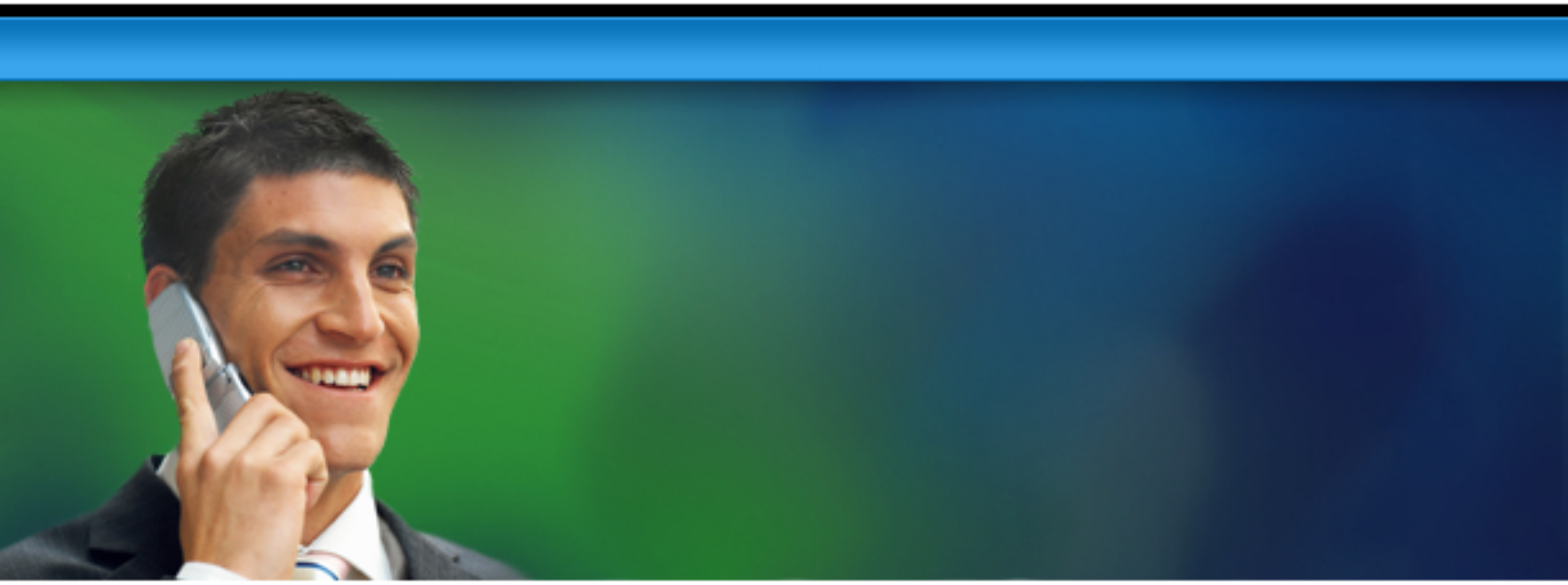
LumenVox
Speech Understood



Grammars

At the heart of speech application development is the grammar, the list that defines which words a speech application can recognize. Grammars will influence — and be influenced by — many other facets of a speech application, including prompts, menu choices, and the overall call flow of the application. In this paper, we explore several strategies to designing grammars to make your grammars more effective, thereby increasing the accuracy and overall success rate of your speech recognition applications.

Key Takeaways

- Keep grammars as small as possible, with "just enough" coverage to serve the majority of your callers.
- Consider grammar design at the same time you design your prompts.
- Test any changes you make with actual call data garnered from a deployed application.

About Grammars

A grammar is a file that contains a list of words that your speech application will recognize. Automatic speech recognition systems that are speaker independent, such as the LumenVox Speech Engine, are grammar based, meaning that they do not recognize any words that are not defined by a grammar.
Most grammars today are written according to the Speech Recognition Grammar Specification (SRGS) format established by the W3C. SRGS includes two equivalent formats, a structured XML format called GrXML and a more human-readable format called Augmented Backus-Naur Form (ABNF).
A grammar written in SRGS defines not just the words that will be recognized, but also the allowable phrases that can be spoken, including single words, short commands, and full sentences.
This paper will not cover the specifics of writing grammars using either format. For that, you may refer to the latest SRGS specification. Instead, this paper explores the fundamental design principles involved in grammar creation.

Provide "Just Enough" Coverage

A common mistake, made especially by designers new to speech recognition, is to try to develop grammars that cover every possible response a user could speak. This attempt at comprehensive coverage, however, causes more problems than it solves.
Generally speaking, recognition accuracy is higher with a smaller grammar. Larger grammars are more likely to include words or phrases that sound similar, and thus are likely to confuse the speech engine. By keeping your grammars small, speech engines are able to better match what speakers say with the words in the grammar.
A well-designed application will give the user a consistent mental model of how it works. Users will have a good idea of what is expected from them and they will understand how to get what they want from the application. In this case, each interaction will produce fairly predictable responses. A very small percentage of callers — around 3 to 5 percent — will provide unpredictable responses.

Because the responses of most callers are predictable, you can get very good coverage of 95 to 98 percent of the interactions with a fairly small grammar. This is an advantage to the developer, since it actually takes less effort to achieve better results. Attempts to cover the remaining small group of outlying callers will result in a very considerable increase in grammar size.

As those extra words are added to cover the small set of callers, overall accuracy tends to drop, since the grammar size has increased significantly. Also, confidence scores — a probability that reflects the likelihood a speaker's utterance matched the grammar — become somewhat less useful with large grammars because of increased confusability. Speech application developers who attempt to cover every possible response hurt the majority of their callers by trying to help a small percentage of callers.

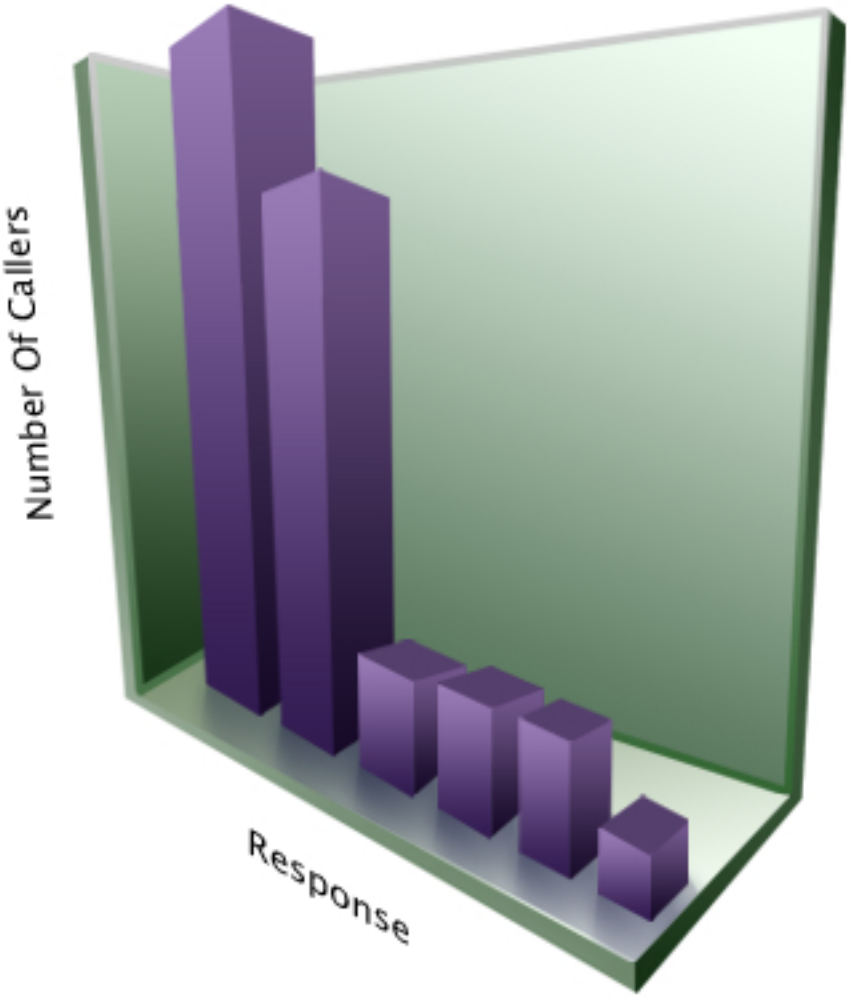


Fig. 1: A directed prompt produces predictable responses from callers. The majority of callers respond with a few phrases. Note there are a large number of responses that are only given by a handful of callers.

Even small changes to grammars to account for very uncommon responses can be detrimental. A change to benefit 1 percent of callers might negatively affect 2 percent of the callers, a net loss in success.

For instance, you might have a prompt that asked callers to say the name of the U.S. state from which they're calling. As a developer, you might be tempted to add a list of the world's countries to the grammars, in case somebody was calling internationally. If the application recognized a foreign country, the application could respond that other countries are not supported. While this might be nice functionality in theory, in practice adding the list of countries increases the chance that a state will be confused for a country. If the majority of your callers are saying states, these misrecognitions will frustrate those callers who follow instructions.

It is better to have a simple grammar, and rely on the confidence score to tell you when a caller has spoken outside the grammar. You can then have a prompt explaining to the caller that the system did not understand what was said. Most callers will then re-phrase their utterance in a more predictable way the second time.

Callers who give wildly unexpected responses, e.g. callers who swear at the system, should never be accommodated in grammars. Consider these responses as simply inappropriate input. It is always counter productive to try and deal with callers who are purposely misusing an application.

In the same way a DTMF (Touch ToneÒ) application would not respond correctly to callers mashing their telephone keypad, a speech application should not be built to handle completely inappropriate input.

Design Grammars and Prompts Together

There is a close relationship between a prompt and its corresponding grammars, and this relationship should be reflected during every phase of the application development. The grammars associated with a prompt determine what responses the speech application can recognize.

A prompt will also determine the sort of responses callers give, and thus shape what a grammar will look like.

Prompts that direct callers to give specific responses tend to produce response distributions like the one seen in **Fig. 1**,

where a majority of the callers give a relatively small number of responses. Open-ended prompts tend to produce distributions like the one seen in **Fig. 2**, where responses are much more spread out.

For this reason, it is usually preferable to create prompts that ask questions that are not very open ended. Not only are predictable responses are easier to handle by your speech applications, but a small cluster of responses being given by a majority of callers also tends to reduce confusability a speech engine.

A good prompt that lives up to its name and prompts users for specific responses will make it much easier to design grammars, since it will be relatively easy to predict the majority of responses. This will make grammars smaller and less complex.

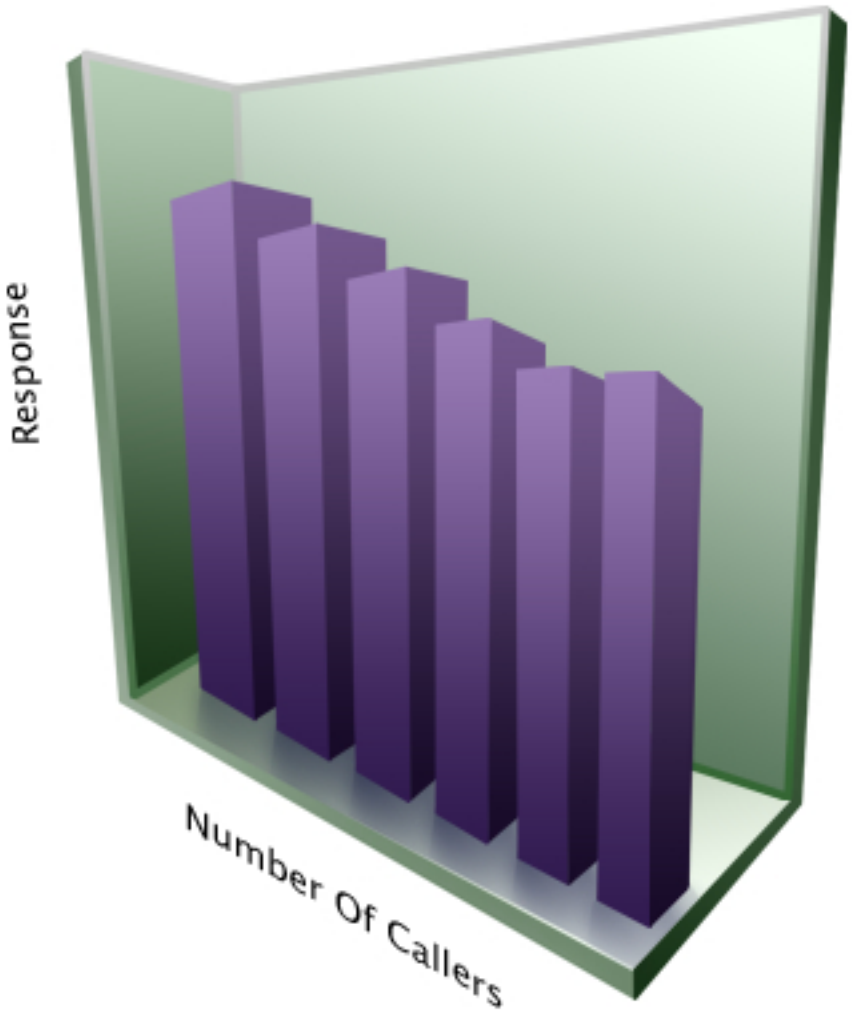


Fig. 2:
Open-ended prompts produce a flat distribution of responses. This variation in responses makes it more difficult to design and tune effective grammars.

Good prompts ask callers a question that has a predictable response. One method to constrain what the caller will say is to provide several responses. A prompt in an interactive voice response system for a pizza restaurant might ask callers a question like "What size pizza would you like? We have small, medium, large, or extra large," placing emphasis on the sizes. If the callers are not provided these examples they may give inappropriate responses such as "16 inches" or "20 inches." Callers are being asked a question and immediately given examples of expected responses, providing them with a clear mental model of what the application expects.

Don't Reinvent the Wheel

Most speech recognition engines, including the LumenVox Speech Engine, include several common built-in grammars. These grammars tend cover domains such as yes/no, digits, natural numbers, dates, and monetary amounts.
Developers should take advantage of these built-in grammars, as they are good examples to follow for grammar creation. The grammars also provide a quick and easy way to recognize utterances needed for many of the most common speech applications.

Build Modular Grammars

One problem that sometimes occurs when developers use built-in grammars is the creation of monolithic grammar files. When using built-in grammars, it is not a good idea to just copy the contents of a built-in grammar into a newly created grammar file.
Speech engines allow developers to activate and deactivate grammars as needed, and thus it is a good idea to keep grammars focused on a specific domain. If a certain prompt calls for several domains — e.g. users can issue prompt-specific commands, say yes/no, or use global commands — it is best to activate a few different grammars. If the next prompt only required one grammar, the others could be deactivated for that recognition.
Using modular grammars accomplishes several things. It keeps the number of words to be recognized at any given time down, helping accuracy. With a single monolithic grammar file that is

always active, a speech recognition engine tries to recognize words that are inappropriate for a specific prompt, increasing the chance that one of those words is returned by the engine. Modular grammars are also much easier to troubleshoot and tune. It is possible to introduce logical errors in grammar files, such as recursion problems, and if grammars contain fewer rules it is easier to diagnose these sorts of problems. It is also easier to make changes as part of the tuning process if grammars are kept small and specific to a prompt.

Keep Interpretation Inside Grammars

The SRGS grammar specification allows for semantic interpretation to be done within a grammar using a standard called Semantic Interpretation for Speech Recognition (SISR). Using ECMAScript (also known as JavaScript), SISR provides a robust method of handling semantic interpretation entirely within a grammar.

Briefly, semantic interpretation is the process of distinguishing between what a speaker says and what a speaker means. Imagine a call router prompt. A caller may ask to speak to "Technical Service" while another caller asks for "Technical Support." Both callers mean the same thing but use different words. Obtaining meaning from the exact utterance is semantic interpretation.

Because there exists such variety in how users may phrase the same response, semantic interpretation must be performed at some point in every speech application. New speech developers, often not familiar with SISR, place semantic interpretation within the code of their speech applications.

It is within grammars that semantic interpretation really belongs. Keeping it within an application can cause several problems, especially when making changes to that application. Imagine again that you have built the call router example previously mentioned. After it has been deployed, you find callers are also saying "Customer Support" to mean "Technical Support." First, you need to add the phrase "Customer Support" to the grammar in order to recognize it. If the semantic interpretation is done in your application code, you must now also update the application so that it can handle the new phrase.

If you use SISR and keep semantic interpretation within the grammar, you would only need to update the grammar. By keeping semantic interpretation within an application's code space, you effectively duplicate the words to be recognized from the grammar, creating problems if they ever become out of synch.

Test Changes With Real Data

An important part of developing speech applications is the tuning cycle, the iterative process where developers review the performance of an application and adjust applications and grammars accordingly.

Before changes in grammars are made to production systems, they should be tested thoroughly. Ideally, you should perform these tests using data gathered from actual calls. Using a tool like the LumenVox Speech Tuner, you can transcribe utterances from callers and then test new grammars against the transcribed audio. The Speech Tuner will let you know exactly how the changes in the grammars have affected recognition results.

Validating grammar changes against real call audio will help you safeguard against the problems mentioned previously, such as adding confusability while trying to account for rare responses. By testing changes before deployment, you will be able to tell if additions to grammars positively affect success rate.

For more information on the Speech Engine go to: www.lumenvox.com/products/speech_engine

Summary

Good grammars are as small as possible while still serving a majority of your callers, and grammars should not include words or phrases that are only used by a small percentage of callers. You should consider grammars an important part of a speech application's overall design. Your grammars are developed in tandem with the application prompts. These prompts should guide users to give predictable responses to help keep grammar size manageable.

As a speech developer, you should take advantage of built-in grammars included with a speech engine, but take care to not simply copy the contents of those grammars into your own. Instead, build small, modular grammars and load and unload them as needed by your speech application. Using standards like SISR, put semantic interpretation within your grammars so as to minimize the possibility of the various parts of the application becoming out of synch. Be sure that whenever you make changes to grammars that you test those changes, using data gathered from calls to that system.

Call LumenVox today at 1-877-977-0707 to discuss your project and learn about ways we can partner to make your speech application a success.

LumenVox