# LANGUAGE MODEL GRAMMAR CONVERSION[1]

*Julie Baca, Wesley Holland, Dhruva Duncan and Joseph Picone*

Center for Advanced Vehicular Systems, Mississippi State University
{baca, wholland, duncan, picone}@cavs.msstate.edu

## ABSTRACT

Supporting popular language model grammar formats, such as JSGF and XML-SRGS, has been an important step forward for the speech recognition community, particularly with respect to integration of human language technology with Internet-based technologies. Industry standard formats, though conceptually straightforward implementations of context free grammars, contain restrictions that make it nontrivial to support probabilistic finite state machines. These restrictions pose serious challenges when applied to all aspects of the speech recognition problem, such as the representation of hidden Markov models using in acoustic modeling. This paper compares and contrasts these formats, discusses the implications for speech recognition systems, and presents solutions that have been implemented in our public domain speech recognition system.

## 1. INTRODUCTION

Several industry standard grammar specifications such as the Java Speech Grammar Format (JSGF) [1] and the W3C XML Speech Recognition Grammar Specification (XML-SRGS) [3] have been created to support development of voice-enabled Internet applications. While these standards allow for the specification of context-free grammars (CFGs), most language models for automatic speech recognition have a regular grammar equivalent and can therefore be modeled as finite state machines (FSMs). To support language model creation using these standards, we developed a suite of software tools in our public domain speech recognition toolkit [3] that convert between these grammar formats.

Issues of theoretical equivalence and restrictions on conversions between regular and context free grammars have been studied and described extensively. No algorithm has been proven to perform conversions from arbitrary CFGs generating regular languages to FSMs without assuming certain restrictions on the grammar, i.e. no center-embedded non-terminals [3]. However, software tools have been developed for conversions between FSMs and CFGs, which assume such restrictions on the grammars handled [5]. Nonetheless, our experience has shown that the specifics of individual grammar formats present unique challenges. The remainder of this paper describes the technical issues encountered in our conversion process, our solutions to these issues, and offers insight into future development of robust, general purpose language model conversion tools.

## 2. GRAMMAR FORMATS

Our internal grammar format, known as IHD, is implemented as a set of hierarchically layered FSMs. An example is shown in Figure 1. Each FSM layer is a generic directed graph class or DiGraph. IHD binds these layers of FSMs together into levels. The top-most level contains a single DiGraph, with the nodes of this DiGraph mapping to more complete DiGraphs. For example, the top-most layer might represent the sentence level, the level below that the word level, the next the state level.

Our goal was to provide a bidirectional conversion tool that could systematically convert to IHD, i.e., down to the phone and state level, so that recognition experiments could be performed completely in JSGF. While not all recognition systems supporting alternate grammar formats provide this capability, we believed it was an important feature to provide our users to reduce development efforts and experimental setup time. We also required the tool to provide the same level of conversion in reverse.

The subtle but important distinctions between JSGF and XML-SRGS and other CFG-based language models have

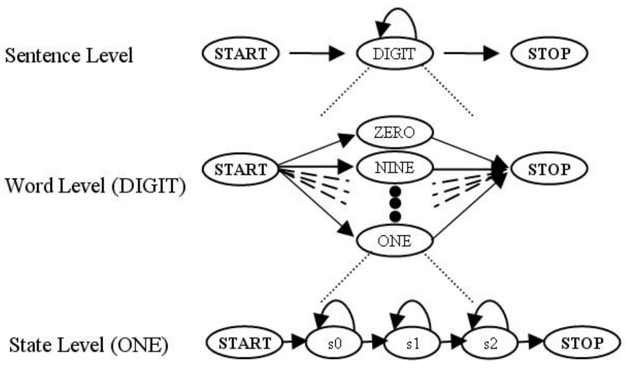

Figure 1: IHD Hierarchical Format

[1] This material is based upon work supported by the National Science Foundation (NSF) under Grant No. IIS-0414450. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the NSF.

proved challenging to the task of developing general purpose conversion tools. However, our deepened understanding of these nuances led to the design of more robust conversion tools for these and future grammar formats. Section 2.1 presents key theoretical and syntactic features of each format, both similarities and differences.

### 2.1 BNF and EBNF

First, JSGF and XML-SRGS are theoretically equivalent in expressive and computational power, both adhering in principle to Backus-Naur Form (BNF) [6], a formal notation for CFGs, more specifically to two equivalent variants, Extended BNF (EBNF) [7] and Augmented BNF (ABNF). Many detailed descriptions of BNF and its variants exist. We briefly introduce key features relevant to our discussion.

Stated simply, BNF defines a method for describing production rules in a CFG, including terminal and non-terminal symbols for rules, and a selection of alternatives among rules. Though numerous variants of the syntax exist, an example rule in a BNF grammar might be:

```
<A> ::= <B>|c
```

where non-terminals are represented in capital letters, A, B, surrounded by brackets <> and can appear on the lefthand side (LHS) or righthand side (RHS) of the rule demarcated by the := symbol. Terminals are often expressed in lower case (though not required), but more importantly can appear only on the rule RHS. Finally, selection or branching among alternative rule definitions is expressed by the | symbol.

BNF also allows the use of recursive rules in a grammar. Such rules directly or indirectly reference themselves. An example of direct recursion might be: <A> ::= a<A>. The use of directly or indirectly recursive rules is useful to represent repetitive actions in an FSM. Consider the simple FSM in Figure 2. This FSM recognizes the regular expression, a(bc)+ that could be represented in BNF with the production rules:

```
<S> ::= <A>
<A>::= aB
<B> ::= bc|B
```

The use of the non-terminal B on the RHS in rule 3 is recursive and indicates that subgraph bc is a cycle that can be repeated one or more times. However, for simple regular expressions such as this, the cycle in this FSM could be represented using the Kleene + operator, a standard notation for regular expressions which denotes 1 or more repetitions. EBNF extends BNF to supports the use of structures, such as the * and + for repetition as well as others. (EBNF also has many variants, but its origins date to [7].) This allows creating a more intuitive set of production rules for regular expressions, so that rules A and B above can be reduced to:

```
<A> ::= a (bc)+
```

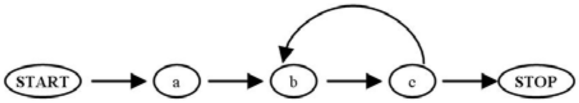

Figure 2: FSM for regexp a(bc)+

### 2.2 JSGF and XML-SRGS

Again, both JSGF and XML-SRGS provide expressive equivalence to EBNF. They differ, however, in syntax. As an example, the above rule could be represented in JSGF as:

```
<A> = a(bc)+;
```

Note that the + operator is supported directly as well as the use of parentheses. Consider the same rule in XML- SRGS:

```
<rule>
   <item> a </item>
   <item repeat='1-'>
        <item> b</item>
        <item> c </item>
    </item>
  </rule>
```

The <rule> tag marks this as a production rule; no non-terminal symbols are needed in this example; the terminal symbols are marked with <item> tags, and "repeat= '1-'" before the non-terminal b denotes the Kleene + operation (1 or more repetitions) applied to the concatenation of b and c, shown by listing each non-terminal in sequence with surrounding item tags and a close item tag for repeat '1-'.

Clearly, the JSGF syntax is more similar to EBNF than XML-SRGS. The differences are due in large part to their origins: XML was designed initially as a markup language for general Internet usage and later modified to provide support for spoken language; JSGF was designed from the outset to support spoken language applications. The W3C SRGS attempted to address these issues first by using JSGF as a theoretical model in defining the XML-SRGS and second, by developing a standard specification for ABNF[4]. ABNF is an EBNF variant, with origins dating back to Arpanet. Any ABNF-SRGS can be mapped to XML-SRGS. The previous example could be written as shown below, using *1 for + before the item (bc) to be repeated:

```
<A> = a *1 (bc)
```

The subtle distinctions among syntax complicated the task of identifying underlying theoretical structures. A more interesting set of issues arose with respect to recursion. While a conformant JSGF grammar processor must provide support for recursive rules, this support is optional for the XML-SRGS upon which we based our conversion. We chose, however, to support recursion for all our grammar format conversion tools.

### 2.3 Recursion in Speech Recognition

The example in Figure 2 illustrates how a lexical construct can be represented more simply with a regular expression than a recursive CFG. However, the converse situation often arises in speech recognition: though a lexical or syntactic construct could be described by a complex regular expression, the recursive form may be simpler to produce algorithmically. Figure 3 shows such an example.

Though still a relatively simple graph, notice the branches, cycles within branches, and self-loops not present in Figure 2. The regular expression for the strings accepted by this FSA can be written as: (ab)+e+|a(ba)*cde+. A rule for this can be written in EBNF as follows:

S ::= (ab)+e+ | a(ba)*cde+

Production rules can also be written for the expression using non-terminals, direct recursion for the self-loop, and indirect recursion for the cycle by creating a non-terminal for each state in the graph, e.g., A for a, as follows:

```
S::= A            C ::= c D
A ::= a (B|C)     D ::= d E
B ::= b (A|E)     E ::= e E | ET
                  T::= t
```

While the single rule in the first grammar eases visualizing a legal input string directly from the rules, the second grammar more directly expresses actions to be encoded in an algorithm with limited lookahead, e.g., rule A states that on input ‘a’, branch to either state B or C, while the indirectly recursive rule B states on input ‘b’, branch back to A or ahead to E.

The use of recursive rules is an attribute that distinguishes regular grammars and CFGs. Although XML and JSGF both provide the expressive power of CFGs, only JSGF requires support for recursion. There are restrictions on the type of recursion supported, however. Notice that the recursive grammar for Figure 3 is right recursive, that is, all recursive references appear on the rightmost side of the RHS. JSGF limits its required support to these type grammars. Several arguments can be made in favor of this restriction. First, any right recursive rule can be rewritten using the Kleene * and + operators where more appropriate. Second, speech recognizers typically use regular grammars, which must be either left or right linear, and thus can contain only left or right recursion. Finally, right recursive grammars can be parsed by top-down parsers with limited lookahead which are arguably among the simplest to construct.

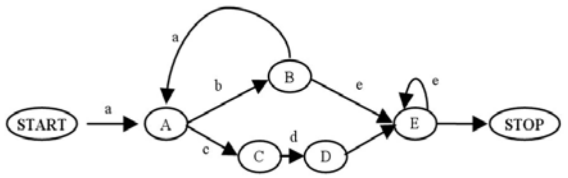

Figure 3: FSM (ab)+e+|a(ba)*cde+

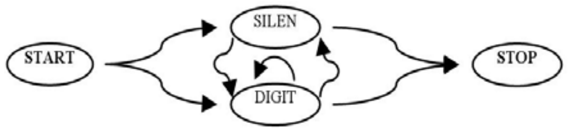

Figure 4: FSM for Digits Grammar

A final example of recursion taken from our speech recognition system further explains our decision to implement support for both the EBNF grammar structures, e.g., * and +, as well as right recursive rules. To provide conversion tools that would allow use of XML-SRGS and JSGF all the way from the sentence level down to the acoustic level, we must support writing and parsing grammars that represent layers of FSMs.

Consider a sentence level view of a simple digits grammar in Figure 4. Assuming SILEN and DIGIT are terminal symbols, a simple regular expression (SILEN DIGIT*)+ can describe this FSM. However, SILEN and DIGIT are actually non-terminals describing inputs that are modeled at lower levels. It is possible to write production rules containing regular expressions for each of the sentence, word and state levels. Our system, however, treats non-terminals on arcs as subroutines, saving the current location in the graph, processing the non-terminals, and then returning when parsed. This meant that our conversion tool for IHD->XML could easily generate grammars with either type of structure, e.g., *, + for self-loops and cycles, or recursive rules. This further reinforced our decision to support both recursive rules and EBNF extensions.

## 3. CONVERSION REDESIGN

Other specific features of XML complicated the design decisions described in the previous section.

### 3.1 XML-SRGS Weights and Probabilities

Two methods are available for specifying weights in XML-SRGS: the *weight* attribute and the *repeat-prob* attribute. It is important to note that a repeat probability is a different logical entity from a weight. It is the probability that a given loop will repeat, while a weight can only be transformed into a probability when compared with the other weights leaving a node. The FSM in Figure 5 is used to illustrate both methods. An XML-SRGS grammar using both the *weight* attribute and the *repeat-prob* attribute is shown below:

```
<grammar>
    <rule>
            <item> a </item>
            <one-of>
                    <item weight='4'> b </item>
            </one-of>
            <item repeat='1-' repeat_prob='.43'> c </item>
    </rule>
</grammar>
```

The first item in this grammar, a, has no weight since its incoming arc has no weight. The second item, b, has a weight on its incoming arc. SRGS dictates that an item may not have a weight unless its immediate enclosing tag is a <one-of>. SRGS does, however, allow for a single item to

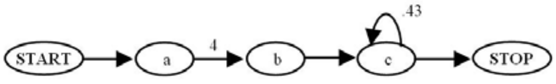

Figure 5: FSM for XML Weights and Probabilities

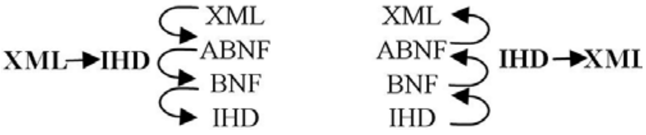

Figure 6: Conversion Redesign

be enclosed in a <one-of> in order to specify a weight where one would not otherwise be allowed. Since speech recognition systems typically have weights on all arcs, this limitation on the SRGS *weight* attribute is significant.

Also important, the *repeat-prob* attribute can only be used with the SRGS *repeat* looping attribute. Recall this attribute implements the EBNF loop extensions for Kleene operations. In this example, “repeat = ‘1-‘” is equivalent to +. An additional limitation is that repeat probability values must lie between 0.0 and 1.0.

The above grammar would be represented as shown below without repeat probabilities, but with recursive rules:

```
<grammar>
    <rule>
      <item> a </item>
       <one-of>
           <item weight='4'> <ruleref uri="#B"/> </item>
       </one-of>
      </rule>
      <rule id="B">
      <item> b </item>
             <item> <ruleref uri="#C"/> </item>
      </rule>
      <rule id="C">
       <item> c </item>
       <one-of>
           <item weight='.43'> <ruleref uri="#C"/> </item>
           <item> <ruleref special="NULL"> </item>
      </one-of>
      </rule>
</grammar>
```

This avoids the use of a repeat probability on node ‘c’ by putting a weight on the recursive rule reference at the end of this grammar. However, special care must be taken in converting weight and repeat-prob attributes consistently.

### 3.2 ABNF and BNF Conversion Modules

Once the underlying theoretical structures of each format were understood in detail, producing a verifiably robust conversion tool would require additional stages of conversion in our process. The first stage would create a common EBNF grammar format to which any other format could be converted. The ABNF-SRGS was an obvious choice as the common EBNF format. The next stage would entail processing the EBNF to remove extensions and thus standardize representation of weights and probabilities. We redesigned our conversion process to include the following steps and corresponding software modules: 1) convert the XML to an equivalent ABNF, 2) convert the ABNF to remove the EBNF extensions and produce a clean BNF with or without recursion, 3) convert the clean BNF to IHD. We also created modules to implement the steps shown above in the reverse conversion from IHD -> XML. The redesigned conversion process is shown in Figure 6.

## 4. SUMMARY AND CONCLUSIONS

Supporting popular CFG-based language model formats has been an important priority in our research. The important nuances of each CFG format implementation have presented equal challenges to producing robust conversion tools. Further, assignment of weights and probabilities for these styles must be carefully considered in converting to any other formats, including CFG or regular grammars. We have addressed these issues by incorporating additional stages and corresponding software modules in our process which perform generic conversions to and from common EBNF and BNF grammar formats. These enhancements have advanced an important goal for the speech research community — producing verifiably robust conversion tools to support popular CFG-based language model standards.

## 5. REFERENCES

[1] Java Speech Grammar Format Specification, Version 1, Sun Microsystems Developer Network, October 26, 1998 (see http://java.sun.com/products/java-media/speech/forDevelopers/ JSGF/JSGF).

[2] Hunt, A. and McGlashan. S., Eds., W3C Speech Recognition Grammar Specification Version 1.0, March 16, 2004 (see http://www.w3.org/TR/speech-grammar/).

[3] Picone, J., *et al*., “A Public Domain C++ Speech Recognition Toolkit,”, ISIP, Mississippi State University, Mississippi State, MS, USA, March 2003.

[4] Chomsky, N., “On Certain Formal Properties of Grammars,” *Info. and Control*,, Vol. 2, 1959, pp. 137-167.

[5] Mohri, M, “Weighted Grammar Tools: The GRM Library,” in J.C. Junqua and G. van Noord (eds), *Robustness in Language and Speech Technology,* .Kluwer Academic Publishers, 2000.

[6] Naur, P. “Revised Report on the Algorithmic Language Algol 60,” *Com. of the ACM*, Vol. 7, No. 12, pp. 735–736, 1963.

[7] Wirth, N. “What Can We Do About the Unnecessary Diversity of Notation for Syntactic Definitions,” *Com. of the ACM*, Vol. 20, No. 11, pp. 822–823, 1977.