Unable to load your voice
page. Network problem, error code: "
```

```
+xmlhttp.status+". Please check your internet
connection.");
}
/** if user's input is not empty, it is a valid URL to next
page. */
else
 window.location=loc;
}
/** if user's input is not empty, and user asked to load URL
page immediately. */
else
 window.location=loc;
}

function menuPage()
{
 window.location=startPage;
}

function processQuestion()
{
 gotoNext = true;
 question = document.getElementById("id_questionField").value;
 document.getElementById("id_questionField").value="";
 mainControl();
}

</script>
```

<!--\*\*\*\*\*

The following script will only be run after a 'vxmlDone' event is thrown after the VoiceXML form finish all its process.

It also means that the answer returned from interpreter contains a link to next interpreter, so it needs to go there.

```

*****_>

<script type="text/javascript" id="gotoNextPage"
declare="declare">
 window.location=nextPage;
</script>

<title>Public-Domain SpeechWeb</title>
</head>
<body id="page.body">
<center><h2>Welcome to our new voice browser!</h2></center>

<center>
<table>
<tr><td colspan="6">Load your own speechapplication :
<input type="text" id="id_nextPage" size="50"
value="http://luna.cs.uwindsor.ca/~speechweb/p_d_speechweb/menu/d
emo_menu.xml"/>

</td></tr>
<!-- Call loadPage() function to setup the interpreter and speech-
grammar location according to the above input text field value; -
->
<tr><td colspan="2"><input type="button" name="submitValidate"
value="Validate Before Go" onclick="loadPage(true)"/></td>
<td colspan="2"><input type="button" name="submitGo" value="Go
Immediately" onclick="loadPage(false)"/></td>
<td colspan="2"><input type="button" name="menuGo"
value="SpeechWeb Menu Page" onclick="menuPage()"/></td></tr>
<tr><td colspan="6">

Say your question or type
it in here:</td></tr>
<tr><td colspan="6"><form onsubmit="processQuestion(); return
false;">
<input type="text" size="70" name="questionField"
id="id_questionField" value=""/></form></td></tr>
</table>

```

```


<table id="logFrame" width="600"></table>

</center>
</body>
<!-- Call a script to reload the vxml form when the current vxml
form has done its process. -->
<ev:listener ev:observer="page.body" ev:event="vxmldone"
ev:handler="#gotoNextPage" ev:propagate="stop" />
<!-- Load 'vxml_form' when the page.body loaded. -->
<ev:listener ev:observer="page.body" ev:event="load"
ev:handler="#vxml_form" ev:propagate="stop" />
</html>
```

## APPENDIX J: AN EXCERPT OF THE INTERPRETER FOR SPEECH APPLICATION Read-A-Book

The interpreter of the speech application Read-A-Book is written in Miranda, the non-strict purely functional programming language. Parts of the source code are as following:

```
|| get the tail string from k to the end of the list,
|| k starts from 0.
tailstr list k = list!k : tailstr list (k+1), if k < #list
 = [], otherwise

|| get the first k characters from the string,
|| k starts from 1.
headstr (a: as) k = a: headstr as (k-1),
 if k>0 & k < #(a:as)
 = a:as, if k >= #(a:as)
 = [], otherwise

|| read page k, page number starts from 1
|| The component function "numToLetter n numLetter " is
 defined elsewhere.
|| It changes a number to a letter to append to a string.
readpage 0 = "The page number is out of range."
readpage n = "starting page: " ++ numToLetter n numLetter
 ++ ". " ++ pages!(n-1) ++ " That's the end of
 page " ++ numToLetter n numLetter,
 if n< #pages
= pages!(n-1) ++ " Congratulations! You have
 reached the end of this book.", if n= #pages
```

```

= "This book has " ++ numToLetter (#pages)
 numLetter ++ " pages. Please reinput your
 command.", otherwise

|| continuously read k pages
readKpages (a:as) k = a ++ " " ++ readKpages as (k-1),
 if k < #(a:as) & k>0
 = foldr (++) [] (a:as), if k >= #(a:as)
 = [], otherwise
|| read from page i and continuously read k pages
readFrom i k = "starting page: " ++ numToLetter i numLetter
 ++ ". " ++ (readKpages (tailstr pages (i-1))
 k) ++ " This is the end of page " ++
 numToLetter (i+k-1) numLetter ,
 if (i<= #pages & i>0 & i+k-1 < #pages)
 = readKpages (tailstr pages (i-1)) k ++ "
 Congratulations! You have reached the end of
 this book." ,
 if (i<= #pages & i>0 & i+k-1 >= #pages)
 = "The starting page number exceeds the maximum
 number of the book! " ++
 "Please reinput your command.", otherwise

```

## APPENDIX K: THE CGI FILE FOR SPEECH APPLICATION

### Read-A-Book

The URL for the CGI file of the speech application Read-A-Book is:

[http://cs.uwindsor.ca/~speechweb/p\\_d\\_speechweb/read\\_a\\_book/read\\_a\\_book.c  
gi](http://cs.uwindsor.ca/~speechweb/p_d_speechweb/read_a_book/read_a_book.cgi)

The content is as follows:

```
#!/bin/csh -f

setenv HOME '/stul/shic/public_html/read_a_book:$HOME'
setenv PATH '/lapps1/mira:$PATH'

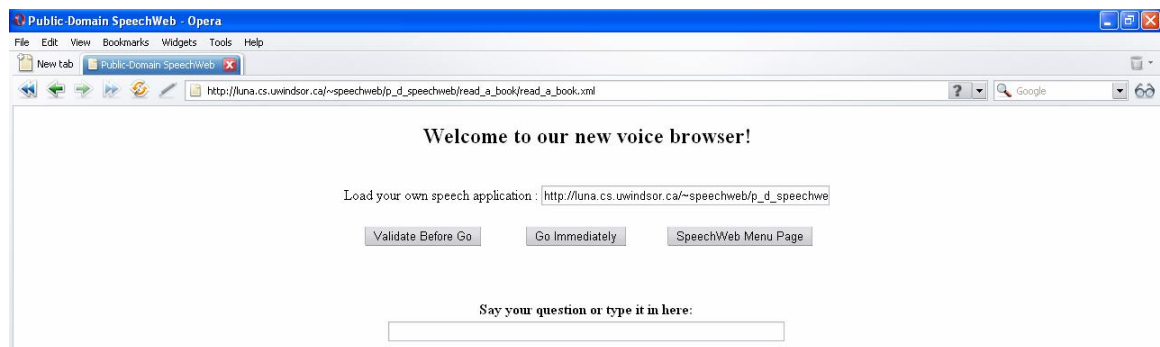
echo "Content-Type:text/plain"
echo ''
setenv v "`/bin/cat`"
/lapps1/mira/bin/mira -heap 10000000
/stul/shic/public_html/read_a_book/read_a_book.m << zzz
(sh_answer "$v")
/q
zzz
```



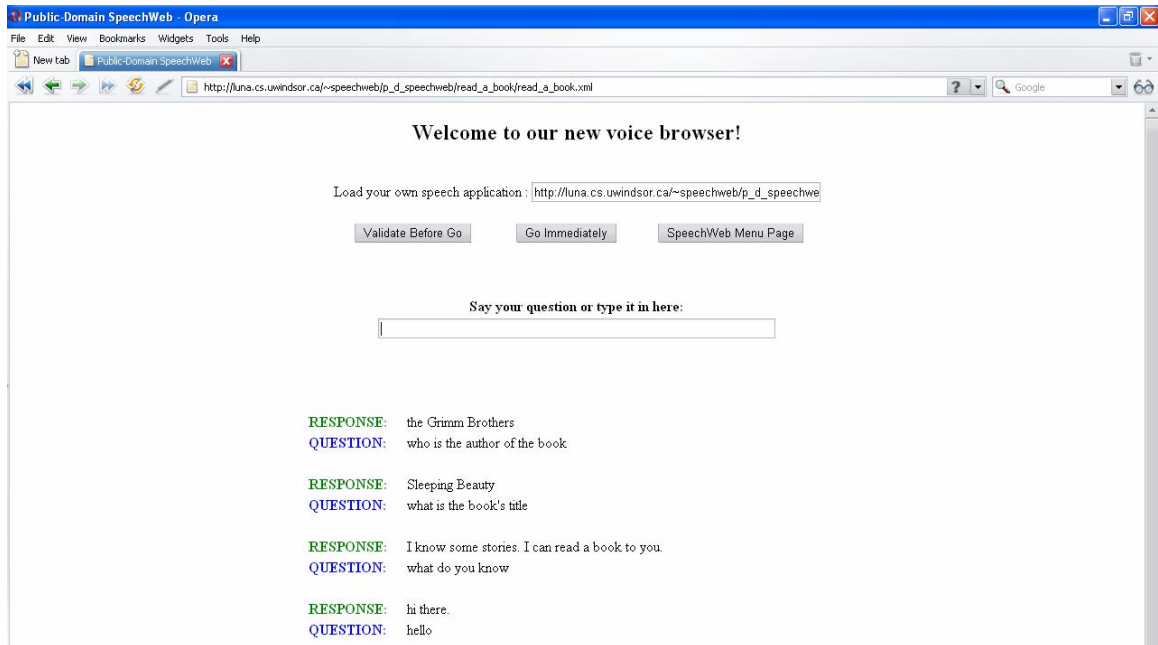
## APPENDIX L: SAMPLE SCREENSHOTS FOR SPEECH APPLICATION Read-A-Book

Note that, by taking the advantages of X+V multi-model, the conversation is available by both voice- and text-input and output. Therefore, we can capture the screenshots of the example conversation between the user and the computer. In the screen, the conversation is recorded in the way of “rolling down”, which means, the first talk is in the bottom, the later response is on the top part of the screen.

The first screenshot on opening the browser to “read a book” application:



Greet the system, and get to know the system:



The user is wondering how to communicate with the SpeechWeb:

**RESPONSE:**

You can ask for help like: help. please help. help please. I need help. please give me a help. You can greet like: hi. hi there. hello. hello there. bye. goodbye. Also, you can ask me to read to you, like: read from the beginning. what is the book's title? who is the book's author? what is the title of the book? who is the author of the book? read page page number. please read page page number. read from page page number. please read from page page number. read chapter chapter number. please read chapter chapter number. read from chapter chapter number. please read from chapter chapter number. what words do you know? which page refers to the word? which pages refer to the word? what page refers to the word? what pages refer to the word? In addition, you can talk to my friends judy, monty, and solar man by asking: can I talk to judy? can I talk to monty? can I talk to solar man? All the others you can say are listed as follows. fine thanks. thanks. yes please. what do you know. tell me a joke know any poems. who do you know? who is judy? who is monty? who is solar man? if I could not understand you, you need to repeat or rephrase your command please.

**QUESTION:** what can I say

Start the reading by page:

<p><b>RESPONSE:</b></p>	<p>starting page: two. But it happened that once when the queen was bathing, a frog crept out of the water on to the land, and said to her, 'Your wish shall be fulfilled, before a year has gone by, you shall have a daughter.' What the frog had said came true, and the queen had a little girl who was so pretty that the king could not contain himself for joy, and ordered a great feast. He invited not only his kindred, friends and acquaintances, but also the wise women, in order that they might be kind and well-disposed towards the child. There were thirteen of them in his kingdom, but, as he had only twelve golden plates for them to eat out of, one of them had to be left at home. The feast was held with all manner of splendor and when it came to an end the wise women bestowed their magic gifts upon the baby - one gave virtue, another beauty, a third riches, and so on with everything in the world that one can wish for. When eleven of them had made their promises, suddenly the thirteenth came in. She wished to avenge herself for not having been invited, and without greeting, or even looking at anyone, she cried with a loud voice, The king's daughter shall in her fifteenth year prick herself with a spindle, and fall down dead. And, without saying a word more, she turned round and left the room. This is the end of page six.</p>
<p><b>QUESTION:</b></p>	<p>please read from page two</p>
<p><b>RESPONSE:</b></p>	<p>starting page: one. A long time ago there were a king and queen who said every day, 'Ah, if only we had a child.' But they never had one. That's the end of page one.</p>
<p><b>QUESTION:</b></p>	<p>please read page one</p>

Continue the reading by chapter:

<p><b>RESPONSE:</b></p>	<p>starting page: three. What the frog had said came true, and the queen had a little girl who was so pretty that the king could not contain himself for joy, and ordered a great feast. That's the end of page three. starting page: four. He invited not only his kindred, friends and acquaintances, but also the wise women, in order that they might be kind and well-disposed towards the child. There were thirteen of them in his kingdom, but, as he had only twelve golden plates for them to eat out of, one of them had to be left at home. That's the end of page four. this is the end of page four, and the end of chapter two.</p>
<p><b>QUESTION:</b></p>	<p>please read chapter two</p>

Specifying the reading by specific pages which covering interested words:

<b>RESPONSE:</b>	starting page: nine. It happened that on the very day when she was fifteen years old, the king and queen were not at home, and the maiden was left in the palace quite alone. That's the end of page nine.
<b>QUESTION:</b>	please read page nine
<b>RESPONSE:</b>	page: one, two, three, nine, fourteen, twenty, twenty six, twenty nine, refer to the word: queen
<b>QUESTION:</b>	which pages refer to the word queen
<b>RESPONSE:</b>	I know: king, queen, child, frog, daughter, joy, feast, women, bestowed, gifts, avenge, spindle, sleep, princess, burnt, fifteenth, maiden, tower, spinning, flax, decree, prick, finger, bed, palace, castle, thorns, briar-rose, son, kiss, marriage
<b>QUESTION:</b>	what words do you know

Leaving the Read-A-Book application, and move to judy :

**Say your question or type it in here:**

The application confirming that is judy:

**Say your question or type it in here:**

**RESPONSE:** hi there. My name is Judy  
**QUESTION:** hello

Getting to know the hyperlinked speech application judy

**RESPONSE:** the dark clouds, lingered, as he lay. Then, the strong winds, blew them both away.  
**QUESTION:** tell me a poem

**RESPONSE:** I only know one poem just now. My colleague, Dr. Frost is looking for someone to teach me some more poems.  
**QUESTION:** what do you know

Leaving judy to monty:

**Say your question or type it in here:**

Confirming monty:

**Say your question or type it in here:**

**RESPONSE:** hi there. My name is Montagew, but you can call me Monty  
**QUESTION:** hello

Getting to know monty:

**RESPONSE:**

Did you here about the two professors? They were walking through the forest, when they saw some tracks. Moose tracks, yelled one of them. Deer tracks, yelled the other. Moose tracks. Deer tracks. They argued for an hour. Then the train hit them.

**QUESTION:**

tell me a joke

**RESPONSE:**

I got a joke or two. Not much else.

**QUESTION:**

what do you know

Leaving monty , moving to solar man :

**Say your question or type it in here:**

can I talk to solar man

Confirming solar man:

**Say your question or type it in here:**

**RESPONSE:**

hello. My name is solar man.

**QUESTION:**

hello

Getting to know solar man:

**RESPONSE:** Hall.

**QUESTION:** who discovered phobos

**RESPONSE:** nobody.

**QUESTION:** who discovered mars

**RESPONSE:** Not much I am afraid. I am just beginning to learn. I know a bit about the planets, the moons, and the people who discovered them. My master will teach me some more when he gets another grant

**QUESTION:** what do you know

Asking questions to solar man and solar man providing answers.

**RESPONSE:** 1

**QUESTION:** how many moons orbit earth

**RESPONSE:** yes.

**QUESTION:** does mercury exist

**RESPONSE:** Phobos, and Deimos.

**QUESTION:** which moons orbit mars

Appreciating solar man:

**RESPONSE:** you are welcome

**QUESTION:** thanks

Leaving the SpeechWeb:

**RESPONSE:** goodbye. See you in the stars.

**QUESTION:** goodbye

## VITA AUCTORIS

**NAME:** Shi, Yue (Sunny)

**DATE OF BIRTH** April 21, 1972

**PLACE OF BIRTH:** Suizhong, Liaoning, China

**EDUCATION:** Southeast University, Nanjing, Jiangsu, China

1989.9 – 1993.7 B.Sc.

University of Windsor, Windsor, Ontario, Canada

2002.1 – 2003.12 M.Sc.

University of Windsor, Windsor, Ontario, Canada

2004.1 – 2010.9 Ph.D.