International Conference on Spoken Language Processing, pp. 2161–2164, Jeju, Korea, 2004

Creating Speech Recognition Grammars from Regular Expressions for Alphanumeric Concepts

Ye-Yi Wang[1] *and Yun-Cheng Ju*[2]

[1]Speech Technology Group, Microsoft Research
[2].Net Speech Group
Microsoft Corporation, One Microsoft Way, Redmond, WA 98052, USA
{yeyiwang,yuncj}@microsoft.com

Abstract

To bring speech recognition mainstream, researchers have been working on automatic grammar development tools. Most of the work focused on the modeling of sentence level commands for mixed-initiative dialogs. In this paper we describe a novel approach that enables the developers with little grammar authoring experience to construct high performance speech grammars for alphanumeric concepts, which are often needed in the more commonly used directed dialog systems in practice. A developer can simply write down a regular expression for the concept and the algorithm automatically constructs a W3C grammar with appropriate semantic interpretation tags. While the quality of the grammar is ultimately determined by the way in which the regular expression is written, the algorithm relieves the developers from the difficult tasks of optimizing grammar structures and assigning appropriate semantic interpretation tags, thus it greatly speeds up grammar development and reduces the requirement of expertise. Preliminary experimental results have shown that the grammar created with this approach consistently outperformed the general alphanumeric rules in the grammar library. In some cases the semantic error rates were cut by more than 50%.

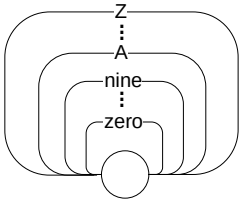
1. Introduction

With the steady improvements over the past two decades, speech recognition technology is on the verge of becoming mainstream in the real world. To catalyze this process, researchers have been working on language learning tools that facilitate the rapid development of speech-enabled dialogs and applications [1-5].

Most of the researches, if not all of them, focused on the learning of the spoken language understanding models for the sentence level commands that are often observed in the mixed-initiative dialogs. While they were good at modeling the commands that contain multiple phrasal semantic units (slots), for example, a "ShowFlight" command like "List the flights from Seattle to Boston on Tuesday that costs no more than $400" in the domain of Air Travel Information System, they seldom studied the acquisition of the phrasal model for the low level concepts like date, time, credit card number, flight number, etc. Instead, they resorted to grammar libraries and database entries (e.g., city names from an application database) for solutions.

On the other hand, a majority of the spoken language systems deployed so far are system-initiative, directed dialog systems. In such systems, most of the grammar development efforts are devoted to the low level concepts. Hence the aforementioned example-based learning tools are less attractive to the dialog system developers in this reality. While the grammar libraries and database entries are viable solutions, they did not solve the problem completely --- grammar library developers cannot foresee all possible domain specific concepts and pre-build grammars for them; the orthographic form of the database entries are often not sufficient to serve as the speech recognition grammar. For example, a proper speech recognition grammar needs to model a variety of expressions for an alphanumeric string like the parts number "ABB123" in a database --- to name a few, "A B B one two three", "A double B one twenty three."

According to the feedback that we received from the speech application developers, grammar development for the alphanumeric concepts like parts number and driver license number is one of the most challenging tasks. Often they started with a single state finite state model that loops over the alphanumeric alphabet, as illustrated by the diagram below:



The simple grammar was quickly found improper by the developers due to the following reasons:

1. The grammar does not capture the specificity of the target sub-languages. Therefore, the perplexity of the model is much higher than it should be. For example, if it is known that the parts number always starts with letter 'B', the grammar should explicitly model the constraint so that recognition errors that confuse 'E' with 'D', 'E', 'G', and 'P' will never occur.
2. The simple grammar does not model the diversity of linguistic expressions for digit strings. It has been shown in the previous example that the string "123" can be read in many ways different from what were modeled by the simple grammar.
3. Special characters like '-' and '*' often appear in the sequence. The general alphanumeric grammar needs to be customized in such cases.

Hence developers often write their own grammar for specific alphanumeric concepts. The process is tedious and error-prone. Unlike the grammar library, the grammars authored by

the less experienced developers are often not optimized, thus have poor performance when used by the decoder.

In this paper, we propose a novel solution to the grammar authoring problems for the domain specific alphanumeric concepts. A speech recognition grammar can be automatically constructed upon a developer's input of the regular expression for an alphanumeric concept.

2. Creating Recognition Grammar from Regular Expressions

2.1. Regular Expressions

W3C standard has the following formal definition for regular expressions [6]:

```
regExp ::= branch ( '|' branch )*
branch ::= pieces*
piece  ::= atom quantifier?
atom   ::= char|charClass|('(' regExp ')')
```

According to this definition, a regular expression consists of one or multiple alternates (branches). Each branch consists of a sequence of pieces. Each piece is an atom that is optionally quantified. The quantifier specifies the repetition of the atom. It can be a number (e.g. {3}), a number range (e.g. {0-3}) or a reserved character (e.g. '+' for more than once, '*' for zero or more times). The atom can be a character, a character class (e.g. [A-Z] for all uppercase letters, \d for the ten digits), or recursively a parenthesized regular expression.

A regular expression with recursive regular expression atoms can be converted to one without the recursive atoms. For example, "(\d{3} | [A-Z]){2}C" defines the same language as "\d{3}[A-Z]C | \d{6}C | [A-Z]\d{3}C | [A-Z]{2}C" does. In this work we require the use of the regular expressions without recursive regular expression atoms.

2.2. The Conversion Algorithm

The pseudo-code in Figure 1 shows the algorithm that creates a W3C speech recognition grammar from a regular expression.

The code assumes that the regular expression objects have the methods to access their components. For example, the method **`pieces`**`()` returns the list of pieces in `branch`, a branch in a regular expression. It represents the right hand side of a rule as an array of **`rule_token`**'s. Each **`rule_token`** is a tuple (`symbol, min, max`) that specifies that `symbol` repeats from `min` to `max` times in the rewriting rule.

The first (main) function calls the second function to create the rules for each branch of the input regular expression, and adds rules that rewrite the root symbol to the symbols for the branches. The second function calls the third function to create the rules for each piece in a branch, and add the rule that rewrites the branch symbol to the piece sequence. The third function (line 22-41) creates the rules for a piece. It first checks the character set defined by the atom of the piece. If it is the digit set, it ruleref's to the library grammar rule of fixed length digit sequences (line 26-34.) Otherwise it creates a rule that covers all the elements of the character set (line 36-37) and calls the function **`create_repeats`** for the repetition of the character set. That function in turn creates the prefixed grammar rules for 0 time occurrence (line 48-52); one time occurrence (line 54-57); more than one occurrence (line 58-80) --- note how it models the expressions like "double zero" in line 64-70; and more than two occurrences (line 72-80).

```
1.  create_regexp_grammar(regexp, gram) {
2.    foreach branch in (regexp.branches()) {
3.      string symbol =
4.        create_branch_grammar(branch, gram);
5.      rule_token token(symbol,1,1);
6.      array RHS = (token);
7.      gram.add_rule(gram.root(), RHS);
8.    }
9.  }
10. create_branch_grammar(branch, gram) {
11.   array RHS = ();
12.   foreach piece in (branch.pieces()) {
13.     string symbol =
14.       create_piece_grammar(piece, gram);
15.     rule_token token(symbol,1,1);
16.     RHS.add(token);
17.   }
18.   string LHS=new_symbol();
19.   gram.add_rule(LHS, RHS);
20.   return LHS;
21. }
22. create_piece_grammar(piece, gram) {
23.   atom unit = piece.atom();
24.   pair (min, max) = piece.quantity();
25.   set charset = unit.CharSet();
26.   if (charset == {0, …, 9}) {
27.     string LHS=new_symbol();
28.     for (int i=min; i<=max; i++) {
29.       string ref = ruleref(lib, digit_i);
30.       array RHS=(rule_token(ref, 1, 1));
31.       gram.add_rule(LHS, RHS);
32.     }
33.     return LHS;
34.   }
35.   else {
36.     string charset_nt =
37.       create_charset_grammar(charset,gram);
38.     return
39.       create_repeats(charset_nt, min, max);
40.   }
41. }
42. create_repeats(symbol, min, max) {
43.   if (hash[symbol, min, max] != null)
44.     return hash(symbol, min, max);
45.   string LHS = new_symbol();
46.   hash[symbol, min, max] = LHS;
47.   array RHS=();
48.   if (min == 0) {
49.     rule_token token(symbol, 0, 0);
50.     RHS.add(token);
51.     gram.add_rule(LHS, RHS);
52.   }
53.   if (max <= 0) return LHS
54.   rule_token token(symbol, 1, 1);
55.   RHS = (token);
56.   if (min <= 1)
57.     gram.add_rule(LHS, RHS);
58.   if (max >= 2) {
59.     string rest1=
60.       create_repeats(symbol, min-1, max-1);
61.     rule_token rest1_token(rest1, 1, 1);
62.     RHS.add(rest1_token);
63.     gram.add_rule(LHS, RHS);
64.     RHS = (rule_token("double", 1, 1));
65.     RHS.add(token);
66.     string rest2=
67.       create_repeats(symbol, min-2, max-2);
68.     rule_token rest2_token(rest2, 1, 1);
69.     RHS.add(rest2_token);
```

```
70.     gram.add_rule(LHS, RHS);
71.   }
72.   if (max >= 3) {
73.     RHS = (rule_token("triple", 1, 1));
74.     RHS.add(token);
75.     string rest3=
76.       create_repeats(symbol, min-3, max-3);
77.     rule_token rest3_token(rest3, 1, 1);
78.     RHS.add(rest2_token);
79.     gram.add_rule(LHS, RHS);
80.   }
81.   return LHS;
82. }
83. create_charset_grammar(charset,gram) {
84.   string LHS=new_symbol();
85.   array RHS=();
86.   foreach ch in (charset) {
87.     switch (ch) {
88.     case '0': RHS=(rule_token("zero",1,1));
89.               gram.add_rule(LHS, RHS);
90.               RHS=(rule_token("oh",1,1));
91.               gram.add_rule(LHS, RHS);
92.               break;
93.     case '1': ……
94.   }
95.   return LHS;
96. }
```

Figure 1. The pseudo-code that constructs a grammar from the regular expression `regexp`.

Not explicitly shown in Figure 1, the algorithm automatically attaches semantic interpretation tags to the rule tokens, so the recognition outputs are appropriately normalized.

3. Experimental Results

We conducted speech recognition experiments with the grammars for three different alphanumeric concepts: social security number (SSN), license plate number (LPN), and Washington State driver license number (WADL). The regular expressions that were used to generate the recognition grammars are listed in Table 1.

Concept	Regular Expression
SSN	\d{3}-\d{2}-\d{4}
LPN	\d[A-Z]{3}\d{3}
WADL	[A-Z]{2}[A-Z*]{3}[A-Z][A-Z*]\d{3}[A-Z]{2}

Table 1. Regular expressions used to generate the W3C speech recognition grammar.

220 samples were generated randomly for each concept from these regular expressions, except for the WADL. The Washington State driver license number starts with the five initial letters of a person's last name (filled with '*' if the last name is shorter than five), followed by the first initial and the middle initial ('*' if no middle initial), and then three digits and two letters. We randomly picked 220 last names from the US Census Bureau's 1990 census [7], took the first five letters from each name and appended it with a string randomly generated according to the regular expression "[A-Z][A-Z*]\d{3}[A-Z]{2}."

Speech data was collected from eleven subjects. The subjects were Microsoft employees, including five speech researchers, four software engineers/testers and two administrative staffs. Seven of the subjects are native US English speakers, among them two are females. The rest four non-native speakers are all males. The subjects were instructed to read out the numbers in the normal way they speak in daily life. 20 utterances per concept were collected from each subject.

Microsoft Yakima, the speech engine that ships with several Microsoft products, was used in the experiments, together with its default speaker independent acoustic model. For each concept, we used as the baseline the general fixed-length alphanumeric grammar from the grammar library provided in the Microsoft Speech SDK, with the modifications that add the words "dash", "hyphen" for the character '-' and "asterisk", "star" for the character '*' in the vocabulary. The same grammar library also contains the rule USSocialSecurity (Lib/SSN). We used it as an alternative (presumably better) baseline for the SSN task. The recognition outputs were then compared with the original numbers used for data collection. The statistics of character error rates and semantic error rates were collected --- a recognition hypothesis is considered a semantic error if one or more character errors occur.

Table 2 shows the error rates of the three different grammars for the SSN task. Compared with the library SSN grammar, the grammar generated from the regular expressions cut the semantic error rate by more than 60% when it is compared with the fixed length alphanumeric grammar, and over 40% when it is compared with the library SSN grammar. The alphanumeric grammar has the highest error rates because it doesn't cover the digit sequences that were read as a number. Lib/SSN has the error rate comparable to that of RE/SSN on nonnative speakers, but much higher error rates on the native speakers. This is due to the fact that one native speaker consistently read out the hyphens explicitly in the social security numbers, which is not modeled by Lib/SSN, and it is not as straightforward to add the hyphen to Lib/SSN as we extended the alphanumeric grammar. This demonstrates that generating grammar from regular expressions provides an effective alternative of a customized grammar when the existing library grammar fails to model some users' peculiar utterances.

		AN9-11	Lib/SSN	RE/SSN
Native Speakers	CER	11.1%	9.1%	0.6%
	SER	18.6%	18.6%	4.3%
Nonnative Speakers	CER	29.4%	4.6%	3.9%
	SER	58.8%	28.8%	28.8%
Overall	CER	17.8%	7.5%	1.8%
	SER	33.2%	22.3%	13.2%

Table 2. Character error rate (CER) and semantic error rate (SER) for the SSN recognition task. AN9-11 is the grammar for the alphanumeric string of length 9 to 11 (with the optional '-' in the SSN). Lib/SSN is the SSN grammar in the grammar library. RE/SSN is generated from the regular expression in Table 1.

Table 3 compares the license plate number grammar created from the regular expression (RE/LPN) with baseline grammar. Because RE/PLN modeled the diversity of digit sequence expressions and the constraints on the locations where a digit or a letter is expected, it cut the error rate by more than 50%. The location constraints significantly reduce the fan-out at each grammar state (hence the perplexity) and eliminate the frequently observed confusion between '8' and 'A'.

		AN-7	RE/LPN
Native Speakers	CER	6.5%	2.6%
	SER	34.3%	14.3%
Nonnative Speakers	CER	18.4%	9.1%
	SER	71.3%	37.5%
Overall	CER	10.8%	4.9%
	SER	47.7%	22.7%

Table 3. Character error rate (*CER*) *and semantic error rate* (*SER*) *for the license plate number recognition task. AN-7 is the grammar for the alphanumeric string of length 7. RE/LPN is generated from the regular expression in Table 1.*

		AN-12	RE/WADL
Native Speakers	CER	17.0%	14.3%
	SER	72.1%	47.9%
Nonnative Speakers	CER	37.7%	39.7%
	SER	97.5%	90.0%
Overall	CER	24.5%	23.6%
	SER	81.4%	63.2%

Table 4. Character error rate (*CER*) *and semantic error rate* (*SER*) *for the Washington State driver license number task. AN-12 is the grammar for the alphanumeric string of length 12. RE/WADL is generated from the regular expression in Table 1.*

Table 4 lists the error rates of the two WADL grammars. The grammar created from the regular expression again has better accuracy. However, both grammars have a very high error rate. Preliminary error analysis shows that the high error rates can be attributed to the following two reasons:

1. More (9) letters exist in a WADL number, and letters are acoustically more confusable than digits. Error analysis shows that the most confusable letters are 'S' vs. 'F', and then the E-class letters B, D, E, G, P, and T. This also explains why LPN has higher error rate than SSN.
2. WADL numbers often contain pronounceable substrings (last names or last name prefixes). Many subjects (often speech researchers who tried to break the system) opted to pronounce the name instead of spelling out the letters. About 8% of the data was read out with the substring pronunciation, which is completely uncovered by either grammar.

4. Discussion and Future Work

While the algorithm in Figure 1 relieves developers from the difficult tasks of optimizing the grammar structure for high performance (partially prefixing a grammar while retaining readability) and assigning appropriate semantic interpretation tags to grammar rule tokens for output normalization, it does not guarantee to output high quality grammars. In fact, the baseline alphanumeric grammars used in the experiments can also be generated from the regular expression "[A-Z0-9]{n}" (with the extra modeling of "double" and "triple" phrases). The quality of the grammar is determined ultimately by the way in which the regular expressions are written. Here are two major guidelines for writing regular expressions that can result in high quality grammar:

1. *Natural grouping*. The regular expressions should group the substrings in the natural way that people normally do. For example, the SSN regular expression in Table 1 is more suitable than "\d{9}", which creates the grammar of nine digit sequence for SSN.
2. *Specific modeling*. Use the specific character set that can occur at a position in an alphanumeric string rather than using its supersets. If it is known to be a digit, don't use [A-Z0-9]. If only 'S' can occur at a position, don't use the letter set [A-Z]. In doing so the recognizer will not confuse it with an 'F'. In the extreme case when an alphanumeric concept has finite instances, the regular expressions can be just the one with these instances as its branches.

As illustrated by the poor accuracy of the WADL task, the confusable letter sets and the substring pronunciation are the two major remaining problems in the alphanumeric concept modeling. As future work, we would like to study the role that character n-gram can play in confusable letter disambiguation, and investigate statistical models that can automatically learn the pronounceable substrings.

5. Conclusions

We have shown that quality speech recognition grammar can be constructed automatically for alphanumeric concepts from appropriate regular expressions. It relieves developers from the difficult tasks of optimizing grammar structures and assigning appropriate semantic interpretation tags, thus it greatly speeds up the grammar development for developers with little speech recognition grammar authoring experience. The approach is complementary to the researches that facilitate the grammar development via language learning algorithms.

6. References

[1] R. Pieraccini and E. Levin, "A Learning Approach to Natural Language Understanding." In the Proceedings of 1993 NATO ASI Summer School, Bubion, Spain, 1993.

[2] S. Miller, R. Bobrow, R. Ingria, and R. Schwartz, "Hidden Understanding Models of Natural Language." In the Proceedings of the 31st Annual Meeting of the Association for Computational Linguistics, New Mexico State University, 1994.

[3] Y.-Y. Wang and A. Acero, "Combination of CFG and N-gram Modeling in Semantic Grammar Learning." In the Proceedings of Eurospeech 2003, Geneva, Switzerland, 2003.

[4] Y. He and S. Young, "Hidden Vector State Model for Hierarchical Semantic Parsing." In the Proceedings of ICASSP 2003, Hong Kong, China, 2003.

[5] J. Glass and E. Weinstein, "SPEECHBUILDER: Facilitating Spoken Dialogue System Development." In the Proceedings of Eurospeech 2001, Aalborg, Denmark, 2001.

[6] http://www.w3.org/TR/xmlschema-2/#regexs.

[7] http://www.census.gov/genealogy/www/freqnames.html.