

# Automatic Grammar Construction

Alexander Gruenstein  
alexgru@stanford.edu  
Symbolic Systems 115

March 18, 2002

## 1 Introduction

Currently, it often requires a small army of linguists, computer scientists, psychologists, and voice talents to design and implement a robust and usable spoken language understanding system. Moreover, their expertise may be required yet again when it comes time to modify the system itself or underlying databases. This process can be expensive, tedious, and error prone. Moreover, in many applications designed for businesses, it seems highly desirable that the people who are actually involved in that business on a day to day basis should have the capability to modify the spoken language understanding system that serves their business without the help of experts.

A domain where non-experts might want to make updates to a voice system serving their business on a day to day basis is that of ordering take-out from restaurants over the phone. Accessing restaurants over the telephone has several interesting aspects that need to be addressed in order for a voice system to be successful:

1. The procedure of browsing through a restaurant's menu and then ordering dishes at a restaurant or over the phone is one that just about everyone is familiar with. Based on their wealth of human-to-human interactions in this domain, users are likely to have many preconceptions and expectations about how to go about this process with an automated voice system.
2. Some users will call already knowing the dishes that they want to order while other users will want to be able to browse through a restaurant's entire menu before deciding on the dishes they'd like to order just as they might the first time they go to a new restaurant. Some people who browse through the menu will want to order items as they browse, others might simply mentally keep track of the dishes they are considering and then decide after they have heard all the options.
3. Restaurants often have menus that change on a daily basis. Many restaurants have daily specials and some change their entire menu every day. As

such, a robust voice-driven system for ordering take-out needs to be one that can adapt on the fly to new menus. Moreover, this adaption must be fully automated: in a large system with many restaurants, it's not feasible for experts to come together on a daily basis and modify the system based on the day's menus.

In this paper, I describe a relatively simple *Restaurant Voice Portal* that is a first stab at creating a voice driven system given the three constraints above. Significantly, I describe a means of generating the recognition grammars for such a system automatically, without the intervention of a professional linguist. I compare my designed system to a control system that doesn't use the automatic grammar generation routines and show that the automatically generated grammars lead to improved system performance and user satisfaction.

## 2 Overview of the *Restaurant Voice Portal*

In this section, I will describe the design and implementation of the *Restaurant Voice Portal*. It is a prototype system with the full menu for one restaurant (Hobees).

### 2.1 Web Database Interface

A major goal of the prototype system was to make it easy for restaurant owners/operators to easily and quickly update their menus, without knowing anything about VXML or recognition-grammar writing. Towards that end, I created a simple web-based interface that allows anyone capable of using a web browser to create and edit menus. Included in this system is the ability to create and edit the names of menu sections (for example *Drinks*, *Pasta Entrees*, or *Desserts*), individual menu items, and descriptions of those menu items. Of course, a more complete version would also have to include more options, like the price for items and side dish options.

The web database interface is implemented as HTML forms generated in PERL. I used the interface to enter the entire menu of the first restaurant that came up for Palo Alto on waiter.com: Hobees. Please refer to my Powerpoint presentation for screenshots of this system.

### 2.2 The Voice Interface

The following is a segment of a sample interaction with the system:

S: Welcome to the Hobee's voice portal! If you know what you want to order, just say the name of the item. To browse through the menu, just say *browse*. How may I help you?

U: Browse.

S: Okay, let's browse! There are 13 sections of the menu. The first three are: *omelets; scrambles; special hashbrowns*. To browse a section of the menu, just say that section's name. To hear more sections, just say *next*. You can always say *main menu* to go back to the main menu. How may I help you?

U: Next.

S: The next three sections are: *classics; favorites; light bites*. How may I help you?

U: Light Bites.

S: Okay, Light Bites! There are 8 items in this section of the menu. The first three are: *our famous blueberry coffee cake; hobee's homemade muffins; side of hobee's-style hashbrowns*. To order an item, just say its name. To hear more items, say *next*. You can always return to the main menu by saying *main menu*. How may I help you?

U: Next.

S: Next we have: *side of toast; fresh bagel; oatmeal bar*. How may I help you?

U: Next.

S: Finally, the last two items are: *Hobee's gourmet granola; seasonal fresh fruit plate*. How may I help you?

U: Granola.

S: Okay! I've added *Hobee's gourmet granola* to your order. Just say *checkout* when you're done ordering...Finally, the last two items are: *hobee's gourmet granola; seasonal fresh fruit plate*. How may I help you?

U: Browse

S: The next three sections are: *Light Bites; Soup Salad and Starters; Sandwich board*. How may I help you?

U: The Sandwich board

S: Okay, Sandwich board! There are nine items in this section of the menu. The first three are: *jack cheese, avocado and tomato sandwich; tuna, cucumber, and tomato sandwich; turkey, avocado, tomato sandwich*. To order an item, just say its name. To hear more items, say *next*. You can always return to the main menu by saying 'main menu.' How may i help you?

U: Turkey avocado.

S: Okay, I've added *turkey, avocado, tomato sandwich* to your order

⋮

The goal in creating the voice interface was to create a system that could be updated dynamically based on the back-end database. Moreover, I aimed to create a system that would adhere to people's expectations and preconceptions about ordering dishes in a restaurant. To that end, I wanted to minimize the tedium involved in browsing potentially long restaurant menus. Going through the menu involves navigating through a series of lists. Rather than forcing users to go through lists on an item by item basis, I chose to let them browse through lists in reasonably sized chunks of three items at a time, in the hopes that this would help them get some idea of what sections were available on the menu before they had to descend into browsing specific sections. Similarly, they could get a sample of the types of dishes in a section without having to navigate tediously up and down a list one item at a time. I made sure that the system incorporated the discourse markers *first*, *next*, and *finally* to make the list navigation more natural. I also ensured that the system confirmed commands by responding with phrases like *Okay! Let's browse*.

The entire interface incorporates these ideas. The user logs on to the system and is greeted by the main menu. From there, he can either order an item on the menu or *browse* through the sections of the menu. If he chooses to browse, then he is presented with all the sections in the menu in groups of three. He can say *next* to get to the next set of three, *main menu* to get back to the main menu, or the name of a particular section to browse. Once he is navigating a particular section, he uses the same commands (*browse*, *main menu*, and *next*) to navigate, or he can say the name of an item on the list that he wants to order. At any point in the system, he can *checkout*. While checking out, he can remove items from his list, go back to the main menu, or check out and exit.

My second major aim in adapting to users' preconceptions about restaurant ordering was to allow users to naturally refer to the dishes they wanted to order. For example, in the dialogue segment above, the user orders *Hobee's gourmet granola* by simply saying *granola*. Similarly, he orders the *Turkey, avocado, tomato sandwich* with the phrase *Turkey avocado*. In the context, these phrases are unambiguous even though they don't precisely echo the phrase the user has just heard.

Moreover, I wanted my system to be able to handle the case of the experienced user, who calls the system already knowing what he wants (or the user who browses the menu and then returns to the main menu with the dishes he wants in mind). The following dialogue fragment is an example of such a dialogue:

S: Welcome to the Hobee's voice portal! If you know what you want to order, just say the name of the item. To browse through the menu, just say *browse*. How may I help you?

U: cardinal omelet

S: Okay! I've added *The Stanford Cardinal Omelet* to your order. Just say *checkout* when you are ready to checkout. How may I help you?

U: the whole wheat pancakes

S: Okay! I've added *Hobee's Hi-Protein Honey Whole Wheat Pancakes* to your order. How may I help you?

:

The above fragment demonstrates the appeal of ordering items without referring to them by their entire, proper name. It's unrealistic to expect that users will be able to remember (or want to remember) a complicated name like *Hobee's Hi-Protein Honey Whole Wheat Pancakes*, especially when they could order them in a person-to-person interaction simply by referring to them as *the whole wheat pancakes* or *the honey whole wheat pancakes*.

In a hand-designed system, the above variability could be captured by a carefully designed recognition grammar. A major goal of my system, however, is to eliminate the need for exactly this sort of expert intervention. As such, I created a PERL subroutine (see Figure 1 for complete code) that takes as input a list of phrases and outputs a recognition grammar that maps every unique substring (along word boundaries) of each phrase onto that phrase (and allows for an optional determiner *the*). For example, given the following input:

dish 1: super veggie scramble

dish 2: salmon spinach scramble

the subroutine returns a grammar of the form:

dish 1: ((the)? [(super) (super veggie) (super veggie scramble) (veggie) (veggie scramble)])

dish 2: ((the)? [(salmon) (salmon spinach) (salmon spinach scramble) (spinach) (spinach scramble)])

Note that while the grammar allows for phrases like *the super veggie* and *veggie scramble* to map onto the first dish, it bans *scramble* since it is ambiguous in terms of which dish it refers to. In the next generation of the system, it might be interesting to initiate a disambiguation sub-dialogue upon recognizing *scramble*; in my prototype system, however, *scramble* is simply prohibited.

This automatic grammar generation was employed throughout the system. Specifically, it was applied in the following areas:

1. The Main Menu: to recognize any item on the entire menu.
2. Section Lists: to recognize one of the three sections just mentioned

3. Item Lists: to recognize one of the three items just mentioned
4. Checkout: to remove any item on the checkout list

In addition to the automatic grammar creation, all of the VXML for the system is generated on the fly by PERL scripts that read the database created by the web-database interface described above. If this overhead were undesirable in a deployed system, a large static cache could be made of all possible navigation trajectories whenever the menu is modified.

### 3 User Testing

Using automatic grammar construction has several conceivable pros and cons. Clearly, its major advantage is that the bulk of the grammar for the system is created automatically and can be produced in real time without the intervention of a linguist. Its drawback is that the grammar is relatively large, and may contain many phrases that users are highly unlikely to say. For instance, the automatically created recognition grammar accepts *cruz scramble* for the phrase *Santa Cruz Scramble*, even though this seems like an awkward way to refer to the dish. My worry was that such a large grammar would lead to an increase in the number of misrecognitions when compared to a recognition grammar that allowed only for the exact name of items as they appeared in the database. I hypothesized, however, that despite this possible limitation, the tradeoff would be worth it because the number of recognition failures would decrease. Moreover, I hypothesized that users would generally be more satisfied with the large-grammar system.

#### 3.1 Procedure

I used TellMe Studio to host my VXML-based systems. Two systems were compared that had identical prompts generated by TellMe's text-to-speech engine (see the dialogue fragments in the previous section for example prompts). The two systems differed only in their recognition grammars: the large-grammar system used grammars generated by the automatic grammar generation subroutine describe above; the small-grammar system allowed the user to refer to menu items and sections only exactly as they were entered into the database and read aloud by the system.

Subjects were given two tasks. They were asked to first browse through the menu and order two items each from four different sections of the menu. Next, they were told to call the system back and order as many of the items as they could remember from the first step again. This time, they were told not to browse through the menu at all, but just to call up the system and order the items from the main menu. All 7 subjects were college students. 4 subjects called the large-grammar system and 3 interacted with the small-grammar version. All calls were done using Voice-Over-IP, since TellMe is experiencing trouble interacting with Stanford's phone system.

Table 1: **Results: Accuracy***Small Grammar*

	Nonrecognitions	Misrecognitions	Total Error
Task 1	45%	10%	55%
Task 2	13%	7%	20%

*Large Grammar*

	Nonrecognitions	Misrecognitions	Total Error
Task 1	11%	15%	26%
Task 2	8%	22%	30%

Table 2: **Results: Average User Ratings**

	How well understood	How easy to use
Small Grammar	3	3
Large Grammar	5	3.7

On a scale where 1 = Not at all; 7 = extremely well

For each of the two tasks, the percent of non-recognitions and misrecognitions were tracked. In addition, after using the system, each subject was asked to rate how well the system understood him or her, and how easy it was to interact with the system.

### 3.2 Results and Discussion

Table 1 contains the average percent of recognition errors made in each task in each system across subjects. Table 2 shows the average usability ratings assigned to each system by the subjects.

Note that for the first task, while the number of misrecognitions made by the large-grammar system is slightly greater than the number made by the small-grammar system, the number of non-recognitions is significantly smaller. As I had hypothesized, the larger grammar did indeed lead to a greater number of misinterpretations. However, in this case, the steep drop in nonrecognitions seems to more than make up for this increase. While in the small-grammar system, almost half of the users' utterances were not recognized, only 11% of them, on average, failed to be recognized by the large grammar system. For task 1, overall, the large-grammar system came up with a correct interpretation, on average, of 74% of user utterances, compared to a meager 45% by the small-grammar system! I often observed subjects becoming frustrated with the small-grammar system, as they would repeatedly try to order an item using some slight variation of its full name over and over again. When I pointed out to them after the experiment that they hadn't been saying exactly what they had

heard the system say in these instances, they were surprised – they had never realized that this was the problem

In the more open-ended task 2 (ordering any dish on the menu from the system’s main menu prompt), the small-grammar system performed better overall. While fewer utterances were outright rejected by the large-grammar system, there were a greater number of misinterpretations. Note, however, that the amount of data here is extremely limited, since most subjects only made five or six utterances in this task. Nonetheless, the data does suggest that a more robust automatic grammar generation algorithm might be needed when generating recognition grammars for extremely long lists (there are approximately 70 items on the menu).

The large-grammar system was preferred by users in the sense that they rated it as understanding them better. While talking to subjects afterwards, several said that they were quite surprised and impressed when the large-grammar system had ‘figured out’ what they wanted based on their hesitant, partial utterances. Overall, subjects also rated the large-grammar system as slightly easier to interact with.

## 4 Conclusion

The algorithm proposed here for automatic grammar construction in the context of navigating lists does lead to increased recognition accuracy compared to a stricter, small-grammar system that recognizes only the exact names of items on the list. However, as the results show, there is still plenty of room for improvement. Future algorithms for automatic grammar construction might contain domain-specific and/or domain-general heuristics to limit the types of sub-phrases constructed.

The mild success of the prototype system here suggests that a more full-featured version of the same system might eventually be deployable. Such a system puts the power in the hands of the people who design menus best: professional restaurant operators.

Figure 1: PERL code to create recognition grammar

```

# Given a list of strings, create a grammar that allows the user to
# say any unique substring of a given string in order to recognize the string,
# where a unique substring is a substring that doesn't appear in any of the
# other strings on the list. (Here, we treat each word in the string as
# a possible element on the substring [rather than each character]).
# Also, allow "the" to appear in front of each substring
sub MakeGrammar {
    my(@names) = @_;
    my(@grammarList); #one slot for each string
    my($length) = scalar(@names);
    @name = lc(@names);
    my($isSubstring);
    for($i = 0; $i < $length; $i++) {
        $name = $names[$i];
        my($grammar) = "((the)? ["; #initialize it
        $name =~ tr/[, -]/ /;
        my(@words) = split(/\s+/, $name); #split name into words separated by whitespace
        #start with the first word only, keep on adding words until we have the entire
        #phrase, checking each time if this subphrase is contained in the other
        #names; then iterate to the second word and so on
        my($phrase);
        my($numWords) = scalar(@words);
        for($startWord = 0; $startWord < $numWords; $startWord++) {
            $phrase = "";
            for($lastWord = $startWord; $lastWord < $numWords; $lastWord++) {
                $space = " ";
                if($phrase eq "") { $space = "";}
                $phrase = $phrase . $space . $words[$lastWord];

                $isSubstring = 0; #false
                for($j = 0; $j < $length; $j++) {
                    if($i != $j) { #don't check against yourself!
                        if(index($names[$j], $phrase) != -1) { #found a substring!
                            $isSubstring = 1;
                            last; #break out of the loop
                        }
                    }
                }
            }
            if(!$isSubstring) { $grammar = "$grammar ($phrase)"; }
        }
        $grammar = "$grammar]";
        push(@grammarList, $grammar);
    }
    return @grammarList;
}

```