Developing Speech Grammars That Rock, Part 1: Best Practices

Flexible grammars are key to more natural interaction and a more pleasant caller experience. In the first installment of this two-part series on speech grammars, we cover the basics of grammars and outline some grammar development best practices.

by Steve Apiki

More Resources
• Fact Sheet: Avaya Global Services—Self-Service Consulting
• Additional DevConnect Resources

Done right, flexible, efficient grammars can make speech applications feel more like real speech, and less like the verbal equivalent of a game of "Simon Says" ("Press 'one' for a list of sandwiches," "If you'd like a slice of cheese, say 'cheese'," and so forth). Creating a grammar is seemingly easy, but getting a complex grammar right—building a grammar that responds quickly, and in ways that callers expect—is an iterative process that has its share of pitfalls. In this article, we'll focus on the "pitfalls", introducing speech grammars and then describing some critical best practices that can help keep your grammar development on track. In part two we'll come back around to the notion of "iterative" grammar development, working our way through the tuning workflow for grammars, and covering tuning techniques.

Grammar Development

In a speech application, a grammar is a set of rules that define the universe of words or phrases that can be recognized when spoken by the caller (the full set of phrases that can match a grammar is said to be generated by that grammar). A separate grammar set is often associated with each input state in a dialog, although some grammars may be used in more than one context.

The application hands the grammar off to a speech recognition engine (a component of the voice browser) for processing. In effect, the application uses the grammar to tell the recognizer what words or patterns of words should be expected at a given point in a dialog. The recognizer chooses the best match (if possible) from entries in the grammar, returning the entry and a confidence score that describes the "closeness" of the match. The confidence score is determined algorithmically by the recognizer, as part of the process of matching the template models from the grammar to the caller's utterance.

A well-designed grammar is flexible enough to match most of the responses you might expect from a caller, but restrictive enough to give the recognizer a reasonably small and distinct set of options from which to select a match. A good grammar optimizes recognition accuracy and enables the recognizer to return matches with higher confidence scores. The challenge of grammar development is choosing the right balance between flexibility and restriction.

Listing 1 shows an example of a GRXML grammar. This example is in the XML form developed by the W3C as part of its SRGS recommendation (this form is commonly called GRXML). There are also a number of other speech grammar formats that may be supported by speech recognition engines, including ABNF (a non-XML format defined in SRGS) and GSL (a proprietary Nuance format).

Listing 1. An example GRXML grammar.

<?xml version="1.0" encoding="UTF-8"?>
<grammar xmlns="http://www.w3.org/2001/06/grammar" xml:lang="en-us"
Listing 1 matches phrases that the caller might use to request website help or to cancel his or her service. Although it's simple, it shows two interesting features typical of speech grammars. First, several synonyms are mapped to a single return value (using the one-of element) for both options. To cancel service, the caller could say either "cancel" or "cancel service", yet "cancel" will be returned as the (semantic) response (or "slot") value in both cases.

Second, the postphrase rule defines an optional utterance (i.e."please") that may occur after the main entry. Because of the postphrase, "cancel please" would also match the cancel rule. The postphrase rule is an example of postfiller. Postfillers consist of utterances that come after keywords that may increase the range of possible responses but which are ignored by the speech recognizer in determining the match. Similarly, prefillers consist of utterances that may occur before the target (e.g., "I want to") but which don't affect the return value.
Some Best Practices
When approaching grammar development it is important to recognize that every application is
different, each with its own set of data rules and its own target audience. What's more, since
callers are human, it's not possible to create hard and fast rules about the best way to build a
grammar. Instead, we've compiled a short list of best practices to use in grammar design. This is
not a comprehensive list, but applying these practices will go a long way toward creating efficient,
responsive grammars.

Best Practice #1: Consider All Relevant Prompts
Carefully review the prompt or set of prompts that are associated with a grammar to come up with
terms that a caller is likely to use. If a grammar is associated with more than one prompt—for
example if the initial prompt and reprompts differ however slightly in what they instruct callers to
say—review each of those prompts independently to come up with a set of appropriate target
phrases and synonyms.

Callers tend to parrot back prompts, so you may hear more of the prompt in the response than
just the expected keyword. For example, a caller might respond to a prompt that says, "Say 'main
menu' to start over" with phrases such as "main menu to start over", "start over" or just "main
menu." This requires that you expand the synonym list to include additional phrases.

Say you have the following prompts, both associated with the same grammar:

"To look up an order, just say "order"..."
"Remember: Just say 'order' to look up another order..."

Although "order" is the keyword you're looking for, the caller might also say "look up an order" in
response to the first prompt and "look up another order" in response to the second. Both must be
handled by the grammar.

Nevertheless, when building the initial grammar, start with expected answers (phrases a
cooperative caller would say) and err on the side of excluding possible additional synonyms
rather than including them. The more synonyms you have, the greater the chance of incorrect
matches. The number and acoustic similarity between in-grammar phrases can also deflate
confidence scores, thus potentially resulting in unnecessary reprompting. Additional synonyms
can be added during application tuning, when you have evidence that a reasonable number of
callers are speaking those phrases.

Best Practice #2: Avoid Overgeneration
Combining prefill, postfill, and a keyword with several synonyms can quickly lead to an explosion
of combinations of valid responses. The number of combinations is given by:

\[(# \text{prefillers} + 1) \times (# \text{entries}) \times (# \text{postfillers} + 1)\]

With just two prefillers (e.g., "help with" and "help me with") and one postfillers ("please"), a
grammar with four entries turns into 24 valid phrases. This can negatively impact recognition
because, as mentioned above, the more in-grammar items there are, the greater the likelihood of
a false or low-confidence match.

But overgeneration can also lead the application to accept nonsensical responses. Consider a
grammar used to locate a hotel. The grammar includes the names of all fifty states, plus the
additional keyword "international". It also includes a prefiller value of "the hotel is in." A user might
reasonably say "the hotel is in Texas." But this grammar would also match on "the hotel is in
international." Accepting nonsense responses makes the grammar needlessly complex, thereby
negatively impacting recognition performance and potentially detracting from the caller experience. Essentially, if a native speaker wouldn't naturally produce it, don't include it in your grammar.

Overgeneration of this kind can be reduced by not automatically applying sub-rules with simple combination. In the example above, each state name might be given a synonym that includes "the hotel is in" phrase, and the prefiller could be removed. Alternatively, this prefiller rule could be applied only in the domains that it makes sense. There are also more sophisticated techniques, for example the use of JavaScript or a dynamic grammar, which can also help to combat overgeneration.

**Best Practice #3: Re-Use, but Re-Use Carefully**

As we said earlier, a grammar is a set of expected phrases for each individual input state. And, because a speech application often has dozens of different prompts, it's tempting to re-use grammars for what appear to be almost identical interactions. But even simple grammars may not be as generally applicable as they seem. Again, it's important to note the differences between candidate prompts where grammar-sharing is being considered. Take, for example, a simple "yes/no" grammar, which may include yes, no, and a set of general synonyms (eg, "OK"). Now let's look at two prompts that may seem on the surface to elicit similar responses:

"That was Austin. Did I get that right?"
"Would you like to place an order?"

In response to the first prompt, a caller may say "right," "that's right," "yes you did" or "correct" as well as the aforementioned generic entries; in response to the second, he or she may say "yes I would" or "yes please".

It may be tempting to salvage the reusability of this grammar by expanding it to include all of these potential entries. But doing so would introduce overgeneration given the domain and its associated recognition pitfalls.

Better candidates for re-use are grammar rules that can be used at prompts throughout the application, such as "help me out" or "operator." Other candidates for reusable grammars are common data formats, such as dates (but be sure that they really are the same kind of "date"—see Best Practice #4).

**Best Practice #4: Know Your Data**

You can exploit the differences among different types of data and among different data formats to build highly selective grammars. By restricting potential matches to those both allowed by data format rules and valid for a specific data type, you can greatly reduce the recognition domain, thus boosting recognition accuracy and confidence.

Account numbers are often restricted to a specific format, such as a certain number of digits, or a requirement that the account starts or ends with a letter. Adding these restrictions to the account number grammar is an example of how you can take advantage of format restrictions to build better grammars. Taking advantage of known data formats is crucial when employing alphanumeric grammars and is recommended even for varying length digit strings when it's not known which length string the caller will use.

Be specific when identifying data types for each element. A reservation date (which must be in the future) and a birth date (which must be in the past), though both dates, belong to completely disjoint sets. You can reduce grammars for these data items by including past and future restrictions.
If you have enough domain knowledge, you can and should introduce additional type restrictions. For example, suppose you are processing inquiries for a retirement community with a minimum age requirement. You can use the minimum age to further narrow the range of probable birth dates.

Including these restrictions up front in the grammar, creates additional benefits in addition to increasing recognition accuracy. It offloads some of the validation logic from that application to the recognizer, reducing the load on the application. It also lowers the risk of confirming an utterance in the dialog ("You said March 4th, 1921. Is that right?") only to have it later rejected by the application ("I'm sorry, that's not a valid date for travel"). Consistency errors such as these can quickly frustrate callers and result in a loss in caller confidence.

Starting Early

To take advantage of structures and patterns within the data that can enhance recognizer accuracy, it's critical that you start the grammar design process early, during the first stages of application design. By starting early, you can ensure that the data you need to restrict grammars appropriately can be made available by the host system, and that the recognition task required for a given input state is technically feasible. For example, you could load a grammar used for movie ticket purchase with a list of features that are currently playing, but you might need to build the web service to deliver that list as part of the application development effort.

Starting early also allows application developers to work closely with speech scientists to coordinate prompts and grammars with other parts of the application. As with any UI elements, speech application prompts and grammars can't be bolted on at the end of the process—they need to be designed and integrated into the flow of the application.

Avaya supplies both tools and professional services expertise in support of grammar development. Avaya Dialog Designer includes a built-in grammar editor for VXML applications that allows the user to create grammars as they work in a graphical dialog editor. The grammar editor in Avaya Dialog Designer creates list-based SRGS-compliant grammars for any of the speech recognition engines supported by Avaya Voice Portal. Avaya Professional Services also offers a wide range of consulting services, including grammar design and development support and application tuning.

Developing flexible and efficient grammars pays off in speech applications that respond to typical speaking patterns accurately the first time. But getting all the way there requires some additional work beyond initial deployment. In part 2 of this series, we'll talk about tuning, and how to analyze data from actual calls to further improve grammars.

Steve Apiki is senior developer at Appropriate Solutions, Inc., a Peterborough, NH consulting firm that builds server-based software solutions for a wide variety of platforms using an equally wide variety of tools. Steve has been writing about software and technology for over 15 years.