Grammar Reference
# Table of Contents

**Table of Contents** .......................................................................................... 3  
**Preface** ............................................................................................................. 1  
  Audience ............................................................................................................. 1  
  Conventions ......................................................................................................... 1  
  References ........................................................................................................... 2  
  1. Using VoiceXML Grammars ............................................................................. 3  
     Referencing Grammars .................................................................................... 4  
     Built-in Grammars .......................................................................................... 4  
     Inline Grammars ............................................................................................ 4  
     External Grammar Files .................................................................................. 4  
     VoiceXML `<grammar>` element ................................................................... 6  
     Scope ................................................................................................................ 7  
     Grammar Formats ............................................................................................ 7  
     Universal Grammars ....................................................................................... 8  
     Active Grammars and Grammar Scope .......................................................... 8  
     Setting Input Variables .................................................................................... 9  
     Without a Semantic Interpretation ................................................................ 9  
     With a Semantic Interpretation ..................................................................... 9  
     Slot Names ...................................................................................................... 10  
     Field-Level Grammars ................................................................................... 10  
     Form Grammars ............................................................................................. 11  
     Ambiguous Grammars .................................................................................... 13  
     Universal Grammars ...................................................................................... 13  
     Predefined Universal Grammars .................................................................. 13  
     Application-Defined Universal Grammars ................................................... 13  
     Activating Universal Grammars .................................................................... 14  
     Responding to Application-Specific Universal Grammars ............................ 14
# Table of Contents

## 2. Using Built-in Grammars .............................................. 15
- Referencing a Built-in Grammar ........................................ 16
- Semantic Interpretations ................................................. 16
- Parameterizing Grammars ................................................ 19
- Extended Built-in Grammars .............................................. 20

## 3. Defining Grammars ..................................................... 21
- Grammar Construction ..................................................... 22
  - Header versus Body ..................................................... 22
  - Inline Definition versus External Grammar Files ................... 22
  - Rule Definitions ....................................................... 23
  - Rule Expansions ....................................................... 23
    - Tokens ........................................................................ 23
    - Rule References ...................................................... 23
    - Semantic Interpretations .......................................... 23
    - Combinations ......................................................... 24
- Compiled Grammar Files .................................................. 24
  - Using the BeVocal Café Grammar Compiler ......................... 24
  - Using Nuance’s Grammar Compilation Tool ......................... 25

## 4. XML Speech Grammar Format ......................................... 27
- XML Tags ......................................................................... 27
- Differences between Inline and External Definitions ................. 28
- Comments ........................................................................ 28
- Header ............................................................................ 28
  - XML Prolog .................................................................... 29
  - Root Element ................................................................... 30
  - Grammar Version ......................................................... 30
  - Language ........................................................................ 30
  - Grammar Mode ............................................................ 31
  - Root Rule ....................................................................... 31
  - Tag Format ...................................................................... 32
  - Base URI ........................................................................ 32
  - Pronunciation Lexicon .................................................. 32
  - Meta and HTTP-Equiv. .................................................. 33
  - Metadata ........................................................................ 33
- Rule Definitions ............................................................... 33
- Basic Rule Definition ....................................................... 33
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rule Recursion</td>
<td>54</td>
</tr>
<tr>
<td>Example Phrases</td>
<td>54</td>
</tr>
<tr>
<td>Rule Expansions</td>
<td>55</td>
</tr>
<tr>
<td>Language</td>
<td>55</td>
</tr>
<tr>
<td>Tokens</td>
<td>55</td>
</tr>
<tr>
<td>Voice Tokens</td>
<td>56</td>
</tr>
<tr>
<td>DTMF Tokens</td>
<td>56</td>
</tr>
<tr>
<td>Rule References</td>
<td>57</td>
</tr>
<tr>
<td>Local Reference</td>
<td>57</td>
</tr>
<tr>
<td>External Reference</td>
<td>57</td>
</tr>
<tr>
<td>Combinations</td>
<td>58</td>
</tr>
<tr>
<td>Alternatives and Weights</td>
<td>58</td>
</tr>
<tr>
<td>Sequence</td>
<td>59</td>
</tr>
<tr>
<td>Grouping</td>
<td>59</td>
</tr>
<tr>
<td>Repeats</td>
<td>60</td>
</tr>
<tr>
<td>Tags (Semantic Interpretations)</td>
<td>61</td>
</tr>
<tr>
<td>Precedence</td>
<td>61</td>
</tr>
<tr>
<td>6. Nuance GSL Grammar Format</td>
<td>63</td>
</tr>
<tr>
<td>Differences between Inline and External Definitions</td>
<td>63</td>
</tr>
<tr>
<td>Comments</td>
<td>64</td>
</tr>
<tr>
<td>Reserved Words</td>
<td>64</td>
</tr>
<tr>
<td>Header Declarations</td>
<td>65</td>
</tr>
<tr>
<td>Self-Identifying Header</td>
<td>65</td>
</tr>
<tr>
<td>Language</td>
<td>65</td>
</tr>
<tr>
<td>Grammar Mode</td>
<td>65</td>
</tr>
<tr>
<td>Root Rule</td>
<td>65</td>
</tr>
<tr>
<td>Tag Format</td>
<td>65</td>
</tr>
<tr>
<td>Pronunciation Lexicon</td>
<td>66</td>
</tr>
<tr>
<td>Meta and HTTP-Equiv.</td>
<td>66</td>
</tr>
<tr>
<td>Rule Definitions</td>
<td>66</td>
</tr>
<tr>
<td>Basic Rule Definition</td>
<td>66</td>
</tr>
<tr>
<td>Rule Names</td>
<td>66</td>
</tr>
<tr>
<td>Special Rules</td>
<td>67</td>
</tr>
<tr>
<td>Scoping of Rule Definitions</td>
<td>67</td>
</tr>
<tr>
<td>Rule Recursion</td>
<td>68</td>
</tr>
<tr>
<td>Rule Expansions</td>
<td>69</td>
</tr>
<tr>
<td>Language</td>
<td>69</td>
</tr>
</tbody>
</table>
Tokens .................................................................................................................. 69
  Voice Tokens ....................................................................................................... 69
  DTMF Tokens ...................................................................................................... 70
Rule References ...................................................................................................... 70
  Local Reference ................................................................................................... 71
  External Reference .............................................................................................. 71
Combinations .......................................................................................................... 72
  Alternatives ......................................................................................................... 72
  Sequence ............................................................................................................... 72
  Repeats ................................................................................................................ 72
  Weights or Probabilities ...................................................................................... 73
Assignment Commands (Semantic Interpretations and Rule Variables) .............. 73
  Slot-Filling Command Example ....................................................................... 74
  Return Command Example .............................................................................. 74
  Literal Values ...................................................................................................... 75
  Variable Expressions ......................................................................................... 75
  Function Calls .................................................................................................... 76

7. JSGF Grammar Format ...................................................................................... 79

8. Nuance SayAnything Grammars ...................................................................... 81
  Introduction to SayAnything Grammars ........................................................... 81
  SLM Grammars .................................................................................................. 81
  Robust Natural Language Interpretation .......................................................... 82
  Controlling SayAnything Grammars ................................................................. 83

9. Voice Enrollment Grammars ............................................................................ 85
  Text-Based Grammars ....................................................................................... 85
  Limitations of Text-based Grammars ................................................................ 86
  Enrollment Basics ............................................................................................... 87
  Usage Model ....................................................................................................... 87
  Security Considerations ..................................................................................... 88
  Enrollment Example .......................................................................................... 90
  Recognizing Enrolled Grammars ...................................................................... 91
  Recognition Example .......................................................................................... 92
  Deleting Enrolled Grammars ............................................................................ 93
VoiceXML applications use grammars to specify sets of valid user utterances at particular points in an interaction with the application. For example, at the beginning of your application, you may ask the user to select among a set of predefined options. In your VoiceXML document, you’ll use a grammar to identify the set of possible things a user can say for each option. The speech-recognition engine uses the grammar to identify which option the user is selecting.

This document provides information on how to use a grammar within your VoiceXML application and how to define grammars in various formats.

**Audience**

This document is for software developers using BeVocal VoiceXML. It assumes that you are familiar with the basic concepts of HTML and that you already have some familiarity with VoiceXML authoring.

**Conventions**

*Italic* font is used for:

- Introducing terms that will be used throughout the document
- Emphasis

**Bold** font is used for:

- Headings

*Fixed width* font is used for:

- Code examples
- Tags and attributes
- Values or text that must be typed as shown

*Italic fixed width* font is used for:

- Variables
- Prototypes or templates; what you actually type will be similar in format, but not the exact same characters as shown
References

For additional or related information, you can refer to:


  [http://www.voicexml.org](http://www.voicexml.org) under Specs for PDF.  
  [http://www.w3c.org/TR/voicexml20/](http://www.w3c.org/TR/voicexml20/) for HTML

- **Nuance Grammar Developer’s Guide.** Nuance.  
  Under developers documentation after logging on at the Nuance web site  
A grammar identifies different words or phrases that a user might say and (optionally) specifies how to interpret a valid expression in terms of values for input variables. Grammars can range from a simple list of possible words to a complex set of possible phrases.

Each field in a form can have a grammar that specifies the valid user responses for that field. An entire form can have a grammar that specifies how to fill multiple input variables from a single user utterance. Each choice in a menu has a grammar that specifies the user input that can select the choice.

A grammar contains one or more **rules** that specify matching input, usually with one rule specified as the **root rule** of the grammar. If the grammar has a root rule, then you can use the grammar in your VoiceXML application without naming which rule to start from.

In some grammars, there is exactly 1 top-level rule that can be used as a starting point for that grammar. For example, a simple yes/no grammar might consist of a single rule, allowing "yes", "no", and various synonyms, such as "yep", "nope", or "no way". In this grammar, the root rule would, of course, be that one starting point.

A larger or more complex grammar, however, may have several rules that can be used as a starting point. For example, consider a grammar for recognizing marine animals. It could have 1 rule that recognizes all marine animals. That rule might itself be composed of rules that recognize smaller sets of animals, such as one for marine mammals, another for types of coral, and a third for species of fish. This marine animals grammar might allow you to specify one of these subrules as the starting point, instead of always using the complete grammar. To use the subrules, you’d have to ask for the rule by name. This grammar might still identify one rule as the root rule, so that you could ask for the grammar without specifying a rule by name.

Your VoiceXML application specifies a grammar to use with the VoiceXML `<grammar>` tag. It can use built-in grammars and application-defined grammars.

A **built-in** grammar is one that is built directly into the VoiceXML interpreter. You can use these grammars without any coding effort.

An **application** grammar, on the other hand, is one that a developer defines from scratch. An application grammar may be a grammar you’ve defined specifically for a particular application or it may part of a general library of grammars you have access to for reuse in multiple VoiceXML applications.

Application grammars can either be **inline** or **external**. The entire definition of an inline grammar appears directly in the `<grammar>` element of the VoiceXML document; the definition of an external grammar appears in a separate file.

If the `<grammar>` element contains content, that content is the definition of an inline grammar. If the element does not contain content, it must have a value for either the `src` attribute or the `expr` attribute. In this case, depending on the value of that attribute, the reference is to either a built-in grammar or an external grammar file.

This chapter is concerned with how you use a grammar in your VoiceXML document and what the interpreter does with the grammar in later processing. This chapter has the following sections:

- Referencing Grammars
- VoiceXML `<grammar>` element
- Active Grammars and Grammar Scope
- Setting Input Variables
- Ambiguous Grammars
Universal Grammars

The rest of this document contains the following chapters:

- Chapter 2, “Using Built-in Grammars” describes built-in grammars and how to use them.
- Chapter 3, “Defining Grammars” describes general information on how to build and use your own grammars.
- Chapter 4, “XML Speech Grammar Format”,
  Chapter 5, “ABNF Grammar Format”,
  Chapter 6, “Nuance GSL Grammar Format”, and
  Chapter 7, “JSGF Grammar Format”
  describe the individual grammar formats you can use to write your grammar—the XML form of the
  W3C Speech Recognition Grammar Format, the Augmented BNF (ABNF) form of the W3C Speech
  Recognition Grammar Format, the Nuance Grammar Specification Language (GSL), and the Java
  Speech Grammar Format (JSGF).
- Chapter 8, “Nuance SayAnything Grammars” describes how to use Nuance Say Anything grammars.
- Chapter 9, “Voice Enrollment Grammars” describes how to create and use a special type of grammar
  that directly captures a user’s utterance for a grammar. You use these grammars from within an ABNF
  grammar.

Referencing Grammars

You reference a grammar with the VoiceXML <grammar> element. You use a different syntax to reference
each type of grammar.

Built-in Grammars

A built-in grammar is defined directly by the VoiceXML interpreter. The primary way your application
references a built-in grammar is from the <grammar> element, by providing a value for the src attribute
using one of the following formats:

\[
\text{builtin:grammar/typeName}
\]
\[
\text{builtin:grammar/typeName?parameters}
\]

Chapter 2, “Using Built-in Grammars” describes alternative ways to specify a built-in grammar in some
circumstances.

Inline Grammars

If a VoiceXML <grammar> tag does not have a value for either the src or the expr attribute, it is an inline
grammar and must have child elements that constitute a valid grammar definition (as described in later
chapters).

By its very nature, you do not reference an internal grammar, it is simply included where it’s needed.

External Grammar Files

If a VoiceXML <grammar> tag has a value for either the src or the expr attribute, it is an external
grammar and the attribute must specify the URI of an external grammar file (src) or be a JavaScript
expression that evaluates to such a URI (expr). If the tag has one of these attributes, it cannot also have
child elements.

Providing a value for the src or expr attribute is called referencing the grammar. As we’ll see later, you
can also reference a grammar from within a grammar. When a grammar is referenced by a VoiceXML
document or another grammar, there must be a way for the VoiceXML interpreter to decide which rule in
the grammar to use as its starting point in that grammar. The reference may explicitly name a rule to use; if it does so (and that rule is available in the grammar), then the interpreter uses the named rule. On the other hand, the reference may only name a grammar and not name a rule within the grammar. In this case, the grammar must itself identify a rule for the interpreter to start from. The rule that the grammar identifies is called the root rule of the grammar.

The URI specified by src or expr can be of one of the following general formats:

<table>
<thead>
<tr>
<th>Syntax</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>GrammarFileURI#RuleName</code></td>
<td>Grammar in a grammar file of any format, using a specified rule to start recognition.</td>
</tr>
<tr>
<td><code>GrammarFileURI</code></td>
<td>Grammar in a grammar file of any format, using the grammar’s root rule to start recognition. If you use this format, the grammar file must specify its root rule.</td>
</tr>
<tr>
<td><code>compiled:grammar/key</code></td>
<td>Grammar in a GSL compiled grammar file; the file identifies its root rule from which to start recognition.</td>
</tr>
</tbody>
</table>

For example, assume `colors.gram` contains 3 rules, Colors, Shades, and ShadeAndColor, where the ShadeAndColor rule is defined in terms of the two other rules and the external grammar file identifies ShadeAndColor as the root rule of the grammar. (Later chapters will discuss how to create such a grammar file.) With this grammar, the paint field can access the ShadeAndColor rule as follows:

```xml
<field name="paint">
  <grammar src="colors.gram"/>
  ...
</field>
```

When this field becomes active, the interpreter uses ShadeAndColor as the top-level rule for recognizing input. This rule itself accesses the other rules in the grammar. On the other hand, the color field can directly use the Colors rule by specifying it as the rule to start from:

```xml
<field name="color">
  <grammar src="colors.gram#Colors"/>
  ...
</field>
```

In this case, the interpreter uses the colors.gram file and uses the rule in that file named Colors as its starting point. It is an error if colors.gram does not contain a rule named Colors.

You can specify the URI in either absolute or relative terms. For example, a VoiceXML document `http://myCompany.com/myvxml.vxml` could use any of the following URIs to refer to the same grammar file:

- Absolute
  ```xml
  http://myCompany.com/vxml/mygram.gram
  ```
- Relative to the host:
  ```xml
  /vxml/mygram.gram
  ```
  Note the initial forward slash.
- Relative to the location of the VoiceXML document:
  ```xml
  vxml/mygram.gram
  ```
  Note the lack of the initial forward slash.
VoiceXML <grammar> element

You use a <grammar> element in your VoiceXML document to specify a grammar to be used within some VoiceXML tag such as <field>, <form>, or <link>. The <grammar> element serves two primary purposes:

- To define the grammar or point to a predefined grammar
  
  The <grammar> element can either contain the entire grammar definition directly, point to an external grammar file in one of several formats, or point to a built-in grammar. “Referencing Grammars” above described the differences in the <grammar> element for identifying where to locate the grammar definition. In addition, if the grammar definition is inline and is in the XML format, then the <grammar> element allows extra attributes to support the grammar definition; these attributes are ignored for an inline grammar in any other format and they are ignored for all external grammars. These special attributes are described in Chapter 4, “XML Speech Grammar Format”.

- To specify aspects of how to use the grammar in the particular containing VoiceXML element.
  
  For this purpose, the <grammar> element supports a set of attributes you can use with any application grammar. You can use these attributes for any grammar format and for both external and internal grammars. You cannot use these attributes for built-in grammars.

The grammar attributes are as follows:

<table>
<thead>
<tr>
<th>Available when</th>
<th>Attributes</th>
<th>For information, see</th>
</tr>
</thead>
<tbody>
<tr>
<td>On a VoiceXML &lt;grammar&gt; element. These identify a built-in or external grammar.</td>
<td>src, expr</td>
<td>“Referencing Grammars” above</td>
</tr>
<tr>
<td>&lt;grammar&gt; element that defines an inline XML grammar. These are part of defining the grammar. The other grammar definition formats do not use attributes to specify this information.</td>
<td>fetchhint, fetchtimeout, maxage, maxstale, caching (VXML 1.0 only)</td>
<td>Chapter 4, “Fetching and Caching Resources” of the VoiceXML Programmer’s Guide</td>
</tr>
<tr>
<td>On a VoiceXML &lt;grammar&gt; element for all application grammars. These are for specifying how to use a grammar, once defined. They are applicable to all grammars except built-in grammars.</td>
<td>scope, type, universal</td>
<td>next</td>
</tr>
<tr>
<td>On the &lt;grammar&gt; element in an external XML grammar file and on the &lt;vxml&gt; element of a VoiceXML document. These are for specifying XML information about the document as a whole. They only make sense on the root element of a VoiceXML or XML document.</td>
<td>xmlns, xmlns:xsi, xsi:schemaLocation</td>
<td>Chapter 4, “XML Speech Grammar Format”</td>
</tr>
<tr>
<td>Never (not implemented)</td>
<td>weight</td>
<td></td>
</tr>
</tbody>
</table>
Scope

The \textit{scope} attribute sets the scope of a form grammar; you cannot set this attribute for other grammars. See “Active Grammars and Grammar Scope” below for information about grammar scopes. The possible values are:

- \textit{document}—The grammar is active throughout the current document. If the document is the application root document, then it is active throughout the application (application scope).
- \textit{dialog}—The grammar is active throughout the current form.

\textbf{Note:} Some grammar formats have the notion of the scope of an individual rule in the grammar. That scoping is separate from the scope of the grammar as a whole.

Grammar Formats

The \textit{type} attribute optionally specifies the MIME type of the grammar; that is, it specifies the grammar’s format. If present, this attribute is only used as a last resort by the interpreter. If the interpreter can determine a grammar’s format in some other way, it does so.

The currently supported types are:

<table>
<thead>
<tr>
<th>Type</th>
<th>Grammar</th>
</tr>
</thead>
<tbody>
<tr>
<td>application/srgs+xml</td>
<td>XML Speech Grammar</td>
</tr>
<tr>
<td>application/srgs</td>
<td>ABNF Speech Grammar</td>
</tr>
<tr>
<td>application/x-nuance-gsl</td>
<td>Nuance GSL</td>
</tr>
<tr>
<td>application/x-nuance-dynagram-binary</td>
<td>Nuance Grammar Object</td>
</tr>
<tr>
<td>application/x-jsgf</td>
<td>Java Speech Grammar Format</td>
</tr>
</tbody>
</table>

In addition, the following types are deprecated, but are currently supported. Support for these values will be removed from a future release.

<table>
<thead>
<tr>
<th>Type</th>
<th>Grammar</th>
</tr>
</thead>
<tbody>
<tr>
<td>application/grammar+xml</td>
<td>XML Speech Grammar</td>
</tr>
<tr>
<td>application/grammar</td>
<td>ABNF Speech Grammar</td>
</tr>
<tr>
<td>application/x-gsl</td>
<td>Nuance GSL</td>
</tr>
</tbody>
</table>

For external grammars, the default type is taken from the \texttt{Content-type} header of the returned file. If not present, the type is inferred from the extension of the URL or from the contents of the grammar (for example, a file beginning with \texttt{<?xml} maps to \texttt{application/srgs+xml}). The recognized extensions are:

<table>
<thead>
<tr>
<th>Extension</th>
<th>Grammar</th>
</tr>
</thead>
<tbody>
<tr>
<td>.grxml, .xml</td>
<td>XML Speech Grammar</td>
</tr>
<tr>
<td>.gram</td>
<td>ABNF Speech Grammar</td>
</tr>
<tr>
<td>.gsl, .grammar</td>
<td>Nuance GSL</td>
</tr>
<tr>
<td>.ngo</td>
<td>Nuance Grammar Object</td>
</tr>
<tr>
<td>.jsgf</td>
<td>Java Speech Grammar Format</td>
</tr>
</tbody>
</table>
For internal grammars, if the grammar definition specifies the grammar type (either directly with a declaration or indirectly by containing XML elements), the interpreter uses that type. If the grammar definition doesn’t indicate the type, the interpreter uses the value of the `type` attribute, if present. Otherwise, the interpreter assumes that the grammar is in GSL format.

Later chapters describe these grammar formats in detail.

**Universal Grammars**

The `universal` attribute is a BeVocal extension that lets you declare the grammar as a "universal" grammar; that is, one that is always active. The string you specify as the attribute value is used as the name of the universal grammar; you use that string to activate and deactivate the universal grammar using the `universals` property. This attribute does not affect the scope of the grammar; it simply assigns it to a universal category.

For more information on universal grammars, see “Universal Grammars” on page 13.

**Active Grammars and Grammar Scope**

The speech-recognition engine uses active grammars to interpret user input. A grammar is active when it is in `scope`. The speech-recognition engine recognizes utterances from any and all active grammars; that is, all grammars that are in scope. There are four scopes:

<table>
<thead>
<tr>
<th>Scope</th>
<th>Grammar is in scope when execution is</th>
<th>Default scope for a grammar defined in</th>
</tr>
</thead>
<tbody>
<tr>
<td>Field</td>
<td>in the <code>&lt;field&gt;</code> where it is defined or referenced.</td>
<td>a <code>&lt;field&gt;</code> element.</td>
</tr>
<tr>
<td>Dialog</td>
<td>in the dialog where it is defined or referenced.</td>
<td>a <code>&lt;form&gt;</code> element or a <code>&lt;choice&gt;</code> element.</td>
</tr>
<tr>
<td>Document</td>
<td>in the document where it is defined or referenced.</td>
<td>a <code>&lt;link&gt;</code> element directly under a <code>&lt;vxml&gt;</code> element.</td>
</tr>
<tr>
<td>Application</td>
<td>anywhere in the application.</td>
<td>any element with document scope that appears in the application root document.</td>
</tr>
</tbody>
</table>

By default, the scope of a grammar is set by the element that contains the grammar:

- If the parent is an input item, the grammar has field scope.
- If the parent is a link, the scope is the element that contains the link.
- If the parent is a menu choice, the grammar scope is specified by the `scope` property of the containing `<menu>` element (or dialog scope by default).

Field scope is the lowest, or narrowest, scope and application scope is the highest, or widest, scope.

For example, a field grammar is active whenever the interpreter is executing that field. A menu-choice grammar is active whenever the interpreter is executing the containing menu. A form grammar is active whenever the interpreter is executing the containing form.

If the interpreter is executing one dialog and the user’s input matches an active grammar for a different dialog, control transfers to the latter dialog. If the user says something that matches a grammar of higher scope, control jumps to the higher-level element that contains the matching grammar. In the case of an event handler, control resumes in the original dialog after the event is handled. If the grammar is in application scope, control might transfer to a dialog in a different document.
A form grammar or the collection of choice grammars in a menu can optionally be made active at higher scopes. In general, you should keep the scope of your grammars as narrow as possible; this allows the speech-recognition engine to be most efficient.

Setting Input Variables

The interpreter uses a grammar to recognize user input. The interpreter then uses the returned information to set appropriate input variables that can be used by other elements in the VoiceXML application. The rules for how the interpreter sets variables are somewhat complicated. For basic information on input variables, see Chapter 1, “Getting Started” in the VoiceXML Programmer’s Guide.

The grammar always returns the text string matched as the utterance. Depending on the grammar, it may also return a semantic interpretation of the utterance. A semantic interpretation is itself a structured object that can provide several pieces of information about what the user said. For example, if the matched text string is "March thirteenth", the semantic interpretation might indicate that the month is "March" and the day of the month is "13".

When the interpreter recognizes a valid utterance against a grammar, it sets the variable application.lastresult$ to a JavaScript object containing information about the recognized utterance. The application.lastresult$.utterance field always contains a text string of the words that were recognized. If the grammar specified a semantic interpretation, the application.lastresult$.interpretation field contains that JavaScript object.

To set input variables, the interpreter uses the semantic interpretation if there is one. If there is not a semantic interpretation, it uses the utterance text string.

Exactly what the interpreter does depends on whether or not there is a semantic interpretation and on whether the grammar was a field-level grammar or a form-level grammar.

Without a Semantic Interpretation

In this case, things are very simple.

If there is no semantic interpretation and recognition was against a form-level grammar, the interpreter does not set any input variables.

If there is no semantic interpretation and recognition was against a field-level grammar, the interpreter sets that field’s input variable to the utterance text string.

For example, if the field-level grammar is a disjunction of a set of simple words, you can use a very simple notation, as shown here (using the GSL format):

```
[  
  january february march april may june july 
  august september october november december 
]
```

If the user says “February”, then the input variable is assigned the value february. If the input variable is date, you can obtain its value in the field’s <filled> element with:

```
<value expr="date">
```

With a Semantic Interpretation

If there’s a semantic interpretation, things are significantly more complicated. The semantic interpretation is a JavaScript object, such as:

```
{
  airline: "aa"
}````
This example object has 3 properties, airline, flight, and freqflyer. Notice that the values of the properties can be a simple text string, another structure, or undefined. We’ll see in a bit how the interpreter uses these different properties. First, however, we need to understand slot names.

Slot Names

Every input item has an associated slot name. The interpreter uses the slot name to extract a part of the full semantic interpretation to use for that input item. If a <field> element has a slot attribute, the value of that attribute is its slot name. For all other input items, and for <field> elements that do not set the slot attribute, the slot name is the value of the name attribute. If neither slot nor name is present, then the slot name is undefined.

For example, with this definition:

```xml
<field name="airline">
```

The slot name is airline and the input variable is airline. But with this definition:

```xml
<field name="carrier" slot="airline">
```

The slot name is airline, but in this case the input variable is carrier.

The interpreter uses the slot name to figure out what value to extract from the semantic interpretation result. It there is a match, it sets the input variable to that value. With the second field definition and the semantic interpretation above, the interpreter uses the slot name airline to extract the value aa from the semantic interpretation and assigns the value aa to the input variable carrier.

Field-Level Grammars

With a field-level grammar, with a given semantic interpretation, the interpreter always sets the input variable for that one field.

Assume that you have a field grammar that recognizes the user utterance “from San Jose to Los Angeles and then on to Phoenix and back.” And that the interpreter returns this semantic interpretation for the recognition:

```json
{
    flight: {
        start: "sjc"
        roundtrip: "yes"
        stops: [ "lax" "phx" ]
        freqflyer: undefined
    }
}
```
The following table shows how the interpreter uses this semantic interpretation to set the input variable for different field definitions.

<table>
<thead>
<tr>
<th>With this field element...</th>
<th>The input variable is set to...</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&lt;field name=&quot;flight&quot;/&gt;</code></td>
<td><code>{ start: &quot;sjc&quot; stops: [ &quot;lax&quot;, &quot;phx&quot; ] }</code></td>
</tr>
<tr>
<td><code>&lt;field name=&quot;itinerary&quot; slot=&quot;flight&quot;/&gt;</code></td>
<td></td>
</tr>
</tbody>
</table>

By default a field is assigned the top-level result property whose name matches the field name. However, if specified, the `slot` attribute overrides the field name for selecting the result property.

In this situation, the input variable is set to a JavaScript object. Your application must handle the components of the object. In the first field, \(<value expr="flight"/>\) evaluates to the object and \(<value expr="flight.start"/>\) and \(<value expr="flight[0]"/>\) both evaluate to "sjc". In the second field, \(<value expr="itinerary"/>\) evaluates to the object and \(<value expr="itinerary[1]"/>\) evaluates to ["lax", "phx"].

Notice that this situation does not take advantage of the VoiceXML form-filling algorithm; if there are still missing slots, the interpreter does not automatically prompt for them. This may be acceptable if your application is prepared to deal with a structured object. Otherwise, you may prefer to use the construct described next.

Also remember that only the field's `name` attribute represents a variable; the `slot` attribute is simply a mapping from the semantic interpretation to a particular input variable. As a result, with the second field element, \(<value expr="flight"/>\) would evaluate to `undefined`.

The slot may be used to select a subproperty of the semantic interpretation. This approach allows you to distribute a single property among a number of fields. Here, \(<value expr="from"/>\) evaluates to `sjc` and \(<value expr="rt"/>\) evaluates to the complex object ["lax", "phx"].

For a field grammar, if the semantic interpretation does not have a top-level property whose name matches the slot name, the interpreter assigns the entire JavaScript object as the value of the input variable.

With the first of these definitions, \(<value expr="itinerary.flight.start"/>\) evaluates to "sjc". With the second, \(<value expr="start.flight.start"/>\) evaluates to "sjc".

### Form Grammars

A form that includes its own grammar is a *mixed-initiative form*. If you are writing a grammar for a mixed-initiative form, you may use multiple slot names, each identifying an input variable to be filled. In this case you can name each field (or set its `slot` attribute) according to the corresponding slot name in the grammar.

Here, the interpreter may set one input variable, many input variables, or none at all.

Assume that you have a form grammar that recognizes the user utterance "I would like to fly American Airlines from San Jose to Los Angeles and then on to Phoenix and back." And that the interpreter returns this semantic interpretation for the recognition:

```javascript
{ airline: "aa" flight: { start: "sjc" roundtrip: "yes" stops: [ "lax", "phx" ] freqflyer: undefined }
```
The following table shows how the interpreter uses this semantic interpretation to set the input variables of different field elements for the form.

<table>
<thead>
<tr>
<th>With this field element...</th>
<th>The input variable is set to...</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&lt;field name=&quot;airline&quot;/&gt;</code>&lt;field name=&quot;carrier&quot; slot=&quot;airline&quot;/&gt;</td>
<td>aa</td>
</tr>
</tbody>
</table>

By default a field is assigned the top-level result property whose name matches the field name. However, if specified, the `slot` attribute overrides the field name for selecting the result property.

In the first case, `<value expr="airline">` evaluates to `aa`; in the second case, `<value expr="carrier">` evaluates to `aa`.

Multiple fields can have the same slot name. If a form grammar sets a value for a particular slot, all fields with that slot name are filled with the value. For example, the form could contain both of the fields shown here; in that case, both input variables would be set.

Also remember that only the field's `name` attribute represents a variable; the `slot` attribute is simply a mapping from the semantic interpretation to a particular field. As a result, if the form had only the second field shown here (with both `name` and `slot` defined), then `<value expr="airline">` would evaluate to `undefined`.

| `<field name="flight"/>`<field name="itinerary" slot="flight"/> | `{ start: "sjc" roundtrip: "yes" stops: [ "lax", "phx" ] }` |

In this situation, the input variable is set to a JavaScript object. Your application must handle the components of the object. In the first of these `<value expr="flight">` evaluates to the object and `<value expr="flight.start">` evaluates to `sjc`. In the second field, `<value expr="itinerary">` evaluates to the object. Again, you can refer to the properties of the JavaScript object either by their names or by their indices. So, with the first field, `<value expr="flight.roundtrip">` and `<value expr="flight[1]">` are equivalent.

This situation does not take advantage of the VoiceXML form-filling algorithm; if there are still missing slots, the interpreter does not automatically prompt for them. This may be acceptable if your application is prepared to deal with a structured object. Otherwise, you may prefer to use the construct described next.

| `<field name="from" slot="flight.start"/>` | "sjc" |
| `<field name="rt" slot="flight.roundtrip"/>` | "yes" |

The slot may be used to select a subproperty of the semantic interpretation. This approach allows you to distribute a single property among a number of fields. Here, `<value expr="from">` evaluates to `sjc` and `<value expr="rt">` evaluates to `yes`.

| `<field name="stops" slot="flight.stops"/>` | ["lax", "phx"] |

The selected property may be a compound object.

| `<field name="freqflyer"/>`<field name="somethingnotthereatall"/> | undefined |

In both of these cases, the semantic interpretation does not include a value for the input variable. In the first case, the slot name occurs, but is set to an undefined value; in the second case, it does not occur at all. In both cases, the interpreter does not set any input variable as a result.
Ambiguous Grammars

A grammar is ambiguous if more than one rule can match a given user utterance. Ambiguous grammars can be a problem if the different rules make different slot assignments. For example:

```abnf
#ABNF 1.0;
root $Cities;
$Cities =
  portland [maine] {city="Portland" state="Maine"} |
  portland [oregon] {city="Portland" state="Oregon"} |
  dallas   [texas]  {city="Dallas"   state="Texas"}
;
```

The `Cities` rule is ambiguous because the utterance “Portland” can match two rules; the `state` slot could be filled either with `Maine` or `Oregon`.

In general, you should avoid using ambiguous grammars. If you choose to use them, you need to enable recognition of multiple interpretations of the user’s speech and implement a mechanism to get user clarification for ambiguous utterances. See Chapter 5, “Using Multiple-Recognition” in the VoiceXML Programmer’s Guide.

Universal Grammars

A universal command is always available—the user can give the command at any point in an interaction. A universal grammar specifies user utterances that can be recognized as a universal command.

Predefined Universal Grammars

The following predefined universal grammars are available to all applications:

<table>
<thead>
<tr>
<th>Grammar</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>help</td>
<td>The user asked for help.</td>
</tr>
<tr>
<td>exit</td>
<td>The user asked to leave the application.</td>
</tr>
<tr>
<td>cancel</td>
<td>The user asked to cancel the prompt that is playing.</td>
</tr>
<tr>
<td>goback</td>
<td>The user wants to retract the last response and go back to an earlier part of the interaction.</td>
</tr>
</tbody>
</table>

If one of these predefined universal grammars is activated and a user utterance matches the grammar, an event of the same name is thrown. For example, a `help` event is thrown when the user says “help.”

Application-Defined Universal Grammars

An application creates its own universal command by defining and enabling a new universal grammar and implementing its response to the command.

To define a universal grammar, set the `universal` attribute in the `<grammar>` tag that defines the grammar for the command. The attribute value is a name that uniquely identifies the grammar among all universal grammars in the application. In the following example, the new universal grammar is named `joke`; the user utterance “Tell me a joke” will be a universal command when this universal grammar is activated.
You should not name your universal grammars help, exit, cancel, goback, or none.

Activating Universal Grammars

An application can activate any of the universal grammars to enable the corresponding universal commands. When a universal grammar is activated, a user utterance that matches the grammar is treated as a universal command.

All universal grammars are deactivated by default. The application can activate some or all universal grammars by setting the universals property. This property specifies which of the universal grammars should be active; all other universal grammars are deactivated.

- Set the universals property to all to activate all universal grammars (both predefined and application-defined):

```xml
<!-- Activate help, exit, cancel, and goback -->
<property name="universals" value="all"/>
```

- Set the universals property to a space-separated list of grammars to activate some universals and deactivate others:

```xml
<!-- Activate only help, goback, and joke -->
<property name="universals" value="help goback joke"/>
```

- Set the universals property to none to deactivate all previously activated universal grammars in the current scope.

```xml
<!-- Deactivate all universal grammars -->
<property name="universals" value="none"/>
```

**Note:** *(VoiceXML 1.0 only)* If the `<vxml>` tag's version attribute is 1.0, all universal grammars are activated by default.

Responding to Application-Specific Universal Grammars

A `<link>` element containing a universal grammar implements the application's response to the corresponding universal command. Your application can respond to the command in whatever manner is appropriate. Typically, the response is to throw an event or to transition to a different form.

If you throw an application-specific event, you must provide an event handler to take the appropriate action. For example:

```xml
<!-- Throw an event when the command is given -->
<link event="joke">
<!-- Define the universal grammar (GSL format) -->
<grammar universal="joke">
  (tell me a joke)
</grammar>
</link>

<!-- Invoke a subdialog when the event is thrown -->
<catch event="joke">
  <subdialog name="joker" src="telljoke.vxml"/>
</catch>
```
The VoiceXML 2.0 specification includes a set of built-in grammars as a convenience to enable developers to get started writing more complex VoiceXML applications quickly. The following basic grammars are built into all standard VoiceXML interpreters:

<table>
<thead>
<tr>
<th>Grammar Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>boolean</td>
<td>Recognizes a positive or negative response.</td>
</tr>
<tr>
<td>currency</td>
<td>Recognizes an amounts of money, in dollars.</td>
</tr>
<tr>
<td>date</td>
<td>Recognizes a calendar date.</td>
</tr>
<tr>
<td>digits</td>
<td>Recognizes a sequence of digits.</td>
</tr>
<tr>
<td>number</td>
<td>Recognizes a number.</td>
</tr>
<tr>
<td>phone</td>
<td>Recognizes a telephone number adhering to the North American Dialing Plan (with no extension).</td>
</tr>
<tr>
<td>time</td>
<td>Recognizes a clock time.</td>
</tr>
</tbody>
</table>

**Note:** All standard built-in grammars are supported in the Spanish language. Currently there are no extended built-in grammars for Spanish. Neither standard nor extended built-in grammars are currently supported in French Canadian. All extended built-in grammars are supported only in English. If you specify a language other than English and refer to an unsupported built-in grammar, a parse error `error.unsupported.builtin` is thrown.
In addition, BeVocal VoiceXML contains a set of extended built-in grammars, so VoiceXML developers can reference these quite complex grammars which have been tuned over the years by caller usage. The extended grammars are:

<table>
<thead>
<tr>
<th>Grammar Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>airport</td>
<td>Recognizes an airport name or code, such as DFW or Dallas-Fort Worth</td>
</tr>
<tr>
<td>airline</td>
<td>Recognizes an airline name or code, such as AA or American Airlines.</td>
</tr>
<tr>
<td>citystate</td>
<td>Recognizes US city and state names, for example, “Sunnyvale, California”.</td>
</tr>
<tr>
<td>datetime</td>
<td>Recognizes a date and time. Please contact your BeVocal sales representative or <a href="mailto:sales@bevocal.com">sales@bevocal.com</a> for further information on pricing and availability.</td>
</tr>
<tr>
<td>equity</td>
<td>Recognizes a company symbol or full name, such as IBM or Cisco Systems.</td>
</tr>
<tr>
<td>stockindex</td>
<td>Recognizes the names of the major US stock indexes, such as “Nasdaq”.</td>
</tr>
<tr>
<td>street</td>
<td>Recognizes a street name (with or without street number).</td>
</tr>
<tr>
<td>streetaddress</td>
<td>Recognizes a street name and a street number.</td>
</tr>
</tbody>
</table>

**Referencing a Built-in Grammar**

You can reference a built-in grammar in one of 3 ways:

- You can use any built-in grammar (standard or BeVocal VoiceXML extension) in a `<grammar>` element by specifying the `src` attribute with a URI of one of these forms:
  
  ```xml
  builtin:grammar/typeName
  builtin:grammar/typeName?parameters
  ```

  See “Parameterizing Grammars” on page 19 for information on grammar parameters. For example:

  ```xml
  <grammar src="builtin:grammar/number" />
  ```

  This is the preferred method for referencing all built-in grammars.

- You can use any built-in grammar (standard or BeVocal VoiceXML extension) in a `<grammar>` element by specifying the `expr` attribute with a JavaScript expression that evaluates to a URI of one of the forms specified above. For example:

  ```xml
  <var name="numgram" value="'builtin:grammar/number'" />
  <grammar expr="numgram" />
  ```

- You can use a standard built-in grammar as the value of the `type` attribute of a `<field>` element. For example:

  ```xml
  <field name="num" type="number">
  ```

  This means that the speech-recognition engine tries to interpret what the user says as a number.

**Semantic Interpretations**

See “Setting Input Variables” on page 9 for a general description of how the interpreter uses recognition results to fill input variables. Most built-in grammars, such as `phone`, return a simple text string, which is used to fill a single slot whose name is the same as the grammar name. For example, the `date` grammar
sets a slot named `date`. However, some of the more complex grammars provide a semantic interpretation that can fill in multiple slots; for example, airline sets `code` and `name`.

The following table lists the properties for the extended built-in grammars which provide a complex semantic interpretation.

<table>
<thead>
<tr>
<th>Field Type</th>
<th>Properties</th>
</tr>
</thead>
</table>
| airport      | • **code**—Code of the airport, such as `DFW`. For a list of known airport codes, see `Airport & Airline Codes` on the Resources page of the BeVocal Café web site.  
• **spokencity**—The city that was spoken, such as `dallas`. This slot is only returned if a domestic (US) airport is being recognized.  
• **airportcity**—The airport city for the recognized airport, such as `ft. worth`  
• **state**—The two-letter abbreviation for the state, such as `CA`.  
• **country**—The country where the airport is located, such as `united states of america`. |
| airline      | • **code**—The airline’s identifying code, such as `AS`. For a list of known airline codes, see `Airport & Airline Codes`.  
• **name**—Full name of the airline, such as `Alaska Airlines`. |
| citystate    | • **city**—The city, such as `Sunnyvale`.  
• **county**—The county in which the city is located, such as `Santa Clara`.  
• **state**—The two-letter abbreviation for the state, such as `CA`.  
• **datacity**—If the city is unlikely to appear in data feeds for traffic, weather, and so on (for example, because it is tiny or unincorporated), the name of an adjacent city that is more likely to work with data feeds. In all other cases, this property is identical to the `city` property. |
A property is accessed with an expression of the form:

\[ \text{fieldName.propertyName} \]

For example, the `city` property of a `citystate` field is accessed as follows:

```xml
<field name="mycity">
  <grammar src="builtin:grammar/citystate"/>
  <filled>
    <prompt>The city is <value expr="mycity.city"/>
  </filled>
</field>
```
## Parameterizing Grammars

Two standard built-in grammars, `digits` and `boolean`, can be parameterized. Specifically, you can set limits on the length of a digit string, and you can set DTMF key presses to mean yes or no. In addition, you must specify the `city` and `state` with the `street` built-in grammar; you can optionally specify the county to disambiguate a case in which the state contains two cities with the same name.

The following table shows the built-in grammars that can be parameterized and indicates which parameters are required.

<table>
<thead>
<tr>
<th>Grammar</th>
<th>Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>boolean</td>
<td>• y—The DTMF key press for an affirmative answer.</td>
</tr>
<tr>
<td></td>
<td>• n—The DTMF key press for a negative answer</td>
</tr>
</tbody>
</table>
| digits    | • `minlength`—The maximum number of digits in a valid utterance.  
• `length`—The exact number of digits in a valid utterance.  
**Note:** If you do not specify `length` or `maxlength`, the built-in grammar accepts an infinite number of digits. You should use `length` or `maxlength` whenever you can; doing so usually results in more accurate speech recognition. |
| equity    | `symbol`—When true, the grammar expects the input to be the stock symbol spelled-out, as in "eye" "bee" "emm" for IBM.                       |
| street    | • `city`—The city in which the street is located (required).  
• `state`—The state in which the specified city is located (required).  
• `county`—The county in which the specified city is located. If you omit the county and the city name is ambiguous, an `error.badfetch` event is thrown with a list of multiple possible counties in the error message.  
**Note:** An `error.badfetch` event is thrown if a required parameter is not specified, if the city is not a recognized city of the specified state, or if there is no street grammar for the specified city. |
| streetaddress | • `city`—The city in which the street is located (required).  
• `state`—The state in which the specified city is located (required).  
• `county`—The county in which the specified city is located. If you omit the county and the city name is ambiguous, an `error.badfetch` event is thrown with a list of multiple possible counties in the error message.  
**Note:** An `error.badfetch` event is thrown if a required parameter is not specified, if the city is not a recognized city of the specified state, or if there is no street grammar for the specified city. |
| airport   | • `domestic`—If true, then the grammar recognizes only the major domestic (US) airports. If false, then it recognizes only the major international airports. If no parameter is specified, then this property defaults to true. |

You express parameter information using URI-style query syntax of the form:

`builtin:grammar/typeName?parameter=value`

For example, the grammar matches a sequence of exactly five digits:

```
<grammar src="builtin:grammar/digits?length=5"/>
```

You can specify more than one parameter, separated by semicolons. For example, the following grammar allows a user to press 7 for an affirmative answer and 9 for a negative answer:

```
<grammar src="builtin:grammar/boolean?y=7;n=9"/>
```
**Note:** The interpreter throws a `error.badfetch` event if it loads a VoiceXML file that contains a built-in grammar with an unrecognized parameter or with inconsistent parameters, such as:

```xml
<!-- ERROR: Inconsistent parameters for built-in grammar-->
<grammar src="builtin:grammar/digits?minlength=5;maxlength=1">
```

### Extended Built-in Grammars

The following table shows example user inputs for each extended built-in grammar.

<table>
<thead>
<tr>
<th>Grammar Type</th>
<th>Example Inputs</th>
</tr>
</thead>
<tbody>
<tr>
<td>airport</td>
<td>San Jose</td>
</tr>
<tr>
<td></td>
<td>DFW</td>
</tr>
<tr>
<td>airline</td>
<td>American</td>
</tr>
<tr>
<td></td>
<td>UA</td>
</tr>
<tr>
<td>datetime</td>
<td>May 12 2:47pm</td>
</tr>
<tr>
<td>equity</td>
<td>Cisco</td>
</tr>
<tr>
<td></td>
<td>ORCL</td>
</tr>
<tr>
<td>stockindex</td>
<td>Nasdaq</td>
</tr>
<tr>
<td>street</td>
<td>Bordeaux Drive</td>
</tr>
<tr>
<td>streetaddress</td>
<td>1380 Bordeaux Drive</td>
</tr>
<tr>
<td>citystate</td>
<td>Sunnyvale, California</td>
</tr>
</tbody>
</table>

In VoiceXML 2.0 the field’s `type` attribute no longer defines an implicit `<say-as>` type to output the field’s value. Instead it plays the value in normal TTS. You must now use the `type` attribute of `<say-as>` for a type-specific read-out of the value (in TTS).

The `bevocal:mode` attribute of `<say-as>` can be used to define recorded output for some types. See the `<say-as>` tag for details.

<table>
<thead>
<tr>
<th>Grammar Type</th>
<th>Say-As Type</th>
<th>Audio Output</th>
<th>String Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>airport</td>
<td>airport</td>
<td>Airport name</td>
<td>Dallas Fort Worth International Airport (DFW)</td>
</tr>
<tr>
<td>airline</td>
<td>airline</td>
<td>Airline name</td>
<td>American Airlines (AA)</td>
</tr>
<tr>
<td>equity</td>
<td>equity</td>
<td>Company name</td>
<td>ORCL</td>
</tr>
<tr>
<td>stockindex</td>
<td>stockindex</td>
<td>Index name</td>
<td>Nasdaq Composite Index</td>
</tr>
<tr>
<td>street</td>
<td>street</td>
<td>Street name</td>
<td>Bordeaux Drive</td>
</tr>
<tr>
<td>streetaddress</td>
<td>address</td>
<td>Street number and name</td>
<td>1380 Bordeaux Drive</td>
</tr>
<tr>
<td>citystate</td>
<td>citystate</td>
<td>City State</td>
<td>Sunnyvale, CA</td>
</tr>
</tbody>
</table>
An application grammar is one that you define from scratch for your application. Although the built-in grammars are very useful, you typically need to also define grammars of your own. BeVocal VoiceXML supports grammars specified in the following formats:

- XML form of the W3C Speech Recognition Grammar Format
- Augmented BNF (ABNF) form of the W3C Speech Recognition Grammar Format
- Nuance Grammar Specification Language (GSL)
- Java Speech Grammar Format (JSGF)

The following chapters describe each of these formats in detail. This chapter describes general information about grammars you define.

A simple grammar can be defined directly in the VoiceXML document. An inline grammar is defined within the `<grammar>` element itself; an inline grammar is sometimes called an embedded grammar. For example, the following field uses an inline GSL grammar that matches the words “add” and “subtract”.

```xml
<field name="operator">
    <grammar>
        (add subtract)
    </grammar>
    ...
</field>
```

Here, if the user says “add,” the input variable `operator` is set to `add`.

More complex grammars are typically written externally. An external grammar is defined in a file separate from the VoiceXML document file and is referenced by the `src` or `expr` attribute of the `<grammar>` element. For example, the following field uses a grammar rule named `Colors` in an external ABNF grammar defined in the file `partGrammar.gram`:

```xml
<field name="part">
    <grammar src="http://www.mySite/partGrammar.gram#Colors"/>
    ...
</field>
```

The interpreter uses the named rule (`Colors` in this example) as the starting point for recognition. The specified file may include other grammar rules. Depending on the grammar definition, some of these rules (the `public` ones) could also be named as the starting point for recognition. Other rules (the `private` ones) cannot be used as the starting point, but are instead used by other rules in the grammar. Rules used by other rules are sometimes referred to as `subrules` or `subgrammars`.

The grammar for a menu choice can be specified explicitly with a `<grammar>` child of the `<choice>` element. Alternatively, a grammar can be generated automatically from the choice text.

For more information on referencing external grammars, see “Referencing Grammars” on page 4.
Grammar Construction

Although VoiceXML grammars can be written using various grammar formats, those formats share some basic characteristics. This section gives an introduction to this basic information. See the individual format chapters for details on each of these.

Header versus Body

Conceptually, a grammar definition is always divided into a header and a body. The header contains information that is relevant to the grammar as a whole, such as the language or the identification of the root rule. The body contains individual rule definitions.

In the ABNF, JSGF, and GSL formats, the header information is contained in declarations at the start of the grammar definition. In the XML format, the header information is split across the attributes and the initial children of the <grammar> element.

In “VoiceXML <grammar> element” on page 6, we said that the attributes mode, root, tag-format, version, xml:base, and xml:lang are allowed only for XML grammars. These attributes correspond to information in the header of the definition; for the other formats, this information (when available) is instead encoded in the header declarations.

Inline Definition versus External Grammar Files

An inline grammar is defined completely within the <grammar> element in a VoiceXML document. An external grammar, on the other hand, is defined completely in an external file and referenced in the VoiceXML document.

When you include a grammar definition directly in your VoiceXML document, you may have to take special care. ABNF, JSGF, and GSL grammars use special characters not normally permitted in XML documents, such as angle brackets (< and >). For safety, it is helpful to always enclose the grammar rules with <![CDATA[ and ]]> . Without the <![CDATA[ and ]]> , you would need to replace some special characters with their entity equivalents, such as &lt; and &gt; instead of the < and > characters. You’d also need to know exactly which characters to replace. Using CDATA simplifies all this for you.

<grammar ...usage attributes...>
  <![CDATA[
    ...grammar header declarations...
    ...grammar rule definitions...
  ]]>  
</grammar>

On the other hand, if you have an external grammar file in ABNF, GSL, or JSGF format, the contents of that file should not be inside a CDATA section. Also, an external file in one of these formats should not contain a <grammar> element. So, an external grammar file for these formats simply looks like:

...grammar header declarations...
...grammar rule definitions...

XML grammars have their own peculiarities (described in detail in Chapter 4, “XML Speech Grammar Format”). An inline XML grammar puts all of its attributes on the enclosing VoiceXML <grammar> tag. Also, an inline XML grammar inherits the XML prolog of the VoiceXML document; it does not have one of its own. Because the XML grammar format uses XML tags, you do not make a CDATA section out of an inline XML grammar. So, an inline XML grammar looks like:

<grammar ...usage and header attributes...>
  ...grammar header elements...
  ...grammar rule definitions...
</grammar>
On the other hand, an external XML grammar file must contain a valid XML prolog. An external XML grammar file also contains its own `<grammar>` element. Remember that this `<grammar>` element is only for defining the grammar, not for specifying its usage. Consequently, it can have fewer attributes than its inline cousin:

```xml
...XML prolog...
<grammar ...header attributes only...>
  ...grammar header elements...
  ...grammar rule definitions...
</grammar>
```

Grammar files start out as uncompiled text files. For the GSL format, the BeVocal VoiceXML interpreter supports compiled versions of the files. See “Compiled Grammar Files” on page 24 for information about how to compile a grammar file and how to use the result.

### Rule Definitions

A grammar body consists of one or more rule definitions. A grammar rule definition can have two parts:

- A **rule name** that identifies the rule for use in other rules. In some GSL grammars, the name is optional; in all other cases, the name is required.
- A **rule expansion**, which defines the possible utterances associated with that rule. In GSL the expansion is also called a grammar expression.

### Rule Expansions

In any format, a rule expansion consists of a combination of tokens, rule references, semantic interpretations, and syntax for combining these into more complex expressions.

#### Tokens

A token corresponds directly to words the user can legally say, such as "yes", "ten", or "elephant", or to the DTMF keys the user can press.

#### Rule References

A **rule reference** is simply a named grammar rule that is referenced (by name) from another rule. GSL grammars sometimes refer to rule references as subgrammars or subrules.

Rule references let you modularize your grammar descriptions. Because a rule reference is just a named rule, it can contain another rule reference, thus allowing you to create hierarchies of grammar rules.

When a grammar rule contains a rule reference, the effect is the same as if the referenced rule’s grammar expansion appeared in place of the rule name.

#### Semantic Interpretations

All of the grammar formats support the ability to specify information for the semantic interpretation of a rule. In GSL, this ability is done with assignment commands; in the other formats, it is done with tags. Do not confuse this use of the word "tag" with its more common meaning in the context of VoiceXML. Here, it refers to a particular piece of syntax in the grammar format—in XML, it happens to be the `<tag>` tag, but in GSL, ABNF, and JSGF, it is the use of curly braces.

Semantic interpretations are not needed in every grammar. For example, you probably don’t need them for grammars in `<link>` or `<choice>` elements. If the user speaks a phrase that matches a link grammar, the `<link>` element is simply activated. If a semantic interpretation is present, the interpreter stores it in the `application.lastresults` variable, but does not use it for any other purpose; your application can use this variable to access the semantic interpretation.

Rules for grammars that appear within `<field>` or `<form>` elements frequently specify semantic interpretations. In some simple grammars, however, you may not need to create a semantic interpretation.
Combinations

The various formats use different syntax for how they combine tokens and rule references. The basic combinations are the same, however. The formats have ways of representing:

<table>
<thead>
<tr>
<th>Combination</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alternation</td>
<td>A set of alternatives (&quot;cat&quot; or &quot;dog&quot; or &quot;goat&quot;)</td>
</tr>
<tr>
<td>Sequence</td>
<td>Multiple expressions that must all be said in a particular order (&quot;great dane&quot;)</td>
</tr>
<tr>
<td>Grouping</td>
<td>Multiple expressions to treat as one for a purpose (use French for &quot;Jacques Cousteau&quot;)</td>
</tr>
<tr>
<td>Repetition</td>
<td>Repeat a single expression some number of times (&quot;very&quot; or &quot;very very&quot; or...)</td>
</tr>
<tr>
<td>Optional</td>
<td>Special case of repeat 0 or 1 times (the &quot;kitty&quot; in match &quot;kitty cat&quot; or &quot;cat&quot;)</td>
</tr>
</tbody>
</table>

Compiled Grammar Files

The first time you reference a particular source grammar file in a VoiceXML document, that grammar file is compiled and the compiled file is cached by the interpreter. Subsequent references to the same source grammar file actually use the compiled file. The grammar file is not recompiled unless it is modified.

If you have an extremely large grammar, such as one that recognizes all company names in a major city, compilation may take a significant amount of time. For example, a 2-megabyte grammar file can take about 15 minutes to compile.

If a large grammar file is compiled when the application references it, the delay may be long enough to cause the application to fail with a timeout error. To avoid this problem, you can compile GSL grammar files before running the application and then refer to the compiled grammar files (instead of the source grammar files) from your application.

Grammar files larger than 200 kilobytes must be compiled (either using the Grammar Compilation tool or with the Grammar Compilation SOAP service), because applications are not allowed to reference that large a source grammar file. (This size restriction also applies to hosting customers.)

Typically, you develop and test your application to reference a small subset of your grammar in a source grammar file. When you are ready to test or deploy the application with the full grammar, you compile the full grammar file and modify your application to reference the compiled file.

You can compile GSL grammars in 3 different ways:
- Use the Grammar Compiler tool in the BeVocal Café
- Use Nuance’s grammar compilation tool
- Use the Grammar Compilation Service

Using the Grammar Compilation Service, you can also compile XML grammars. For information on this service, see the Grammar Compilation Service documentation.

Finally, you can compile Nuance SayAnything grammars using Nuance’s grammar compilation tool. For information on SayAnything grammars, see Chapter 8, “Nuance SayAnything Grammars”.

Using the BeVocal Café Grammar Compiler

You can use the Grammar Compiler tool to submit a request for offline compilation of a GSL grammar file. For a complete description of this tool, see Chapter 7, “Grammar Compiler” in Using the BeVocal Café Tools.

The request must specify the URI of the grammar file and an email address where you can be notified when the compilation is completed. You may also specify the root rule of the grammar; if you do not, the
first public rule in the grammar file is used as the root rule. When the grammar file is compiled, it is assigned a unique key and you are sent email informing you of this key. You use this key in a VoiceXML application to reference the compiled grammar.

To reference the grammar in a grammar file compiled with the Grammar Compiler, you set the src attribute of the <grammar> element to a URI of the form:

compiled:grammar/key

where key is the unique key for the compiled file that you received by email after the file was compiled.

If you want to combine a compiled grammar in a more complex way with another grammar, you can refer to a compiled grammar from within an ABNF grammar:

<grammar>
<![CDATA[
    #ABNF 1.0 en-US;
    root $script;
    $script = prescription $<compiled:/key>;
]]>
</grammar>

Using Nuance’s Grammar Compilation Tool

Nuance developers can create a Nuance Grammar Object to use with their BeVocal VoiceXML application. You create a Nuance Grammar Object from a GSL grammar file using Nuance compilation tools.

Nuance developers can get information about its tools by visiting http://extranet.nuance.com. The basic steps for creating and using a Nuance Grammar Object (NGO) with BeVocal VoiceXML are:

1. Compile a static package against the English.America.3 master package and with the -enable_jit option turned on. The command line for the nuance-compile utility would look like:

   nuance-compile Main.grammar English.America.3 -enable_jit

   where Main.grammar contains the contents of the grammar to precompile into the NGO.

2. Compile the NGO against this static package. The command line for nuance-compile-ngo utility would look like:

   nuance-compile-ngo Main.grammar -package main

   where main is the name of the package compiled in step 1 and Main.grammar contains the contents of the grammar that will be precompiled into the NGO.

3. From your VoiceXML document, reference the resultant NGO in the same way as any other external grammar. For example:

   <grammar src="http://www.yourserver.com/grammar/foo.ngo"/>

   Either the extension of the NGO must be .ngo or its Content-Type HTTP header must be application/x-nuance-dynagram-binary.

There are trade-offs to be considered before compiling your own NGOs:

- After compilation of NGOs, you reference them as you do any other resource (such as a grammar, audio file, or JSP). This means that NGOs are subject to the caching rules that apply to any resource and you must take care to appropriately cache them.
- An NGO is very much larger in size than the original uncompiled GSL grammar. Consequently, fetch of the NGO might take a long time.
- The advantage of using NGOs is that the platform does not need to compile your grammars, keeping you in total control of the compilation. Allowing you to maintain control over the compilation could be considered to align more closely with the concept of VoiceXML applications as “Web resources”, because you can now view your compiled grammar as you would any other Web resource.
This chapter describes the syntax for defining a grammar using the XML format. This format uses XML elements to represent the grammar constructs. The structure and possible content of a grammar are almost the same, whether the grammar occurs inline in the `<grammar>` tag of a VoiceXML document or externally as a separate grammar file referenced by a URI from the `<grammar>` tag of a VoiceXML document.

A grammar consists of a header followed by a body. The header is information relevant to defining the grammar as a whole. The body is a set of rule definitions that are used to match against user input. All the rules defined in the grammar are active only when the grammar is active.

The current W3C specification for this syntax is the *Speech Recognition Grammar Specification*, http://www.w3c.org/TR/2002/CR-speech-grammar-20020626. That specification defines the syntax for both ABNF and XML grammars and guarantees that these two formats are semantically equivalent. That is, you can represent exactly the same set of utterances in either grammar format. See Chapter 5, “ABNF Grammar Format” for details of the ABNF syntax.

This chapter contains the following sections:

- XML Tags
- Differences between Inline and External Definitions
- Comments
- Header
- Rule Definitions
- Rule Expansions

**XML Tags**

An XML grammar, like HTML and VoiceXML, uses markup tags and plain text. A *tag* is a keyword enclosed by the angle bracket (`<` and `>`) characters. A tag may have *attributes* inside the angle brackets. Each attribute consists of a *name* and a *value*, separated by an equal (`=`) sign; the value must be enclosed in quotes.

Tags occur in pairs; corresponding to the start tag `<keyword>` is the end tag `</keyword>`. Between the start and end tag, other tags and text may appear. Everything from the start tag to the end tag is called an *element*. If one element contains another, the containing element is called the *parent* element of the contained element. The contained element is called a *child* element of its containing element. The parent element may also be called a *container*.

If an element contains no child elements, you can omit the end tag by replacing the final ">" of the start tag with "="/>. Such elements are called *empty elements*. 
Differences between Inline and External Definitions

An inline grammar is defined completely within the `<grammar>` element in a VoiceXML document. An external grammar, on the other hand, is defined completely in an external file and referenced in the VoiceXML document.

The recognized extensions for an external XML grammar file are `.grxml` and `.xml`. Because the `.xml` extension can be used for any XML file, not just a grammar file, `.grxml` is the preferred extension for XML grammar files.

When you include an XML grammar definition directly in your VoiceXML document, you put all of the attributes on the enclosing VoiceXML `<grammar>` tag. Also, an inline XML grammar inherits the XML prolog of the VoiceXML document; it does not have one of its own. So, an inline XML grammar looks like:

```
<grammar ...usage and header attributes...>
  ...grammar header elements...
  ...grammar rule definitions...
</grammar>
```

On the other hand, an external XML grammar file must contain a valid XML prolog. An external XML grammar file also contains its own `<grammar>` element. Remember that this `<grammar>` element is only for defining the grammar, not for specifying its usage. Consequently, it can have fewer attributes than its inline cousin:

```
...XML prolog...
<grammar ...header attributes only...>
  ...grammar header elements...
  ...grammar rule definitions...
</grammar>
```

The following sections describe exactly what goes in the header and body for both inline and external XML grammars.

Comments

Comments may be placed in most places in a grammar definition. Use standard XML comments of the form:

```
<!-- this is a comment -->
```

Header

The header of an XML grammar is split across attributes of the `<grammar>` element and some special elements that, if present, must be the first children of the `<grammar>` element.

The header consists of the legal XML Prolog (only for an external grammar) and an appropriately constructed Root Element (always). The root `<grammar>` element can have attributes specifying the following information:

- XML Version
- XML namespace
- Schema attributes
• Language
• Grammar Mode
• Root Rule
• Tag Format
• Base URI

The XML version, XML namespace, grammar version, and language attributes are required; all of the other attributes are optional. If the grammar is inline, the root `<grammar>` element may include additional attributes described in “VoiceXML `<grammar>` element” on page 6.

The `<grammar>` element can contain any number of the following elements in any order:
• Pronunciation Lexicon (any number)
• Meta and HTTP-Equiv (any number)
• Metadata (any number)

If the definition contains any of these subelements, they must all occur before any rule definitions.

These attributes and subelements of the `<grammar>` element constitute the header of the document. The rest of the subelements of the `<grammar>` are the actual rule definitions and constitute the body of the document.

The following is an example header for an external grammar file:

```xml
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE grammar PUBLIC "-//W3C//DTD GRAMMAR 1.0//EN"
  "http://www.w3.org/TR/speech-grammar/grammar.dtd">
<grammar version="1.0"
  xml:lang="en"
  mode="voice"
  root="myRule"
  xmlns="http://www.w3.org/2001/06/grammar"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.w3.org/2001/06/grammar
                      http://www.w3.org/TR/speech-grammar/grammar.xsd"
  xml:base="http://www.example.com/base-file-path">
```

**XML Prolog**

The XML prolog contains the XML declaration and an optional DOCTYPE declaration referencing the grammar DTD. The XML prolog may also contain XML comments, processor instructions and other content.

If the grammar is an internal grammar, then it inherits its prolog from the prolog for the VoiceXML document as a whole; that is, an inline XML grammar cannot have a separate XML prolog. An external XML grammar file, on the other hand, must start with its own XML prolog.

The required version number of the XML declaration indicates which version of XML is being used; currently, its value must be 1.0. The encoding attribute indicates the scheme used for encoding character data in the document. For example, for US applications it would be common to use US-ASCII, UTF-8 (8-bit Unicode) or ISO-8859-1. For Japanese grammars, character encodings such as EUC-JP and UTF-16 (16-bit Unicode) could be used. The declaration of the character encoding is optional but strongly recommended.

The following are examples of XML headers with and without the character encoding declaration.

```xml
<?xml version="1.0" encoding="ISO-8859-1"?>
<?xml version="1.0" encoding="EUC-JP"?>
<?xml version="1.0"?>
```
The optional DOCTYPE declaration, if present, must be as follows:

```
<!DOCTYPE grammar PUBLIC "-//W3C//DTD GRAMMAR 1.0//EN"
 "http://www.w3.org/TR/speech-grammar/grammar.dtd">
```

**Note:** To be a legal XML document, the first 4 characters of any XML file (including an external XML grammar file) must be:

```
<?xml
```

No characters, not even whitespace characters such as space or newline, can come before these 4 characters in an external grammar file.

**Root Element**

The root element of the grammar definition is a `<grammar>` element. The attributes of this element and some of its subelements constitute the rest of the grammar’s header. There are four standard attributes whose values, if present, never change:

<table>
<thead>
<tr>
<th>Attributes</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>version</td>
<td>Indicates which version of the grammar specification is being used; required. Currently, the value must be 1.0. For an inline grammar, this attribute must occur on the VoiceXML <code>&lt;grammar&gt;</code> tag. For an external grammar, it must occur on the <code>&lt;grammar&gt;</code> tag in the external file, not on the <code>&lt;grammar&gt;</code> tag in the VoiceXML document.</td>
</tr>
<tr>
<td>xmlns</td>
<td>Indicates the grammar namespace. The value must be <a href="http://www.w3.org/2001/06/grammar">http://www.w3.org/2001/06/grammar</a>. This attribute is required on the <code>&lt;grammar&gt;</code> tag in an external grammar file; it cannot occur on the <code>&lt;grammar&gt;</code> tag in a VoiceXML document.</td>
</tr>
<tr>
<td>xmlns:xsi, xsi:schemaLocation</td>
<td>Indicates the location of the grammar schema. These attributes are optional on the <code>&lt;grammar&gt;</code> tag in an external grammar file; they cannot occur on the <code>&lt;grammar&gt;</code> tag in a VoiceXML document.</td>
</tr>
</tbody>
</table>

So, in an external file, these standard attributes would be as follows:

```
<grammar version="1.0"
            xmlns="http://www.w3.org/2001/06/grammar"
            xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
            xsi:schemaLocation="http://www.w3.org/2001/06/grammar
                                 http://www.w3.org/TR/speech-grammar/grammar.xsd">
  ...
</grammar>
```

The other header attributes are described in the following sections.

**Grammar Version**

In addition to the `version` attribute of the XML prolog, you must also specify the `version` attribute for the grammar. This attribute specifies the version of the grammar format. It does not let you specify a version for a grammar you write. This attribute is required for all XML grammars; currently, its value must be 1.0.

**Language**

The `xml:lang` attribute indicates the primary language contained in the document and optionally indicates a country or other variation. Additionally, any legal rule expansion may be labeled with a language identifier to override the language for that expansion. The format is:

```
<grammar ... xml:lang="lang">
```
Currently, the supported values for `lang` are:

<table>
<thead>
<tr>
<th>Language</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>en</td>
<td>English</td>
</tr>
<tr>
<td>en-US</td>
<td>American English</td>
</tr>
<tr>
<td>es</td>
<td>Spanish</td>
</tr>
<tr>
<td>es-US</td>
<td>American Spanish</td>
</tr>
<tr>
<td>fr-ca</td>
<td>French Canadian</td>
</tr>
</tbody>
</table>

This attribute is ignored for DTMF grammars; it is required for voice grammars.

**Grammar Mode**

The mode of a grammar indicates whether the grammar is for recognizing speech or DTMF input. The default mode is speech. A single grammar cannot recognize both DTMF and user speech. If you want to have both recognized in a single recognition state, you must have multiple grammars active in that state.

The format is:

```xml
<grammar ... mode="theMode"/>
```

where `theMode` is either `voice` or `dtmf`.

In a DTMF grammar, the language header is ignored and any language attachments found in rules are ignored.

**Root Rule**

As described in Chapter 1, “Using VoiceXML Grammars”, a `reference` to a grammar may or may not specify a rule from which to start recognition. If the reference does not specify a rule to start from, the grammar *must* specify a root rule; in this case, recognition will start from the root rule.

For an internal grammar, the grammar definition *must always* specify the rule to use as its root rule, even if the grammar contains only 1 rule. This requirement is because there is no rule reference from which the interpreter can determine which rule to use.

For an external grammar, however, specifying the root rule for the grammar in the definition is optional.

If you specify a root rule in the external grammar definition, then when you reference the grammar, you do not have to include a rule name in the URI that references the grammar. If you do not specify a root rule in the grammar definition, then you must include a rule name in the URI (see “Referencing Grammars” on page 4).

Whether or not the grammar definition specifies a root rule, the URI that references the grammar can specify a different rule from which to start recognition.

For example, you might have a general purpose grammar you use for recognizing all manner of animal. There might be one rule that recognizes all animals; that could be the specified root rule. However, within the grammar, you might also have a large number or rules for recognizing specific groups of animals—birds, fish, land mammals, marine mammals, coral, and so on.

You could use this same grammar in different recognition states in your VoiceXML document, specifying different rules in the grammar reference, to recognize more or less specific groups of animals. So, in one place you’d specify the rule as the one for all mammals, in another the rule for fish, and so on.

The rule specified as the root rule must be defined within the grammar. It can be scoped as either public or private. (For information on rule scoping, see “Scoping of Rule Definitions” on page 35.)
The format for specifying the grammar’s root rule is:

```xml
<grammar root="ruleName" ...>

where ruleName is the name of a rule defined later in the grammar file.
```

**Tag Format**

The tag-format declaration is a URI that indicates the content type of all tags contained within the grammar. It should also indicate a version. Tags typically contain content for a semantic interpretation processor and in such cases the identifier, if present, should indicate the semantic processor to use. See “Tags (Semantic Interpretations)” on page 44 for a discussion of tags.

The format is:

```xml
<grammar tag-format="uri/ver" ...>

Currently, the only accepted value is `semantics/1.0`. The BeVocal VoiceXML interpreter ignores the value; it treats any tags it finds as semantic interpretations and expects them to be in the format described in “Tags (Semantic Interpretations)” on page 44.
```

**Base URI**

Relative URIs are resolved according to a base URI, which may come from a variety of sources. You can use the base URI attribute to specify a grammar’s base URI explicitly. The path information specified by the base URI declaration only affects URIs in the grammar where the element appears. The format is:

```xml
<grammar ... xml:base="uri">

For example:

```xml
<grammar ... xml:base="http://www.myCompany.com/grammars/base-file-path">
```

The VoiceXML interpreter calculates the base URI for resolving relative URIs according to the following precedences (highest priority to lowest):

- The base URI is set by the `xml:base` attribute on the `<grammar>` element.
- The base URI is provided in a meta declaration.
- The base URI is given by metadata discovered during a protocol interaction, such as from an HTTP header.
- The base URI is that of the current document. Not all documents have a base URI. A grammar is not valid if it contains relative URIs and relies on a default base URI.

**Pronunciation Lexicon**

A grammar may optionally reference one or more external pronunciation lexicon documents. A lexicon document is identified by a URI with an optional media type. A lexicon is a child element of the root `<grammar>` element. The format is:

```xml
<lexicon uri="URI" type="mediaType"/>
```

where URI identifies the location of the pronunciation lexicon and mediaType optionally identifies the media type of the lexicon document. All `<lexicon>` elements must occur before any `<rule>` elements.

Currently, the BeVocal VoiceXML interpreter ignores the `<lexicon>` element.
Meta and HTTP-Equiv

Metadata is information about the grammar definition itself, rather than information about the content of the grammar definition. You can specify metadata with the meta attribute, the <metadata> element, or the http-equiv attribute. You can have as many as you want in your grammar.

The only predefined meta property name is seeAlso; it is used to specify a resource that might provide additional metadata information about the containing grammar; however, you can specify any name-value pair you want. Providing a meta property name lets you provide general document information such as author, copyright, description, keywords, and so on.

Providing an http-equiv lets you provide information to use as HTTP response headers. You can provide any http-equiv value you want; however, the only ones that the BeVocal VoiceXML interpreter recognizes are those that have to do with caching and fetching of resources. See Chapter 4, “Fetching and Caching Resources” in the VoiceXML Programmer’s Guide.

The legal formats for the meta and http-equiv attributes are:

```xml
<meta content="propName" content="value"/>
<meta http-equiv="headerName" content="value"/>
```

where propName is the meta property name, headerName is an HTTP header, and value is an appropriate value for the property or header. For example:

```xml
<grammar ...
   <meta name="Creator" content="Jane Doe"/>
   <meta name="seeAlso" content="http://example.com/my-grammar-metadata.xml"/>
   <meta http-equiv="Expires" content="0"/>
   <meta http-equiv="Date" content="Thu, 12 Dec 2000 23:27:21 GMT"/>
...
</grammar>
```

All <meta> elements must occur before any <rule> elements.

Metadata

The <metadata> element is container in which information about the document can be placed using a metadata schema.

Currently, the BeVocal VoiceXML interpreter ignores the <metadata> element.

Rule Definitions

The body of a grammar consists of a set of rule definitions or, simply, rules. The format of rule definitions is the same, regardless of whether they appear inline or in an external file.

Each rule definition associates a rule name with a rule expansion. The definition also defines the scope of the rule—whether it is local to the grammar in which it is defined or whether it may be referenced within other grammars. Finally, the XML format provides a special syntax for you to provide examples of matching user utterances.

Basic Rule Definition

The core purpose of a rule definition is to associate a legal rule expansion with a rule name. You use the <rule> element to define a rule:

```xml
<rule id="ruleName" scope="theScope">
```
The `<rule>` element is the rule expansion. See “Scoping of Rule Definitions” on page 35 for a description of the `scope` attribute.

Because most grammars in VoiceXML identify a set of possible words that a user might say, the top-level rule expansion in a grammar rule is usually a set of alternatives. For example:

```xml
<rule id="fish">
  <one-of>
    <item>Garibaldi</item>
    <item>"Nassau Grouper"</item>
    <item>Trout</item>
  </one-of>
</rule>
```

This rule is named `fish`, and the rule expansion is the set of alternatives. This grammar is matched if the user says “Garibaldi”, “Nassau Grouper”, or “Trout”.

### Rule Names

The rule name is simply a string that identifies the rule. The rule name for each rule definition must be unique within the grammar. The same rule name may be used in multiple grammars.

In a rule reference (described in “Rule References” below), you can use the rule name to specify a particular rule as the rule from which to start recognition.

A rule name is a case-sensitive character string that does not contain any of the characters:

- . :
- ~

In addition, a rule name cannot be the name of a special rule. That is, you cannot name your rule any of the following:

- `GARBAGE`
- `NULL`
- `VOID`

### Special Rules

Rules with the names `GARBAGE`, `NULL`, and `VOID` are predefined. Your grammar must not contain rules with these names. The special rules have the following meanings:

- **GARBAGE**
  
  Defines a rule that matches any speech up until the next rule match, the next token, or until the end of spoken input. For example, you can use the following grammar to match "queen trigger fish", "picasso trigger fish", and "mean old trigger fish", as well as simply "trigger fish":

    ```xml
    <rule id="trigger">
      <ruleref special="GARBAGE"/>
      <token>Trigger Fish</token>
    </rule>
    ```

- **NULL**
  
  Defines a rule that matches if the user doesn’t say anything. For example, you can use the following grammar to match "queen", "picasso", or silence.

    ```xml
    <rule id="trigger">
      <one-of>
        <item><ruleref special="NULL"/></item>
        <item><token>Queen</token></item>
      </one-of>
    </rule>
    ```
VOID

Defines a rule that can never be spoken. Inserting VOID into a sequence automatically makes that sequence unspeakable. For example, nothing a user says can match this rule:

```xml
<rule id="trigger">
  <item><ruleref special="VOID"/></item>
  <one-of>
    <item><token>Queen</token></item>
    <item><token>Picasso</token></item>
  </one-of>
</rule>
```

You can use the NULL and VOID rules together to dynamically change whether or not a rule is active. For example, assume your chamber of commerce application provides information that varies from season to season. During the winter, it answers questions about the amount of snow at the local ski resorts; during the summer, it answers questions about the hours of the local amusement parks; all year round, it answers questions about the plays at local theaters. You don't want the application to recognize questions about ski resorts during the summer.

You could completely change your grammars at every season. This might be difficult to maintain. You could ease the problem by organizing your grammars with a judicious combination of VOID and NULL rules. You would use the VOID and NULL rules to turn on and off recognition of the appropriate questions at the appropriate times of year.

The format you use to refer to a special rule is a particular instance of a rule reference. The `<ruleref>` element can either specify the `special` attribute, as shown here, or the `uri` attribute; it cannot specify both (see “Rule References” below).

### Scoping of Rule Definitions

Each defined rule has a scope of either private or public.

- A rule with `public` scope is visible outside its grammar. A rule with public scope can be activated for recognition; that is, it can define the top-level syntax of spoken input. For example, a rule reference can specify a public rule by name.
- A rule with `private` scope is visible only within its containing grammar and may be referenced only by other rules within the same grammar. The exception to this is that a rule that has been declared the root rule can always be referenced externally, even if it has private scope.

By default, a rule has private scope. The format is:

```xml
<rule id="ruleName" scope="theScope">
  ... ruleExpansion ...
</rule>
```

where `theScope` is either `public` or `private`. If you do not provide the `scope` attribute, the rule is made private.

For example, the following set of definitions creates one public rule named `snapper` and two private rules named `snapperType` and `fishColors`:

```xml
<rule id="snapper" scope="public">
  <ruleref uri="#snapperType"/>
  <token>Snapper</token>
</rule>
```
<rule id="snapperType">
  <token>Mutton</token>
  <ruleref uri="#fishColors"/>
</rule>

<rule id="fishColors" scope="private">
  <one-of>
    <item>Black</item>
    <item>Gray</item>
    <item>Red</item>
  </one-of>
</rule>

You should only make public the rules in your grammar that you want to be visible to other grammars. The use of private and public scope allows you to write more modular and maintainable grammars. For example, you can define a grammar that has a set internal "worker" rules that are combined to provide a smaller number of externally-accessible rules. Hiding worker rules prevents their accidental misuse.

**Note:** Do not confuse the scope of a rule with the scope of its containing grammar. The scope of the grammar indicates where in the VoiceXML application the grammar is active. The scope of the rule indicates whether or not the rule is available to other active grammars.

**Rule Recursion**

You can write a rule that refers to itself either directly or indirectly through a rule that it references.

<!-- Rule that refers to itself directly -->
<rule id="digits">
  <one-of>
    <ruleref uri="digit"/>
    <item>
      <ruleref uri="digit"/>
      <ruleref uri="digits"/>
    </item>
  </one-of>
</rule>

<!-- Rule that indirectly refers to itself -->
<rule id="nounPhrase">
  <ruleref uri="noun"/>
  <ruleref uri="prepositionalPhrase"/>
</rule>

<rule id="prepositionalPhrase">
  <ruleref id="preposition"/>
  <ruleref id="nounPhrase"/>
</rule>

You should be careful in writing rules that refer to themselves. GSL does not support left-recursive rules. That is, it does not support defining a rule whose first sequential subcomponent contains itself:
Rule Definitions

<!-- Legal -->
<rule id="digits">
  <one-of>
    <ruleref uri="digit"/>
    <item>
      <ruleref uri="digit"/>
      <ruleref uri="digits"/>
    </item>
  </one-of>
</rule>

<!-- Also legal -->
<rule id="digits">
  <one-of>
    <item>
      <ruleref uri="digit"/>
      <ruleref uri="digits"/>
    </item>
    <ruleref uri="digit"/>
  </one-of>
</rule>

<!-- Illegal -->
<rule id="digits">
  <one-of>
    <item>
      <ruleref uri="digits"/>
      <ruleref uri="digit"/>
    </item>
    <ruleref uri="digit"/>
  </one-of>
</rule>

<!-- Also illegal -->
<rule id="digits">
  <one-of>
    <item>
      <ruleref uri="digits"/>
      <ruleref uri="digit"/>
    </item>
    <ruleref uri="digit"/>
  </one-of>
</rule>

This restriction ensures that the interpreter doesn’t get lost down an infinite path trying to match a rule.

Example Phrases

It can be a great help to people using your grammar if you include with a rule examples of phrases that match the rule definition. The XML format supports a special syntax for including example phrases. You can provide any number of example phrases in each definition. Because the examples follow a specific
syntax, instead of simply being free-form comments, automated tools for regression testing and grammar documentation can make use of the examples.

You include an example phrase in an `<example>` element. Any `<example>` elements must be before any other content in the `<rule>` element. For example:

```xml
<rule id="snapper" scope="public">
  <!-- matches several common varieties of snapper -->
  <example>red snapper</example>
  <example>mutton snapper</example>
  <ruleref uri="#snapperType"/> <token>Snapper</token>
</rule>
```

Rule Expansions

A *rule expansion* is the part of a rule definition that actually describes what utterances match the rule. A rule expansion is a token, a rule reference, a semantic interpretation, or an arbitrarily complex combination of these things. For example, a rule expansion might express any of these ideas:

- Match the phrase "North Dakota"
- Match the name of any of the 50 United States
- Match a phrase that optionally starts with politeness words (like "please") is followed by an action phrase (like "I want to go to") and then by a destination (like "Fargo, North Dakota") and possibly ends with another politeness phrase (like "thank you") and return just the destination

Language

You can change the language associated with any rule expansion; that is, with any token, rule reference, or combination. This allows you to change the language for a short time. The new language affects only the tokens of the expansion; it does not effect rule references or semantic interpretations.

Currently, the only allowed languages are `en-us` and `en`. To change the language of a rule expansion, assign a value to the `xml:lang` attribute of the appropriate element. For example:

```xml
<!-- Default grammar language is English -->
<grammar ... xml:lang="en-US">
<!-- Single language attachment to tokens -->
<rule id="yes">
  <one-of>
    <item>yes</item>
    <item xml:lang="es-US">si</item>
  </one-of>
</rule>
<!-- Single language attachment to an expansion -->
<rule id="painters">
  <one-of xml:lang="es-US">
    <item>Frida Kahlo</item>
    <item>Diego Rivera</item>
  </one-of>
</rule>
```

This ability to change language will be further illustrated in later sections.
Tokens

A token is the part of a rule expansion that actually mentions a word that a user might actually speak or a DTMF key a user might press. If the mode of the grammar (see “Grammar Mode” on page 31) is voice, then all tokens must be voice tokens; conversely, if the mode of the grammar is dtmf, then all tokens must be DTMF tokens.

Voice Tokens

Voice tokens are words that a user can say. Any unmarked text is a token. A token that contains whitespace or other special characters can be enclosed in double quotes. Alternatively, the token can be contained in a <token> element, in which case it must contain only CDATA and is not enclosed in double quotes. The following table shows examples of tokens.

<table>
<thead>
<tr>
<th>Token Type</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Single unquoted token</td>
<td>Garibaldi</td>
</tr>
<tr>
<td>Single unquoted non-alphabetic token</td>
<td>2</td>
</tr>
<tr>
<td>Single quoted token with white space</td>
<td>&quot;Triton Trigger Fish&quot;</td>
</tr>
<tr>
<td>Single quoted token without white space</td>
<td>&quot;garibaldi&quot;</td>
</tr>
<tr>
<td>Two tokens delimited by white space</td>
<td>trigger fish</td>
</tr>
<tr>
<td>Single XML token in &lt;token&gt;</td>
<td>&lt;token&gt;San Francisco&lt;/token&gt;</td>
</tr>
</tbody>
</table>

You can specify the language for an individual token, if that language is different from the language of the entire grammar. To do so, use this format:

<token xml:lang="lang"/>

where token is the token and is lang a language identifier, as described in “Language” above.

To improve the portability of your grammar, you should follow some simple rules:

- Avoid acronyms. For example, replace "USA" with "u s a" and replace "VXML" with "v x m l".
- Expand abbreviations. For example, replace "st." with "street" or "saint" and replace "dr." with "doctor" or "drive".
- Spell out punctuation. For example, replace "." with "period" or "dot" and replace "<" with "less than" or "open angle bracket".
- Spell out numbers larger than 9. For example, replace "10" with "ten" and "6031" with "sixty thirty one" or "six thousand thirty one".

DTMF Tokens

DTMF tokens are keys that a user can press. The DTMF tokens are as follows:

0 1 2 3 4 5 6 7 8 9 * # A B C D

In a DTMF grammar, any unmarked text must be a legal token. As in a speech grammar, tokens must be separated by white space.
Rule References

Every rule has a name that is unique within the grammar. You can refer to a rule from within another rule either in the same or a different grammar definition. In summary, the formats for doing so are:

<table>
<thead>
<tr>
<th>Reference to a...</th>
<th>Format</th>
</tr>
</thead>
<tbody>
<tr>
<td>Local rule</td>
<td><code>&lt;ruleref uri=&quot;#rulename&quot;/&gt;</code></td>
</tr>
<tr>
<td>Named rule of a grammar identified by a URI</td>
<td><code>&lt;ruleref uri=&quot;grammarURI#rulename&quot;/&gt;</code></td>
</tr>
<tr>
<td>Root rule of a grammar identified by a URI</td>
<td><code>&lt;ruleref uri=&quot;grammarURI&quot;/&gt;</code></td>
</tr>
<tr>
<td>Rule of a grammar identified by a URI and specifying a media type</td>
<td><code>&lt;ruleref uri=&quot;grammarURI&quot; type=&quot;mediaType&quot;/&gt;</code></td>
</tr>
<tr>
<td>Rule of a grammar identified by a URI and specifying an attached language</td>
<td><code>&lt;ruleref uri=&quot;grammarURI&quot; xml:lang=&quot;lang&quot;/&gt;</code></td>
</tr>
<tr>
<td>Special rule definition</td>
<td><code>&lt;ruleref special=&quot;NULL&quot;/&gt;</code>&lt;br&gt;<code>&lt;ruleref special=&quot;VOID&quot;/&gt;</code>&lt;br&gt;<code>&lt;ruleref special=&quot;GARBAGE&quot;/&gt;</code></td>
</tr>
</tbody>
</table>

See “External Grammar Files” on page 4 for a description of how relative URIs work in grammar files. See “Special Rules” on page 34 for information on the special rules.

When a grammar rule contains a rule reference, the effect is the same as if the referenced rule’s rule expansion appeared in place of the rule name. For example, here the `PrimaryColors` grammar rule refers to the `Shades` rule:

```xml
<rule id="PrimaryColors">
  <ruleref uri="Shades" repeat="0-1"/>
  <one-of>
    <item>red</item>
    <item>blue</item>
    <item>green</item>
  </one-of>
</rule>
```

The `PrimaryColors` rule could also be written as:

```xml
<rule id="PrimaryColors">
  <one-of repeat="0-1">
    <item>dark</item>
    <item>light</item>
  </one-of>
  <one-of>
    <item>red</item>
    <item>blue</item>
    <item>green</item>
  </one-of>
</rule>
```
Local Reference
When referencing rules defined in the same grammar as the reference, you should always use a simple rule name reference which consists of the local rule name only.

The `<ruleref>` is always an empty element with a `uri` attribute. Here, the value of the `uri` attribute is simply the rule name prefixed by a # character (sometimes called a URI fragment). For example:

```xml
<ruleref uri="#fish"/>
<ruleref uri="#marineMammals"/>
```

External Reference
You can reference rules defined in a different grammar. Here, the other grammar must be an external grammar which you can reference by its URI.

From one grammar, you cannot reference a rule in another grammar if that other grammar is of a different mode. That is, you cannot reference a rule in a DTMF grammar from a rule in a speech grammar or vice-versa.

You can optionally specify a specific rule in that grammar from which to start recognition. If you do not specify a rule, then the referenced grammar must itself specify a root rule.

You can optionally specify a media type, indicating the content type of the referenced grammar. For details on accepted media types, see “Grammar Formats” on page 7.

You can optionally specify a language, indicating the language of the referenced grammar. For details on accepted languages, see “Language” above. XML, ABNF, and JSGF grammars all require that a grammar declare its language. For these grammar formats, any language you specify with a rule reference is ignored. The GSL format does not provide a way to specify the language of a grammar; if a rule reference to a GSL grammar includes the `xml:lang` attribute, that attribute is used by the interpreter.

An external reference has one of the following formats:

```xml
<ruleref uri="uri"/>
<ruleref uri="uri" type="mediaType"/>
<ruleref uri="uri" xml:lang="lang"/>
```

where `uri` is a standard URI (optionally followed by a `#ruleName` fragment to indicate a rule in the grammar), `mediaType` is one of the media types described in “Grammar Formats” on page 7, and `lang` is one of the languages described in “Language” above. For example:

```xml
// Reference to a specific rule of an external grammar
<ruleref uri="/fish.gram#butterflies"/>

// Reference to the root rule of an external grammar
<ruleref uri="http://www.myCompany.com/grammars/fish.gram"/>

// References with associated media types
<ruleref
    uri="http://www.myCompany.com/grammars/fish#butterflies"
    type="application/srgs"/>

<ruleref uri="/fish" type="application/srgs"/>

// Reference with an associated media type and language
<ruleref
    uri="http://www.myCompany.com/grammars/animals#butterflies"
    type="application/x-nuance-gsl"
    xml:lang="en-US"/>
```
XML grammars combine tokens and rule references into more complex expressions. The basic types of combination are:

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alternation</td>
<td>A set of alternatives (&quot;cat&quot; or &quot;dog&quot; or &quot;goat&quot;). <code>&lt;one-of&gt;</code> element.</td>
</tr>
<tr>
<td>Sequence</td>
<td>Multiple expressions that must all be said in a particular order (&quot;great dane&quot;). <code>&lt;item&gt;</code> element.</td>
</tr>
<tr>
<td>Grouping</td>
<td>Multiple expressions to treat as one for a purpose (use French for &quot;Jacques Cousteau&quot;). <code>&lt;item&gt;</code> element.</td>
</tr>
<tr>
<td>Repetition</td>
<td>Repeat a single expression some number of times (&quot;very&quot; or &quot;very very&quot; or...). <code>&lt;item&gt;</code> repeat attribute of <code>&lt;item&gt;</code> element</td>
</tr>
<tr>
<td>Optional</td>
<td>Special case of repeat 0 or 1 times (the &quot;kitty&quot; in match &quot;kitty cat&quot; or &quot;cat&quot;).repeat attribute of <code>&lt;item&gt;</code> element</td>
</tr>
</tbody>
</table>

**Alternatives and Weights**

A set of alternatives matches if the caller says one of the things in the set. You specify a set of alternatives with the `<one-of>` element. You use the `<item>` element to indicate each alternative in the set. For example:

```xml
<one-of>
  <item>cat</item>
  <item>dog</item>
  <item>fish</item>
</one-of>
```

You can have as many alternatives as you want in the set.

You can optionally specify a different language for the set of alternatives. You specify a language with the same syntax as described in "Language" above. That is, you add the `xml:lang` attribute to the `<one-of>` element or the appropriate `<item>` element. These two `<one-of>` elements express the same thing:

```xml
<one-of xml:lang="es">
  <item>Diego</item>
  <item>Rivera</item>
</one-of>
<one-of>
  <item xml:lang="es">Diego</item>
  <item xml:lang="es">Rivera</item>
</one-of>
```

You can optionally provide a weight for any number of the alternatives. A `weight` indicates how likely a particular alternative is. You specify a weight as a positive floating point number, such as 2, 2.5, 0.8, or .4. A weight of 1 is the same as not specifying a weight at all. A weight larger than 1 indicates that the alternative is more likely; a weight less than 1 indicates that the alternative is less likely.

You specify a weight by specifying the `weight` attribute on the `<item>` element. For example:

```xml
<one-of>
  <item weight="3.1415">cat</item>
  <item>dog</item>
  <item weight=".25">fish</item>
</one-of>
```
<one-of>
  <item weight="10">fish</item>
  <item weight="2">angel fish</item>
  <item weight=".1">anthia</item>
</one-of>

The first of these says that it is quite likely the user will say "cat", less likely but still fairly likely the user will say "dog", and not very likely the user will say "fish". The second says the user will almost certainly say "fish", might say "angelfish", and probably will not say "anthia".

**Sequence**

A sequence is a set of expansions that must all be said in the order specified. Sequence is not a separate piece of syntax; rather, think of sequence as the natural order of things. That is, the interpreter always uses all of the expressions in the order presented, unless the expression specifies alternation. Some examples:

<!-- sequence of tokens -->
what is coral

<!-- sequence of rule references -->
<ruleref uri="#question"/> <ruleref uri="#subject"/>

<!-- sequence of tokens and rule references -->
<ruleref uri="#subject"/> is <ruleref uri="#type"/>

**Grouping**

Grouping is merely a way to treat a set of things as a single term. You use grouping, for example, to attach a language identifier or repeat operator (next) to the whole group. The <item> element indicates a group. For example:

<item>this is a group</item>

**Repeats**

You can specify that an expression should be repeated some particular number of times. The syntax for this allows you to specify a variety of repetition types:

<table>
<thead>
<tr>
<th>Syntax</th>
<th>Behavior</th>
</tr>
</thead>
<tbody>
<tr>
<td>repeat=&quot;n&quot;</td>
<td>Repeat the expression exactly n times, where n is 0 or a positive integer. The example says to repeat the expression exactly 6 times.</td>
</tr>
<tr>
<td>repeat=&quot;6&quot;</td>
<td>Repeat the expression exactly 6 times.</td>
</tr>
<tr>
<td>repeat=&quot;n-m&quot;</td>
<td>Repeat the expression between n and m times, where n and m are both either 0 or positive integers and n is less than or equal to m. The example says to repeat the expression 4, 5, or 6 times.</td>
</tr>
<tr>
<td>repeat=&quot;4-6&quot;</td>
<td>Repeat the expression 4, 5, or 6 times.</td>
</tr>
<tr>
<td>repeat=&quot;n-&quot;</td>
<td>Repeat the expression n times or more, where n is 0 or a positive integer. The example says to repeat the expression 3 or more times.</td>
</tr>
<tr>
<td>repeat=&quot;3-&quot;</td>
<td>Repeat the expression 3 or more times.</td>
</tr>
<tr>
<td>repeat=&quot;0-1&quot;</td>
<td>The expression is optional.</td>
</tr>
</tbody>
</table>

Repetition is indicated by use of the repeat attribute on the appropriate element. For example:

<rule id="gear">
  <one-of>
    <item>mask</item> <item>fins</item> <item>snorkel</item>
    <item>booties</item> <item>gloves</item> <item>regulator</item>
  </one-of>
</rule>
<rule id="action">
  <one-of>
    <item>buy</item>
    <item>rent</item>
  </one-of>
</rule>

<rule id="makeRequest">
  <item>I want to</item>
  <ruleref uri="action">
    <ruleref uri="gear" repeat="0-5">
      <item repeat="0-1">
        <token>and</token>
        <ruleref uri="gear">
          <ruleref uri="gear" repeat="0-1">
            <token>and</token>
            <ruleref uri="gear">
          </ruleref>
        </ruleref>
      </item>
    </ruleref>
  </ruleref>
</item>
</rule>

This set of rules matches a variety of utterances, including:

I want to buy gloves.
I want to rent mask and fins.
I want to rent mask, fins, snorkel, and regulator.

Although you’re allowed to create a rule that recognizes an unbounded number of expressions, users do not actually speak forever. The speech recognition will proceed more effectively if you are more careful and only indicate a limited range of occurrences.

You can attach a probability to a repeat operator. The value indicates the probability of successive repetition of the repeated expression. A repeat probability must be in the range between 0.0 and 1.0; note that this is different from a weight attached to an entire expression.

A simple example is an optional expansion (zero or one occurrences) with a probability, for example, of 0.6. The grammar indicates that the chance that the expansion will be matched is 60% and that the chance that the expansion will not be present is 40%.

You can only use a repeat probability when specifying a range of repetitions. The syntax is:

repeat="n-m" repeat-prob="prob"
repeat="n-" repeat-prob="prob"

That is, you use the repeat-prob attribute to specify the repeat probability; you can only use this attribute if you also use the repeat attribute. Here, n and m are integers and prob is the probability.

For example:

<-- The word "angel" is optional and is not very likely to occur. -->
=item repeat="0-1" repeat-prob="0.25">angel</item>

<-- The rule reference to digit must occur between 2 and 4 times -->
<-- and it is very likely it will occur 3 or 4 times. -->
=item repeat="2-4" repeat-prob="0.8">
  <ruleref uri="#digit"/>
</item>

Tags (Semantic Interpretations)

In general, a tag is an arbitrary string that may be included inline within any rule expansion. You can include as many tags as you want within a single expansion. Tags do not affect what constitutes a legal utterance for a rule nor do they affect how the recognition proceeds.

You use tags to return information—a semantic interpretation—about a recognition to the element that invoked the grammar. Upon successful recognition, the BeVocal VoiceXML interpreter will create a JavaScript object whose properties and values are determined by the tags occurring in the matched rule.

If you want the BeVocal VoiceXML interpreter to make use of your tags for this purpose, they must be of a specific format:
Rule Expansions

<tag>
  property="value"
</tag>

where property and value are arbitrary names and values. For example, the following grammar sets two different tags:

<grammar ...>
  <rule id="coloredObject">
    <ruleref id="color"/>
    <ruleref id="object"/>
  </rule>

  <rule id="color">
    <one-of>
      <item> red <tag> color="red" </tag> </item>
      <item> pink <tag> color="red" </tag> </item>
      <item> yellow <tag> color="yellow" </tag> </item>
      <item> canary <tag> color="yellow" </tag> </item>
      <item> green <tag> color="green" </tag> </item>
      <item> khaki <tag> color="green" </tag> </item>
    </one-of>
  </rule>

  <rule id="object">
    <one-of>
      <item>
        <tag> object="vehicle" </tag>
        <one-of><item>truck</item><item>car</item></one-of>
      </item>
      <item>
        <tag> object="toy" </tag>
        <one-of><item>ball</item><item>block</item></one-of>
      </item>
      <item>
        <tag> object="clothing" </tag>
        <one-of><item>shirt</item><item>blouse</item></one-of>
      </item>
    </one-of>
  </rule>
</grammar>

This grammar recognizes phrases such as "yellow shirt" or "canary blouse". For both of those phrases, it will return the same semantic interpretation:

```
{ 
  color: yellow;
  object: clothing;
}
```

This simple example allows your grammar to accept synonyms and return a more canonical result that can be used later in your VoiceXML application.

See “Setting Input Variables” on page 9 for information on how the interpreter will use the semantic interpretation.
This chapter describes the syntax for defining a grammar using the Augmented BNF (ABNF) format. This is a plain-text (non-XML) representation which is similar to a traditional BNF grammar. It is also similar to the JSGF format.

A grammar consists of a header followed by a body. The header is information relevant to defining the grammar as a whole. The body is a set of rule definitions that are used to match against user input. All the rules defined in the grammar are active only when the grammar is active.

The current W3C specification for this syntax is the Speech Recognition Grammar Specification, http://www.w3c.org/TR/2002/CR-speech-grammar-20020626. That specification defines the syntax for both ABNF and XML grammars and guarantees that these two formats are semantically equivalent. That is, you can represent exactly the same set of utterances in either grammar format. See Chapter 4, “XML Speech Grammar Format” for details of the XML syntax.

This chapter contains the following sections:

- Differences between Inline and External Definitions
- Comments
- Keywords
- Header Declarations
- Rule Definitions
- Rule Expansions

Differences between Inline and External Definitions

An inline grammar is defined completely within the `<grammar>` element in a VoiceXML document. An external grammar, on the other hand, is defined completely in an external file and referenced in the VoiceXML document.

The recognized extension for an external ABNF grammar file is `.gram`.

When you include a grammar definition directly in your VoiceXML document, you must take special care. ABNF grammars use special characters not normally permitted in XML documents, such as angle brackets (`<` and `>}). Consequently, you should wrap your inline grammar as a section of CDATA:

```xml
<grammar ...usage attributes...>
  <![CDATA[
    ...grammar header declarations...
    ...grammar rule definitions...
  ]]>  
</grammar>
```
On the other hand, if you have an external ABNF grammar file, the contents of that file should *not* be inside a CDATA section. Also, an external file should not contain a `<grammar>` element. So, an external grammar file simply looks like:

```abnf
...grammar header declarations...
...grammar rule definitions...
```

The following sections describe exactly what goes in the header and body.

**Comments**

Comments may be placed in most places in a grammar definition. You can use comments of these forms:

- // C++/Java-style single-line comment
- /* C/C++/Java-style comment */
- /** Java-style documentation comment */

**Keywords**

ABNF uses some keywords which will be described in later sections. In summary, they are:

<table>
<thead>
<tr>
<th>Context</th>
<th>Keywords</th>
</tr>
</thead>
<tbody>
<tr>
<td>Language declaration</td>
<td>language</td>
</tr>
<tr>
<td>Mode declaration</td>
<td>mode</td>
</tr>
<tr>
<td>Root declaration</td>
<td>root</td>
</tr>
<tr>
<td>Tag format declaration</td>
<td>tag-format</td>
</tr>
<tr>
<td>Base URI declaration</td>
<td>base</td>
</tr>
<tr>
<td>Pronunciation lexicon</td>
<td>lexicon</td>
</tr>
<tr>
<td>Meta or HTTP-equiv declaration</td>
<td>meta, http-equiv, is</td>
</tr>
<tr>
<td>Rule definition</td>
<td>public, private</td>
</tr>
</tbody>
</table>

These keywords are case sensitive. They are not reserved, which means that they may be used as rule names and as tokens. For example, the following is a legal grammar that accepts as input a sequence of one or more "public" tokens:

```abnf
#ABNF 1.0 ISO-8859-1;

language en-AU;
root $public;
mode voice;

public $public = public $public | public;
```

To improve the readability of your grammar and to avoid confusion, do not use reserved keywords as rule names or as tokens.
Header Declarations

The header of an ABNF grammar is contained completely in a set of declarations. The first line of an ABNF file must be the ABNF Self-Identifying Header. After that, it can have any of these declarations in any order:

- Language
- Grammar Mode
- Root Rule
- Tag Format
- Base URI
- Pronunciation Lexicon (any number)
- Meta and HTTP-Equiv (any number)

Other than the ABNF self-identifying header, all declarations are optional.

ABNF comments may appear between the declarations in the ABNF header after the ABNF self-identifying header. The header declarations are followed by the rule definitions of the grammar.

The following is an example of a header:

```
#ABNF 1.0 ISO-8859-1;
language en;
mode voice;
root $myRule;
tag-format FORMAT-STRING;
base <http://www.myCompany.com/grammars/base-file>;
meta "Author" is "Jacques Cousteau";
http-equiv "Date" is "Mon, 11 November 2002 17:27:21 GMT";
```

Note that all of the header information for an ABNF grammar is contained in these declarations. If the `<grammar>` tag of the VoiceXML document that uses or defines the grammar also contains attributes corresponding to these declarations, the interpreter ignores the attributes and this information comes from the grammar definition itself.

ABNF Self-Identifying Header

The ABNF self-identifying header must be present in any legal ABNF grammar. The self-identifying header looks like one of the following:

```
#ABNF ver;
#ABNF ver char-encoding;
```

where `ver` is the grammar version and `char-encoding` is an optional character encoding.

The grammar version indicates the version of the grammar specification the grammar supports; it is not for versioning of the grammar content. You can use a meta declaration for content versioning. Currently, the only legal value for `ver` is `1.0`.

The character encoding declaration indicates the scheme used for encoding character data in the document. For example, for US applications it would be common to use US-ASCII, UTF-8 (8-bit Unicode) or ISO-8859-1. For Japanese grammars, character encodings such as EUC-JP and UTF-16 (16-bit Unicode) could be used. The declaration of the character encoding is optional but strongly recommended.
Language

The language declaration indicates the primary language contained in the document and optionally indicates a country or other variation. Additionally, any legal rule expansion may be labeled with a language identifier to override the language for that expansion. The format is:

```
language lang;
```

Currently, the only supported values for `lang` are `en` and `en-US`. This declaration is ignored for DTMF grammars; it is required for voice grammars.

Grammar Mode

The mode of a grammar indicates whether the grammar is for recognizing speech or DTMF input. The default mode is speech. A single grammar cannot recognize both DTMF and user speech. If you want to have both recognized in a single recognition state, you must have multiple grammars active in that state.

The format is:

```
mode theMode;
```

where `theMode` is either `voice` or `dtmf`.

In a DTMF grammar, the language header is ignored and any language attachments found in rules are ignored.

Root Rule

As described in Chapter 1, “Using VoiceXML Grammars”, a reference to a grammar may or may not specify a rule from which to start recognition. If the reference does not specify a rule to start from, the grammar must specify a root rule; in this case, recognition will start from the root rule.

For an internal grammar, the grammar definition must specify the rule to use as its root rule (because there is no other way for the interpreter to know which rule to use. For an external grammar, however, specifying the root rule for the grammar in the definition is optional.

If you specify a root rule in the external grammar definition, then when you reference the grammar, you do not have to include the rule name in the URI that references the grammar. If you do not specify a root rule in the grammar definition, then you must include a rule name in the URI (see “Referencing Grammars” on page 4).

Whether or not the grammar definition specifies a root rule, the URI that references the grammar can specify a different rule from which to start recognition.

For example, you might have a general purpose grammar you use for recognizing all manner of animal. There might be one rule that recognizes all animals; that could be the specified root rule. However, within the grammar, you might also have a large number or rules for recognizing specific groups of animals—birds, fish, land mammals, marine mammals, coral, and so on.

You could use this same grammar in different recognition states in your VoiceXML document, specifying different rules in the grammar reference, to recognize more or less specific groups of animals. So, in one place you’d specify the rule as the one for all mammals, in another the rule for fish, and so on.

The rule specified as the root rule must be defined within the grammar. It can be scoped as either public or private. (For information on rule scoping, see “Scoping of Rule Definitions” on page 53.)

The format for specifying the grammar’s root rule is:

```
root $rulename;
```

where `rulename` is the name of a rule defined later in the grammar file.
Tag Format
The tag-format declaration is a URI that indicates the content type of all tags contained within the grammar. It should also indicate a version. Tags typically contain content for a semantic interpretation processor and in such cases the identifier, if present, should indicate the semantic processor to use. See “Tags (Semantic Interpretations)” on page 61 for a discussion of tags.

The format is:
\[
\text{tag-format } \langle \text{uri/ver} \rangle;
\]
Currently, the only accepted value is \text{semantics/1.0}. The BeVocal VoiceXML interpreter ignores the value; it treats any tags it finds as semantic interpretations and expects them to be in the format described in “Tags (Semantic Interpretations)” on page 61.

Base URI
Relative URIs are resolved according to a base URI, which may come from a variety of sources. You can use the base URI declaration to specify a grammar’s base URI explicitly. The path information specified by the base URI declaration only affects URIs in the grammar where the element appears. The format is:
\[
\text{base } \langle \text{uri} \rangle;
\]
For example:
\[
\text{base } \langle \text{http://www.myCompany.com/grammars/base-file-path} \rangle;
\]
The VoiceXML interpreter calculates the base URI for resolving relative URIs according to the following precedences (highest priority to lowest):
- The base URI is set by the \text{base} declaration.
- The base URI is provided in a meta declaration.
- The base URI is given by metadata discovered during a protocol interaction, such as from an HTTP header.
- The base URI is that of the current document. Not all documents have a base URI. A grammar is not valid if it contains relative URIs and relies on a default base URI.

Pronunciation Lexicon
A grammar may optionally reference one or more external pronunciation lexicon documents. A lexicon document is identified by a URI with an optional media type. The format is:
\[
\text{lexicon } \langle \text{URI} \rangle;
\]
where \text{URI} identifies the location of the pronunciation lexicon and \text{mediaType} optionally identifies the media type of the lexicon document.

Currently, the BeVocal VoiceXML interpreter ignores the \text{lexicon} declaration.

Meta and HTTP-Equiv
Metadata is information about the grammar definition itself, rather than information about the content of the grammar definition. You can specify metadata with the \text{meta} declaration or the \text{http-equiv} declaration. You can have as many as you want in your grammar.

The only predefined meta property name is \text{seeAlso}; it is used to specify a resource that might provide additional metadata information about the containing grammar; however, you can specify any name-value pair you want. Providing a meta property name lets you provide general document information such as author, copyright, description, keywords, and so on.

Providing an http-equiv lets you provide information to use as HTTP response headers. You can provide any http-equiv value you want; however, the only ones that the BeVocal VoiceXML interpreter recognizes...
are those that have to do with caching and fetching of resources. See Chapter 4, “Fetching and Caching Resources" in the VoiceXML Programmer’s Guide.

The legal formats for the meta and http-equiv declarations are:

```plaintext
meta "propName" is "value";
http-equiv "headerName" is "value";
```

where `propName` is the meta property name, `headerName` is an HTTP header, and `value` is an appropriate value for the property or header. You can enclose the name and value either in matching double quotes or matching single quotes. For example:

```plaintext
meta "Creator" is "Jane Doe";
meta 'seeAlso' is 'http://example.com/my-grammar-metadata.xml';

http-equiv "Expires" is '0';
http-equiv 'Date' is "Thu, 12 Dec 2000 23:27:21 GMT";
```

Rule Definitions

The body of a grammar consists of a set of rule definitions or, simply, rules. The format of rule definitions is the same, regardless of whether they appear inline or in an external file.

Each rule definition associates a rule name with a rule expansion. The definition also defines the scope of the rule—whether it is local to the grammar in which it is defined or whether it may be referenced within other grammars. Finally, the ABNF format provides a special syntax for you to provide examples of matching user utterances.

Basic Rule Definition

The core purpose of a rule definition is to associate a legal rule expansion with a rule name. The basic format of a definition is:

```plaintext
$ruleName = ruleExpansion;
public $ruleName = ruleExpansion;
private $ruleName = ruleExpansion;
```

where `ruleName` is the name of the rule; `ruleExpansion` is its expansion, and public and private specify the scope of the rule. (See “Scoping of Rule Definitions” on page 53 for a discussion of rule scoping.)

Because most grammars in VoiceXML identify a set of possible words that a user might say, the top-level rule expansion in a grammar rule is usually a set of alternatives. For example:

```plaintext
$fish = Garibaldi | "Nassau Grouper" | Trout;
```

This rule is named `fish`, and the rule expansion is the set of alternatives. This grammar is matched if the user says “Garibaldi”, “Nassau Grouper”, or “Trout”.

Rule Names

The rule name is simply a string that identifies the rule. The rule name for each rule definition must be unique within the grammar. The same rule name may be used in multiple grammars.

In a rule reference (described in “Rule References” below), you can use the rule name to specify a particular rule from which to start recognition.

A rule name is a case-sensitive character string that does not contain any of the characters: `.` `-`
In addition, a rule name cannot be the name of a special rule. That is, you cannot name your rule any of the following:

GARBAGE
NULL
VOID

You must always precede a rule name with $ (for example, $myRule), both when defining the rule and when referencing it.

**Special Rules**

Rules with the names GARBAGE, NULL, and VOID are predefined. Your grammar must not contain rules with these names. The special rules have the following meanings:

- **GARBAGE**
  
  Defines a rule that matches any speech up until the next rule match, the next token, or until the end of spoken input. For example, you can use the following grammar to match "queen trigger fish", "picasso trigger fish", and "mean old trigger fish", as well as simply "trigger fish":

  \[
  \text{
  \$trigger = \$GARBAGE "Trigger Fish";}
  \]

- **NULL**

  Defines a rule that matches if the user doesn't say anything. For example, you can use the following grammar to match "queen", "picasso", or silence.

  \[
  \text{
  \$trigger = \$NULL | "Queen" | "Picasso";}
  \]

- **VOID**

  Defines a rule that can never be spoken. Inserting VOID into a sequence automatically makes that sequence unspeakable. For example, nothing a user says can match this rule:

  \[
  \text{
  \$trigger = \$VOID ("Queen" | "Picasso");}
  \]

You can use the NULL and VOID rules together to dynamically change whether or not a rule is active. For example, assume your chamber of commerce application provides information that varies from season to season. During the winter, it answers questions about the amount of snow at the local ski resorts; during the summer, it answers questions about the hours of the local amusement parks; all year round, it answers questions about the plays at local theaters. You don’t want the application to recognize questions about ski resorts during the summer.

You could completely change your grammars at every season. This might be difficult to maintain. You could ease the problem by organizing your grammars with a judicious combination of VOID and NULL rules. You would use the VOID and NULL rules to turn on and off recognition of the appropriate questions at the appropriate times of year.

The format you use to refer to a special rule is a particular instance of a rule reference (see “Rule References” below).

**Scoping of Rule Definitions**

Each defined rule has a scope of either private or public.

- A rule with **public** scope is visible outside its grammar. A rule with public scope can be activated for recognition; that is, it can define the top-level syntax of spoken input. For example, a rule reference can specify a public rule by name.

- A rule with **private** scope is visible only within its containing grammar and may be referenced only by other rules within the same grammar. The exception to this is that a rule that has been declared the root rule can always be referenced externally, even if it has private scope.
By default, a rule has private scope. The format is:

\[ \text{theScope } \text{$ruleName$} = \text{ruleExpansion}; \]

where \text{theScope} is either \text{public} or \text{private}. If you do not provide a keyword, the rule is made private.

For example, the following set of definitions creates one public rule named \text{snapper} and two private rules named \text{snapperType} and \text{fishColors}:

\[
\text{public } \text{$snapper$} = \text{$snapperType$} \text{ Snapper;}
\text{$snapperType$} = \text{Mutton } | \text{$fishColors$;}
\text{private } \text{$fishColors$} = \text{Black } | \text{Gray } | \text{Red;}
\]

You should only make public the rules in your grammar that you want to be visible to other grammars. The use of private and public scope allows you to write more modular and maintainable grammars. For example, you can define a grammar that has a set internal "worker" rules that are combined to provide a smaller number of externally-accessible rules. Hiding worker rules prevents their accidental misuse.

\textbf{Note:} Do not confuse the scope of a rule with the scope of its containing grammar. The scope of the grammar indicates where in the VoiceXML application the grammar is active. The scope of the rule indicates whether or not the rule is available to other active grammars.

\textbf{Rule Recursion}

You can write a rule that refers to itself either directly or indirectly through a rule that it references.

// Rule that refers to itself directly
\[
\text{$Digits$} = \text{$Digit$ } | (\text{$Digit$ } \text{$Digits$});
\text{$Digit$} = 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9;
\]

// Rule that indirectly refers to itself
\[
\text{$NounPhrase$} = \text{$Noun$ } [\text{$PrepositionalPhrase$}];
\text{$PrepositionalPhrase$} = \text{$Preposition$ } \text{$NounPhrase$};
\]

You should be careful in writing rules that refer to themselves. ABNF does not support left-recursive rules. That is, it does not support defining a rule whose first sequential subcomponent contains itself:

// Legal
\[
\text{$Digits$} = \text{Digit } | (\text{Digit } \text{$Digits$});
\text{$Digits$} = (\text{Digit } \text{$Digits$}) \text{ } | \text{Digit};
\]

// Illegal
\[
\text{$Digits$} = (\text{$Digits$ } \text{Digit}) \text{ } | \text{Digit};
\text{$Digits$} = \text{Digit } | (\text{$Digits$ } \text{Digit});
\]

This restriction ensures that the interpreter doesn’t get lost down an infinite path trying to match a rule.

\textbf{Example Phrases}

It can be a great help to people using your grammar if you include with a rule examples of phrases that match the rule definition. The ABNF format supports a special syntax for including example phrases. You can provide any number of example phrases in each definition. Because the examples follow a specific syntax, instead of simply being free-form comments, automated tools for regression testing and grammar documentation can make use of the examples.

You include an example phrase in a comment, by using the @example syntax. The comment which includes the example phrases must immediately precede the relevant rule definition. For example:

/\**
* matches several common varieties of snapper
*
* @example red snapper
Rule Expansions

A *rule expansion* is the part of a rule definition that actually describes what utterances match the rule. A rule expansion is a token, a rule reference, a semantic interpretation, or an arbitrarily complex combination of these things. For example, a rule expansion might express any of these ideas:

- Match the phrase "North Dakota"
- Match the name of any of the 50 United States
- Match a phrase that optionally starts with politeness words (like "please") is followed by an action phrase (like "I want to go to") and then by a destination (like "Fargo, North Dakota") and possibly ends with another politeness phrase (like "thank you") and return just the destination

**Language**

You can change the language associated with any rule expansion; that is, with any token, rule reference, or combination. This allows you to change the language for a short time. The new language affects only the tokens of the expansion; it does not effect rule references or semantic interpretations.

Currently, the only allowed languages are `en-us` and `en`. To change the language of a rule expansion, put an exclamation point (`!`) followed by the language identifier immediately after the expansion, without intervening whitespace. For example:

```plaintext
#ABNF 1.0 ISO-8859-1;
// Default grammar language is US English
language en-US;

// Single language attachment to tokens
$yes = yes | oui!fr-CA;

// Single language attachment to an expansion
$submariners = (Jacques Cousteau | Jacques Mayol)!fr-CA;
```

This ability to change language will be further illustrated in later sections.

**Tokens**

A *token* is the part of a rule expansion that actually mentions a word that a user might actually speak or a DTMF key a user might press. If the mode of the grammar (see “Grammar Mode” on page 50) is `voice`,
then all tokens must be voice tokens; conversely, if the mode of the grammar is dtmf, then all tokens must be DTMF tokens.

**Voice Tokens**

Voice tokens are words that a user can say. Any unmarked text is a token. A token that contains whitespace or other special characters can be enclosed in double quotes. The following table shows examples of tokens.

<table>
<thead>
<tr>
<th>Token Type</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Single unquoted token</td>
<td>Garibaldi</td>
</tr>
<tr>
<td>Single unquoted non-alphabetic token</td>
<td>2</td>
</tr>
<tr>
<td>Single quoted token with white space</td>
<td>&quot;Triton Trigger Fish&quot;</td>
</tr>
<tr>
<td>Single quoted token without white space</td>
<td>&quot;garibaldi&quot;</td>
</tr>
<tr>
<td>Two tokens delimited by white space</td>
<td>trigger fish</td>
</tr>
</tbody>
</table>

You can specify the language for an individual token, if that language is different from the language of the entire grammar. To do so, use this format:

`token!lang`

where `token` is the token and `lang` a language identifier, as described in “Language” above.

You cannot use the following special characters in tokens:

`= ; $ ! | / < > ( ) [ ] { }`

To improve the portability of your grammar, you should follow some simple rules:

- Avoid acronyms. For example, replace “USA” with “u s a” and replace “VXML” with “v x m l”.
- Expand abbreviations. For example, replace “st.” with “street” or “saint” and replace “dr.” with “doctor” or “drive”.
- Spell out punctuation. For example, replace “.” with “period” or “dot” and replace “<” with “less than” or “open angle bracket”.
- Spell out numbers larger than 9. For example, replace “10” with “ten” and “6031” with “sixty thirty one” or “six thousand thirty one”.

**DTMF Tokens**

DTMF tokens are keys that a user can press. The DTMF tokens are as follows:

0 1 2 3 4 5 6 7 8 9 * # A B C D

Because * is a reserved symbol in the ABNF grammar format, you must either enclose a * in double quotes ("**") or instead use the token `star`. Similarly, you should either enclose a # in double quotes ("#") or instead use the token `pound`.

In a DTMF grammar, any unmarked text must be a legal token. As in a speech grammar, tokens must be separated by white space.
Rule References

Every rule has a name that is unique within the grammar. You can refer to a rule from within another rule either in the same or a different grammar definition. In summary, the formats for doing so are:

<table>
<thead>
<tr>
<th>Reference to a...</th>
<th>Format</th>
</tr>
</thead>
<tbody>
<tr>
<td>Local rule</td>
<td>$rulename</td>
</tr>
<tr>
<td>Named rule of a grammar identified by a URI</td>
<td>$&lt;grammarURI#rulename&gt;</td>
</tr>
<tr>
<td>Root rule of a grammar identified by a URI</td>
<td>$&lt;grammarURI&gt;</td>
</tr>
<tr>
<td>Rule of a grammar identified by a URI and specifying a media type</td>
<td>$&lt;grammarURI#rulename&gt;--&lt;mediaType&gt;</td>
</tr>
<tr>
<td>Rule of a grammar identified by a URI and specifying an attached language</td>
<td>$&lt;grammarURI&gt;!lang</td>
</tr>
<tr>
<td>Special rule definition</td>
<td>$NULL $VOID $GARBAGE</td>
</tr>
</tbody>
</table>

See “External Grammar Files” on page 4 for a description of how relative URIs work in grammar files. See “Special Rules” on page 53 for a description of the special rules.

When a grammar rule contains a rule reference, the effect is the same as if the referenced rule’s rule expansion appeared in place of the rule name. For example, here the $primaryColors grammar rule refers to the $shades rule:

$primaryColors = [$shades] (red | blue | green)
$shades = dark | light

The $primaryColors rule could also be written as:

$primaryColors [dark | light] (red | blue | green)

Local Reference

When referencing rules defined in the same grammar as the reference, you should always use a simple rule name reference which consists of the local rule name only. Here, the rule reference is simply rule name is prefixed by a $ character. For example:

$fish
$marineMammals

External Reference

You can reference rules defined in a different grammar. Here, the other grammar must be an external grammar which you can reference by its URI.

From one grammar, you cannot reference a rule in another grammar if that other grammar is of a different mode. That is, you cannot reference a rule in a DTMF grammar from a rule in a speech grammar or vice-versa.

You can optionally specify a specific rule in that grammar from which to start recognition. If you do not specify a rule to use, then the reference grammar must itself specify a root rule.

You can optionally specify a media type, indicating the content type of the referenced grammar. For details on accepted media types, see “Grammar Formats” on page 7.

You can optionally specify a language, indicating the language of the referenced grammar. For details on accepted languages, see “Language” above. XML, ABNF, and JSGF grammars all require that a grammar declare its language. For these grammar formats, any language you specify with a rule reference is
ignored. The GSL format does not provide a way to specify the language of a grammar; if a rule reference to a GSL grammar includes a language attachment, that attachment is used by the interpreter.

An external reference has one of the following formats:

$<\text{uri}>
$<\text{uri}>\text{-<mediatype>}
$<\text{uri}>!\text{lang}

where \text{uri} is a standard URI (optionally followed by a \#\text{ruleName} fragment to indicate a rule in the grammar), \text{mediatype} is one of the media types described in “Grammar Formats” on page 7, and \text{lang} is one of the languages described in “Language” above. Note that there is no whitespace between the $ and the < and also none between the characters >--< or between the > and the !. For example:

// Reference to a specific rule of an external grammar
$<../fish.gram#butterflies>

// Reference to the root rule of an external grammar
$<http://www.myCompany.com/grammars/fish.gram>

// References with associated media types
$<http://www.myCompany.com/grammars/fish#butterflies>--<application/srgs>
$<../fish>--<application/srgs>

// Reference with an associated media type and language
$<../fish>--<application/srgs>!en-AU

Combinations

ABNF grammars combine tokens and rule references into more complex expressions. The basic types of combination are:

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alternation</td>
<td>A set of alternatives (&quot;cat&quot; or &quot;dog&quot; or &quot;goat&quot;). Separated with the vertical bar (</td>
</tr>
<tr>
<td>Sequence</td>
<td>Multiple expressions that must all be said in a particular order (&quot;great dane&quot;). Implied by order of occurrence.</td>
</tr>
<tr>
<td>Grouping</td>
<td>Multiple expressions to treat as one for a purpose (use French for &quot;Jacques Cousteau&quot;). Indicated by parentheses ( ( and ) ).</td>
</tr>
<tr>
<td>Repetition</td>
<td>Repeat a single expression some number of times (&quot;very&quot; or &quot;very very&quot; or...). Use of angle brackets ( &lt; and &gt; ).</td>
</tr>
<tr>
<td>Optional</td>
<td>Special case of repeat 0 or 1 times (the &quot;kitty&quot; in match &quot;kitty cat&quot; or &quot;cat&quot;) Enclosed in square brackets ( [ and ] ).</td>
</tr>
</tbody>
</table>

Alternatives and Weights

A set of alternatives matches if the caller says one of the things in the set. You specify a set of alternatives with vertical bar ( | ). Each alternative is simply another legal expression. For example:

cat | dog | fish

You can have as many alternatives as you want in the set.

You can optionally specify a different language for the set of alternatives. To assure the correct precedence, you must use parentheses to group the alternatives; see “Grouping” below. You specify a
language with the same syntax as described in “Language” above. That is, you attach an exclamation point ( ! ) followed by the language identifier. These expressions say the same thing:

Jacques!fr | Renee!fr
(Jacques | Renee)!fr

You can optionally provide a weight for any number of the alternatives. A weight indicates how likely a particular alternative is. You specify a weight as a positive floating point number, such as 2, 2.5, 0.8, or .4. A weight of 1 is the same as not specifying a weight at all. A weight larger than 1 indicates that the alternative is more likely; a weight less than 1 indicates that the alternative is less likely.

You specify a weight by enclosing the weight in forward slashes ( / ) and putting it before the appropriate item in the alternatives list. For example:

/3.1415/ cat | dog | /.25/ fish
/10/ fish | /2/ (angel fish) | /.1/ anthia

The first of these says that it is quite likely the user will say "cat", less likely but still fairly likely the user will say "dog", and not very likely the user will say "fish". The second says the user will almost certainly say "fish", might say "angelfish", and probably will not say "anthia".

**Sequence**

A sequence is a set of expansions that must all be said in the order specified. Sequence is not a separate piece of syntax; rather, think of sequence as the natural order of things. That is, the interpreter always uses all of the expressions in the order presented, unless the expression specifies alternation. Some examples:

what is coral      // sequence of tokens
$question $subject // sequence of rule references
$subject is $type  // sequence of tokens and rule references

**Grouping**

Grouping is merely a way to treat a set of things as a single term. You use grouping, for example, to specify precedence or to attach a language identifier or repeat operator (next) to the whole group. Parentheses indicate a group. For example, consider the following expansion:

cat | dog fish

It is equivalent to:

cat | (dog fish)

This says to match "cat" or match "dog fish".

(cat | dog) fish

Putting parentheses in a different location results in a different interpretation—match "cat fish" or match "dog fish". While the parentheses are not required in the first situation, they might make it easier for a person to read the rule.
Repeats

You can specify that an expression should be repeated some particular number of times. The syntax for this allows you to specify a variety of repetition types:

<table>
<thead>
<tr>
<th>Syntax</th>
<th>Behavior</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;n&gt;</td>
<td>Repeat the expression exactly n times, where n is 0 or a positive integer. The example says to repeat the expression exactly 6 times.</td>
</tr>
<tr>
<td>&lt;6&gt;</td>
<td></td>
</tr>
<tr>
<td>&lt;n-m&gt;</td>
<td>Repeat the expression between n and m times, where n and m are both either 0 or positive integers and n is less than or equal to m. The example says to repeat the expression 4, 5, or 6 times.</td>
</tr>
<tr>
<td>&lt;4-6&gt;</td>
<td></td>
</tr>
<tr>
<td>&lt;n-&gt;</td>
<td>Repeat the expression n times or more, where n is 0 or a positive integer. The example says to repeat the expression 3 or more times.</td>
</tr>
<tr>
<td>&lt;3-&gt;</td>
<td></td>
</tr>
<tr>
<td>&lt;0-1&gt;</td>
<td>The expression is optional.</td>
</tr>
<tr>
<td>[...]</td>
<td>This is actually just a special case of the &lt;n-m&gt; syntax, but because it is such a common thing to do, ABNF provides the alternative syntax of enclosing the optional piece in square brackets.</td>
</tr>
</tbody>
</table>

The repeat operator appears after the expression to which it is attached. For example:

```
#ABNF 1.0;
root = $makeRequest;
$gear = mask | fins | snorkel | booties | gloves | regulator;
$action = buy | rent;
$polite = please | thank you;
$makeRequest = I want to $action $gear<1-5> [and $gear] $please<0-1>;
```

This set of rules matches a variety of utterances, including:

- I want to buy gloves.
- I want to rent mask and fins.
- I want to rent mask, fins, snorkel, and regulator, please.

Although you’re allowed to create a rule that recognizes an unbounded number of expressions, users do not actually speak forever. The speech recognition will proceed more effectively if you are more careful and only indicate a limited range of occurrences.

You can attach a probability to a repeat operator. The value indicates the probability of successive repetition of the repeated expression. A repeat probability must be in the range between 0.0 and 1.0; note that this is different from a weight attached to an entire expression.

A simple example is an optional expansion (zero or one occurrences) with a probability, for example, of 0.6. The grammar indicates that the chance that the expansion will be matched is 60% and that the chance that the expansion will not be present is 40%.

You can only use a repeat probability when specifying a range of repetitions. The syntax is:

```
<n-m /prob/>  
<n- /prob/>  
```

Here, n and m are integers and prob is the probability.
For example:

// the word "angel" is optional and is not very likely to occur.
angel <0-1 /0.25/> fish

// the rule reference $digit must occur between 2 and 4 times and
// it is very likely it will occur 3 or 4 times.
$digit <2-4 /.8/>

Tags (Semantic Interpretations)

In general, a tag is an arbitrary string that may be included inline within any rule expansion. You can include as many tags as you want within a single expansion. Tags do not affect what constitutes a legal utterance for a rule nor do they affect how the recognition proceeds.

You use tags to return information—a semantic interpretation—about a recognition to the element that invoked the grammar. Upon successful recognition, the BeVocal VoiceXML interpreter will create a JavaScript object whose properties and values are determined by the tags occurring in the matched rule.

If you want the BeVocal VoiceXML interpreter to make use of your tags for this purpose, they must be in one of the following formats:

```
{ property="value" }
{!{ property="value" }!}
```

where property and value are arbitrary names and values. For example, the following grammar sets two different tags:

```bash
#ABNF 1.0;
root $coloredObject;
$coloredObject = $color $object;
$color =
  (red | pink)   { color="red" }
  | (yellow | canary) { color="yellow" }
  | (green | khaki)  { color="green" };
$object =
  (truck | car)   { object="vehicle" }
  | (ball | block)  { object="toy" }
  | (shirt | blouse) { object="clothing" };
```

This grammar recognizes phrases such as "yellow shirt" or "canary blouse". For both of those phrases, it will return the same semantic interpretation:

```
{
  color: yellow;
  object: clothing;
}
```

This simple example allows your grammar to accept synonyms and return a more canonical result that can be used later in your VoiceXML application. See “Setting Input Variables” on page 9 for information on how the interpreter will use the semantic interpretation.

Precedence

The following is the ordering of precedence of rule expansions. Parentheses may be used to explicitly control rule structure.

- A rule reference, a quoted token, an unquoted token or a tag.
- Parentheses ( ( and ) ) for grouping and square brackets [ [ and ] ] for optional grouping.
Repeat operator (for example, \langle 0-1 \rangle) and language attachment (for example, !en-AU) apply to the tightest immediate preceding rule expansion. To apply them to a sequence or to alternatives, use parentheses or square brackets for grouping.

- Sequence of rule expansions.
- Set of alternative rule expansions separated by vertical bars ( | ) with optional weights.
This chapter describes the syntax for defining a grammar using the Nuance Grammar Specification Language (GSL) format. This is a plain-text (non-XML) representation, developed by Nuance.

For detailed information on this format, Nuance developers can find the Grammar Developer’s Guide at http://extranet.nuance.com. Nuance restricts access to its extranet to its partners and customers. Contact Nuance for access details.

Nuance’s Grammar Developer’s Guide describes how to create several different types of Nuance grammar—static grammars, just-in-time grammars, and other dynamic grammars. With the BeVocal VoiceXML interpreter, you cannot use static grammars or just-in-time grammars; you can only use other dynamic Nuance grammars.

Note: Nuance documentation uses slightly different terminology for various aspects of creating grammars than do the specifications for the other grammar formats supported by the BeVocal VoiceXML interpreter. In this document, we have primarily followed the terminology used by the W3C. In this chapter, we note the differences between that terminology and what you’ll find in Nuance documentation.

The structure and possible content of a grammar may be slightly different depending on whether the grammar occurs inline in the <grammar> tag of a VoiceXML document or externally as a separate grammar file referenced by a URI from the <grammar> tag.

A grammar consists of a header followed by a body. The header is information relevant to defining the grammar as a whole. The body is a set of rule definitions that are used to match against user input. All the rules defined in the grammar are active only when the grammar is active.

This chapter contains the following sections:

- Differences between Inline and External Definitions
- Comments
- Reserved Words
- Header Declarations
- Rule Definitions
- Rule Expansions

Differences between Inline and External Definitions

An inline grammar is defined completely within the <grammar> element in a VoiceXML document. An external grammar, on the other hand, is defined completely in an external file and referenced in the VoiceXML document.

In an external grammar document, all rules must be named. In an internal grammar, however, if the grammar consists of exactly 1 rule, that rule does not have to have a name. If an internal grammar consists of multiple rules, however, all of the rules must have names.

The preferred extension for an external GSL grammar file is .gsl; the interpreter also recognizes the .grammar extension. For a compiled Nuance Grammar Object, the only recognized extension is .ngo. For information on creating Nuance Grammar Objects, see “Compiled Grammar Files” on page 24.
When you include a grammar definition directly in your VoiceXML document, you must take special care. GSL grammars use special characters not normally permitted in XML documents, such as angle brackets (< and >). Consequently, you should wrap your inline grammar as a section of CDATA:

```
<grammar ...usage attributes...>
  <!--[CDATA[
    ...grammar header...
    ...grammar rule definitions...
  ]]>}
</grammar>
```

On the other hand, if you have an external GSL grammar file, the contents of that file should *not* be inside a CDATA section. Also, an external file should not contain a `<grammar>` element. So, an external grammar file simply looks like:

```
;GSL2.0
...grammar rule definitions...
```

The following sections describe exactly what goes in the header and body.

---

### Comments

Comments may be placed anywhere in a grammar. You indicate the start of a comment with a semi-colon. Everything on the line following the semi-colon is ignored:

```
; this is a comment
RuleName [rule expansion] ; this is a comment, after a non-comment
```

### Reserved Words

The following strings are reserved. You cannot use them in your grammar as tokens or to define rule names:

- `AND-n`, `OR-n`, `OP-n`, `KC-n`, `PC-n`

where *n* is an integer.
Header Declarations

In most grammar formats, the header contains information relevant to the grammar as a whole. The GSL grammar format provides only a minimal amount of information in the header. So that you can easily compare this grammar format with others, this section describes both the header information available in GSL and what your options are for things it does not support but that are supported by ABNF and XML grammars.

Self-Identifying Header

Your GSL grammar can start with a self-identifying header, indicating the version of the grammar format. If it does, currently the header must be:

;GSL2.0

At present, this header is optional. For an external grammar file, it is strongly recommended that you include the header, however, as later versions may require it. For an inline GSL grammar, if your grammar is only a single rule fragment, do not include the header; if it is named rules, then you can include the header.

Language

The ABNF, XML, and JSGF grammar formats allow you to specify the language contained in the grammar. The GSL grammar format does not provide a way to specify the language of the grammar. However, if you refer to a GSL grammar with a rule reference in another grammar, you can specify the language of the GSL grammar in the rule reference. For information on how to do this in XML, see “Rule References” on page 40; in ABNF, see “Rule References” on page 57.

Grammar Mode

Both the ABNF and XML grammar formats restrict a single grammar to being either for recognizing speech input or for recognizing DTMF input. Those formats do not allow a single grammar to include both speech and DTMF; the mode of the grammar indicates whether it is for speech or for DTMF.

GSL does not have this restriction. Any GSL grammar can include both DTMF and speech tokens in the same grammar and even in the same rule.

Root Rule

A top-level grammar rule is one that the interpreter can use as the top-level syntax for matching spoken input. Nuance refers to a top-level rule in a grammar as the grammar and refers to other rules that occur in the grammar as subgrammars or grammar rules.

As described in Chapter 1, “Using VoiceXML Grammars”, a reference to a grammar may or may not specify a top-level rule. Consequently, all grammars need a way to specify the rule for the interpreter to use when you don’t name one; this is called the root rule of the grammar. ABNF and XML allow you to explicitly name a rule as the root rule. In a GSL grammar, on the other hand, the root rule is always the first public rule in the grammar. For information on public rules, see “Scoping of Rule Definitions” on page 67.

Tag Format

The ABNF and XML grammar formats allow you to specify a document that describes the allowed format for semantic interpretations, or tags, in that format.

In GSL, this format is not under the control of the grammar writer. The GSL grammar format specifies the syntax for semantic interpretations provided in that grammar format. In GSL, they are called assignment commands; see “Assignment Commands (Semantic Interpretations and Rule Variables)” on page 73.
Pronunciation Lexicon

The ABNF and XML grammar formats allow you to point to a lexicon document to provide extra pronunciation information to the interpreter.

GSL does not have syntax for pointing to a lexicon document.

Meta and HTTP-Equiv

The ABNF and XML grammar formats have syntax for providing metadata about the grammar itself. GSL does not have syntax for providing metadata.

Rule Definitions

The body of a grammar consists of a set of rule definitions or, simply, rules. The format of rule definitions can be slightly different, depending on whether they appear inline or in an external file.

Each rule definition can associate a rule name with a rule expansion or grammar description. The definition also defines the scope of the rule—whether it is local to the grammar in which it is defined or whether it may be referenced within other grammars.

Other grammar formats have special syntax for you to provide examples of matching user utterances. GSL does not have special syntax for this purpose. You simply put your examples in GSL comments.

Basic Rule Definition

A rule can have two parts:

- A rule name (optional) that identifies the rule for use in other rules.
- A rule expansion (required) that defines the possible utterances associated with that rule.

You create rule expansions using a set of operators to combine words and/or other expressions. You use grammar assignment commands to associate an assignment operation to a particular rule expansion, either the top expression in the rule, or a component expression.

Because most grammars in VoiceXML identify a set of possible words that a user might say, the top-level rule expansion in a grammar rule is usually enclosed in square brackets ( [ ] ) to represent alternatives.

The following example shows a simple named grammar rule:

```
Fish [garibaldi "nassau grouper" trout]
```

This rule is named Fish and the rule expansion is the set of alternatives. This grammar is matched if the user says “Garibaldi”, "Nassau Grouper”, or “Trout”.

Rule Names

The rule name is simply a string that identifies the rule. The rule name for each rule definition must be unique within the grammar. The same rule name may be used in multiple grammars.

In an external grammar, all rules must have names. In an internal grammar, if the grammar consists of exactly one rule, that rule does not need to have a name. In all other cases, all rules must have names.

In a rule reference (described in “Rule References” on page 70), you can use the rule name to specify a particular rule from which to start recognition.

A rule name is a case-sensitive string that starts with an uppercase letter. You can use the following characters within rule names:

- upper case or lower case letters
• digits
• - (hyphen), _ (underscore), ' (single quote), @ (at sign), and . (period).

Special Rules

GSL has 3 predefined rules:
<special:passthrough>
<special:roadblock>
<special:resistor>

NULL is a synonym for <special:passthrough> and VOID is a synonym for <special:roadblock>. Your grammar must not contain rules with these names. The special rules have the following meanings:

• <special:passthrough>, NULL
  Defines a rule that matches if the user doesn't say anything. For example, you can use the following grammar to match "queen", "picasso", or silence.
  
  Trigger = [NULL queen picasso]
• <special:roadblock>, VOID
  Defines a rule that can never be spoken. Inserting VOID into a sequence automatically makes that sequence unspeakable. For example, nothing a user says can match this rule:
  
  Trigger = (VOID [queen picasso])
• <special:resistor?weight=n>
  Changes the probability that a rule can be spoken. The weight, n, is a floating point number representing the probability. A weight less than 1 reduces the probability of matching the rule; the closer to 0, the less probable that the rule will be matched. A weight greater than 1 increases the probability of matching the rule.

The format you use to refer to a special rule is a particular instance of a rule reference (see “Rule References” below).

You can use the NULL and VOID rules together to dynamically change whether or not a rule is active. For example, assume your chamber of commerce application provides information that varies from season to season. During the winter, it answers questions about the amount of snow at the local ski resorts; during the summer, it answers questions about the hours of the local amusement parks; all year round, it answers questions about the plays at local theaters. You don't want the application to recognize questions about ski resorts during the summer.

You could completely change your grammars at every season. This might be difficult to maintain. You could ease the problem by organizing your grammars with a judicious combination of VOID and NULL rules. You would use the VOID and NULL rules to turn on and off recognition of the appropriate questions at the appropriate times of year.

If instead of completely not recognizing out of season requests, you simply want to indicate that those requests are much less likely, you could use of the special:resistor rule instead of VOID.

Scoping of Rule Definitions

Each defined rule has a scope of either private or public.

• A rule with public scope is visible outside its grammar. A rule with public scope can be activated for recognition; that is, it can define the top-level syntax of spoken input. For example, a rule reference can specify a public rule by name.
• A rule with private scope is visible only within its containing grammar and may be referenced only by other rules within the same grammar.

To mark a rule as public, the format is:
There is not a special syntax for marking a rule as private. If no rules in the grammar are explicitly marked with :public, then all rules in the grammar are public. On the other hand, if any rule in the grammar is marked with :public, then all public rules must be so marked. In this case, rules that do not have :public are private.

The root rule in a GSL grammar is always the first public rule. In a grammar that has no rules explicitly marked as public (and so all rules are public), this means that the first rule in the grammar is the root rule. If the grammar explicitly marks some rules as public, the root rule is the first rule so marked.

For example, the following set of definitions creates one public rule named Snapper and two private rules named SnapperType and FishColors:

```
SnapperType [mutton FishColors]
FishColors [black gray red]
Snapper:public (SnapperType snapper)
```

The root rule of this grammar is Snapper.

You should only make public the rules in your grammar that you want to be visible to other grammars. The use of private and public scope allows you to write more modular and maintainable grammars. For example, you can define a grammar that has a set internal "worker" rules that are combined to provide a smaller number of externally-accessible rules. Hiding worker rules prevents their accidental misuse.

**Note:** Do not confuse the scope of a rule with the scope of its containing grammar. The scope of the grammar indicates where in the VoiceXML application the grammar is active. The scope of the rule indicates whether or not the rule is available to other active grammars.

### Rule Recursion

You can write a rule that refers to itself either directly or indirectly through a rule that it references.

```
; Rule that refers to itself directly
Digits [Digit (Digit Digits)]
Digit [0 1 2 3 4 5 6 7 8 9]

; Rule that indirectly refers to itself
NounPhrase (Noun ?PrepositionalPhrase)
PrepositionalPhrase (Preposition NounPhrase)
```

You should be careful in writing rules that refer to themselves. GSL does not support left-recursive rules. That is, it does not support defining a rule whose first sequential subcomponent contains itself:

```
; Legal
Digits [Digit (Digit Digits)]
Digits [(Digit Digits) Digit]

; Illegal
Digits [(Digits Digit) Digit]
Digits [Digit (Digits Digit)]
```

This restriction ensures that the interpreter doesn’t get lost down an infinite path trying to match a rule.
A rule expansion is the part of a rule definition that actually describes what utterances match the rule. A rule expansion is a token, a rule reference, a semantic interpretation, or an arbitrarily complex combination of these things. For example, a rule expansion might express any of these ideas:

- Match the phrase "North Dakota"
- Match the name of any of the 50 United States
- Match a phrase that optionally starts with politeness words (like "please") is followed by an action phrase (like "I want to go to") and then by a destination (like "Fargo, North Dakota") and possibly ends with another politeness phrase (like "thank you") and return just the destination

Tokens and rule references must be separated from each other by whitespace. Whitespace is optional between operators and their operands.

Language

The ABNF, XML, and JSGF grammar formats allow you to specify the language for a rule; the GSL grammar format does not. However, if you refer to a particular rule in a GSL grammar with a rule reference in another grammar, you can specify the language of the GSL rule in the rule reference. For information on how to do this in XML, see “Rule References” on page 40; in ABNF, see “Rule References” on page 57.

Tokens

A token is the part of a rule expansion that actually mentions a word that a user might speak or a DTMF key a user might press. Unlike the other grammar formats supported by the BeVocal VoiceXML interpreter, in GSL you can mix DTMF tokens and voice tokens in the same rule.

Voice Tokens

You can use the following characters within voice tokens, without requiring double quotes:

- lower case letters only
- digits
- the special characters - (hyphen), _ (underscore), ' (single quote), @ (at sign), and . (period).

If you enclose a token in double quotes, you can use other special characters as well, with the exception of whitespace characters. For example, "new^york" is a valid token, but "new york" is not.

To improve the portability of your grammar, you should follow some simple rules:

- Avoid acronyms. For example, replace "USA" with "u s a" and replace "VXML" with "v x m l".
- Expand abbreviations. For example, replace "st." with "street" or "saint" and replace "dr." with "doctor" or "drive".
- Spell out punctuation. For example, replace "." with "period" or "dot" and replace "<" with "less than" or "open angle bracket".
- Spell out numbers larger than 9. For example, replace "10" with "ten" and "6031" with "sixty thirty one" or "six thousand thirty one".
DTMF Tokens

You can express valid touchtone sequences in a rule expansion using the following DTMF notation.

<table>
<thead>
<tr>
<th>Key Press</th>
<th>DTMF Notation</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>dtmf-0</td>
</tr>
<tr>
<td>1</td>
<td>dtmf-1</td>
</tr>
<tr>
<td>2</td>
<td>dtmf-2</td>
</tr>
<tr>
<td>3</td>
<td>dtmf-3</td>
</tr>
<tr>
<td>4</td>
<td>dtmf-4</td>
</tr>
<tr>
<td>5</td>
<td>dtmf-5</td>
</tr>
<tr>
<td>6</td>
<td>dtmf-6</td>
</tr>
<tr>
<td>7</td>
<td>dtmf-7</td>
</tr>
<tr>
<td>8</td>
<td>dtmf-8</td>
</tr>
<tr>
<td>9</td>
<td>dtmf-9</td>
</tr>
<tr>
<td>*</td>
<td>dtmf-star</td>
</tr>
<tr>
<td>#</td>
<td>dtmf-pound</td>
</tr>
</tbody>
</table>

The following rule expansion allows either spoken or telephone keypad input:

```
[(john smith)(dtmf-4 dtmf-9 dtmf-7 dtmf-3)]
{<emp john_smith>}
```

The user can either say “John Smith” or use the touchtone sequence 4973 to create the same semantic interpretation for employee John Smith.

Rule References

Every rule in a grammar can have a name; if it does, that name is unique within the grammar. You can refer to a rule by name from within another rule either in the same or a different grammar definition. In summary, the formats for doing so are:

<table>
<thead>
<tr>
<th>Reference to a…</th>
<th>Format</th>
</tr>
</thead>
<tbody>
<tr>
<td>Local rule</td>
<td>RuleName</td>
</tr>
<tr>
<td>Local rule with a variable</td>
<td>RuleName:variableName</td>
</tr>
<tr>
<td>Root rule of a grammar identified by a URI</td>
<td>&lt;grammarURI&gt;</td>
</tr>
<tr>
<td>Named rule of a grammar identified by a URI</td>
<td>&lt;grammarURI#RuleName&gt;</td>
</tr>
<tr>
<td>Special rules</td>
<td><a href="">special:passthrough</a>, NULL</td>
</tr>
<tr>
<td></td>
<td><a href="">special:roadblock</a>, VOID</td>
</tr>
<tr>
<td></td>
<td><a href="">special:resistor?weight=n</a></td>
</tr>
</tbody>
</table>

See “External Grammar Files” on page 4 for a description of how relative URIs work in grammar files. See “Special Rules” above for information on the special rules.
When a grammar rule contains a rule reference, the effect is the same as if the referenced rule’s rule expansion appeared in place of the rule name. For example, this `PrimaryColors` grammar rule refers to the `Shades` rule:

```
PrimaryColors ( ?Shades [red blue green])
Shades [ dark light ]
```

The `PrimaryColors` rule could also be written as:

```
PrimaryColors [ ( ?[dark light] [red blue green] ) ]
```

**Local Reference**

A local rule reference is a reference to another rule in the same grammar. A local rule reference has one of the following two forms:

- `RuleName`
- `RuleName:variableName`

Both forms reference the rule named `ruleName`. The second form additionally specifies a variable named `variableName` to be set to any value returned by that rule. See “Assignment Commands (Semantic Interpretations and Rule Variables)” on page 73 for information on variables.

An expansion can reference any grammar rule defined in the grammar that contains the rule expansion.

**External Reference**

You can reference rules defined in a different grammar. Here, the other grammar must be an external grammar which you can reference by its URI. You can optionally specify a specific rule in that grammar to use to start recognition. If you do not specify a rule to use, then the reference grammar must itself specify a root rule. The formats are:

- `<grammarURI>`
- `<grammarURI)#RuleName>`

For example:

- `; Reference to the root rule of an external grammar <http://www.myCompany.com/grammars/fish.gram>`
- `; Reference to a specific rule of an external grammar <../fish.gram#butterflies>`

Note that you cannot use a rule reference in a GSL grammar to refer to a built-in grammar of the BeVocal VoiceXML interpreter. To refer to these built-in grammars, you must use the syntax described in Chapter 2, “Using Built-in Grammars”.
Combinations

GSL grammars combine tokens and rule references into more complex expressions. The basic types of combination are:

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alternation</td>
<td>A set of alternatives (&quot;cat&quot; or &quot;dog&quot; or &quot;goat&quot;). Enclosed in square brackets ([ ]).</td>
</tr>
<tr>
<td>Sequence</td>
<td>Multiple expressions that must all be said in a particular order (&quot;great dane&quot;). Enclosed in parentheses ( ( ) ).</td>
</tr>
<tr>
<td>Repetition</td>
<td>Repeat a single expression some number of times (&quot;very&quot; or &quot;very very&quot; or...). The * and + operators.</td>
</tr>
<tr>
<td>Optional</td>
<td>Special case of repeat 0 or 1 times (the &quot;kitty&quot; in match &quot;kitty cat&quot; or &quot;cat&quot;). The ? operator.</td>
</tr>
<tr>
<td>Weighting</td>
<td>Likelihood that an expression will be said. Indicated with ~ operator.</td>
</tr>
</tbody>
</table>

Alternatives

A set of alternatives matches if the caller says one of the things in the set. You specify a set of alternatives by surrounding them with square brackets ([ ]). Each alternative is simply another legal expression. For example:

[ cat dog fish ]

You can have as many alternatives as you want in the set.

Sequence

A sequence is a set of expansions that must all be said in the order specified. You specify a set of alternatives by surrounding them with parentheses ( ⟨ ⟩ ). Some examples:

(what is coral) // sequence of tokens
(Question Subject) // sequence of rule references
(Subject is Type) // sequence of tokens and rule references

Repeats

You can specify that an expression should be repeated some particular number of times. The syntax for this allows you to specify a variety of repetition types:

<table>
<thead>
<tr>
<th>Syntax</th>
<th>Behavior</th>
</tr>
</thead>
<tbody>
<tr>
<td>?expr</td>
<td>The expression is optional.</td>
</tr>
<tr>
<td>*expr</td>
<td>Repeat the expression 0 or more times.</td>
</tr>
<tr>
<td>+expr</td>
<td>Repeat the expression 1 or more times.</td>
</tr>
</tbody>
</table>

The repeat operator appears before the expression to which it is attached. For example:

Gear = [mask fins snorkel booties gloves regulator]
Action = [buy rent]
MakeRequest = (I want to Action +Gear ?(and Gear) )

This set of rules matches a variety of utterances, including:

I want to buy gloves.
I want to rent mask and fins.
I want to rent mask, fins, snorkel, and regulator.
Although you’re allowed to create a rule that recognizes an unbounded number of expressions, users do not actually speak forever. The speech recognition will proceed more effectively if you are more careful and only indicate a limited range of occurrences. For example, the MakeRequest rule above allows the Gear rule to occur at least once, but any number of times more than that. It’s unlikely that a user will request the same item more than once, so you could change the rule to allow Gear to be matched between 1 and 5 times:


Weights or Probabilities
You can optionally assign transition weights, or probabilities for any number of the alternatives. A weight indicates how likely a particular alternative is. You specify a weight as a positive floating point number, such as 2, 2.5, 0.8, or .4. A weight of 1 is the same as not specifying a weight at all. A weight larger than 1 indicates that the alternative is more likely; a weight less than 1 indicates that the alternative is less likely. All probabilities in the set of alternatives are normalized to add to 1.0. If you don’t specify any probabilities, all expressions are considered to be equally likely.

You specify a probability as follows:

\[ \text{grammarExpression-probability} \]

For example:

\[
[\text{cat-3.1415 dog fish-.25}]
[\text{fish-10 (angel fish)-2 anthia-0.1}]
\]

The first of these says that it is quite likely the user will say "cat", less likely but still fairly likely the user will say "dog", and not very likely the user will say "fish". The second says the user will almost certainly say "fish", might say "angelfish", and probably will not say "anthia".

You can also attach a probability to a repeat operator. The value indicates the probability of successive repetition of the repeated expression. Attaching a probability to a repeat operator is slightly different than attaching it to a standard expression:

<table>
<thead>
<tr>
<th>Syntax</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>?expr-.75</td>
<td>It is 75% certain that expr will be said once.</td>
</tr>
<tr>
<td>+expr-.75</td>
<td>Says nothing about the probability of expr being said the first time (since it must be said at least once). But, after expr has been said once, it is 75% certain that it will be repeated a second time. If it is said a second time, it is 75% certain it will be repeated a third time. And so on.</td>
</tr>
<tr>
<td>*expr-.75</td>
<td>As soon as this expression can be matched, it is 75% certain that it will be said a first time. Then, it is 75% certain that it will be repeated a second time. If it is said a second time, it is 75% certain it will be repeated a third time. And so on.</td>
</tr>
</tbody>
</table>

For example:

; the word "angel" is optional and is not very likely to occur.
?angel-0.25 fish

; the rule reference can occur 0 or more times and it is very likely it will occur more than once.
*Digit-.8

Assignment Commands (Semantic Interpretations and Rule Variables)
You use grammar assignment commands to associate an assignment operation with a particular rule expansion, either the top expression in the rule, or a component expression. The assignment commands
for a rule expansion are enclosed in curly braces ({} ) and immediately follow the associated rule expansion. The curly braces can enclose one or more individual assignment commands.

Each time the speech-recognition engine encounters a word or phrase that matches an expression in an active grammar, it flags a match and executes any assignment command associated with the expression.

An individual assignment command is either a slot-filling command or a return command:

- **A slot-filling command** specifies a value to return from the recognition; it has the form:

  `<slotname value>
  
  A slot-filling command is a semantic interpretation which returns information about a recognition to the VoiceXML element that invoked the grammar. Upon successful recognition, the BeVocal VoiceXML interpreter will create a JavaScript object whose properties and values are determined by the slot-filling commands occurring in the matched rule.

- **A return command** returns a value from a subrule to the rule that invoked it; it has the form:

  `return (value)
  
  A return command is useful when the rule reference to the subrule specifies a variable. When the subrule is matched, the corresponding variable is set to the value returned by the return command.

  **Note:** A grammar rule can include a return command only if it is always used as a subrule. If the rule is ever used as a top-level grammar rule it must not include a return command.

The **value** expression in an assignment command can be a literal value, a variable expression, or a function call, as described in sections below.

### Slot-Filling Command Example

The following grammar has 2 different slot-filling commands:

```gsl
;GSL 2.0;
ColoredObject:public (Color Object)
Color {
  [red pink]   { <color red> }
  [yellow canary] { <color yellow> }
  [green khaki]   { <color green> }
}
Object {
  [truck car]   { <object vehicle> }
  [ball block]   { <object toy> }
  [shirt blouse] { <object clothing> }
}
```

This grammar recognizes phrases such as "yellow shirt" or "canary blouse". For both of those phrases, it will return the same semantic interpretation:

```javascript
{
  color: yellow;
  object: clothing;
}
```

The VoiceXML interpreter will use this semantic interpretation to set the slots of active elements. This simple example allows your grammar to accept synonyms and return a more canonical result that can be used later in your VoiceXML application. See “Setting Input Variables” on page 9 for information on how the interpreter uses the semantic interpretation.

### Return Command Example

The following example returns a value from the City subrule to the Flight rule:

```gsl
Flight:public [
```
Rule Expansions

( [from leaving] City:frCity ) { <origin $frCity> }
( [to (arriving in)] City:toCity ) { <destination $toCity> }

City [
  atlanta { return ("Atlanta") }
  chicago { return ("Chicago") }
  dallas { return ("Dallas") }
]

Literal Values

A *literal value* is an integer or a quoted or unquoted string. A literal value is evaluated as follows:

- A quoted string evaluates to itself.
- An integer evaluates to its integer value.
- An unquoted string that has no integer interpretation evaluates to itself.

As the preceding list suggests, values that can be interpreted as integers are treated as integers. For example, the value 01 in the following grammar rule can be interpreted as an integer:

DigitValue [
  ([zero oh] one) { return (01) }
  ... ]

If the user says "oh one", the DigitValue rule will return the integer 1.

If you want an integer value to be treated as a string, you can enclose it in double quotes, for example:

DigitString [
  ([zero oh] one) { return ("01") }
  ... ]

If the user says "oh one", the DigitString rule will return the string "01".

Variable Expressions

As shown in the return command example, if a rule includes a rule reference to a subrule and that rule reference specifies a variable *variableName*, an assignment command of the containing rule can use a variable expression of the form:

$variableName

This expression evaluates to the value returned from the referenced rule.

A special variable, *string*, is used to capture the portion of the utterance that matched a rule expansion; its value can be obtained with expression:

$string

For example, the following rule accepts the name of a day of the week and sets the *day* property to the word that the user said:

DayOfWeek [
  sunday monday tuesday wednesday
  thursday friday saturday
] { <day $string>}
Function Calls

You can use a *function call* to an arithmetic or string function to compute the value in an assignment command. The parameters to the function can be value literals, variable expressions, or other function calls.

The following integer and string functions are available for use in value expressions. The third column indicates the default value for a parameter that is undefined.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
<th>Default Parameter Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>add</td>
<td>Adds two integers.</td>
<td>0</td>
</tr>
<tr>
<td>sub</td>
<td>Subtracts the second integer from the first.</td>
<td>0</td>
</tr>
<tr>
<td>mul</td>
<td>Multiplies two integers together.</td>
<td>1</td>
</tr>
<tr>
<td>div</td>
<td>Divides the first integer by the second and returns the truncated result.</td>
<td>0 for first parameter; 1 for second parameter</td>
</tr>
<tr>
<td>neg</td>
<td>Returns the negative or positive inverse of an integer.</td>
<td>0</td>
</tr>
<tr>
<td>strcat</td>
<td>Concatenates two strings, but may be nested to effectively concatenate any number of strings, for example: strcat($a1 strcat($a2 $a3))</td>
<td>&quot; &quot; (empty string)</td>
</tr>
</tbody>
</table>

The following example illustrates how functions can be used to compute the value of the number input variable from values returned by subrules.

TwoDigit [  
(Digit:n1 Digit:n2) { <number add((mul(10 $n1)) $n2) }  
TeenAndTen:n { <number $n> }  
(DecadeFromTwenty:n1 ?NonZeroDigit:n2) { <number add($n1 $n2)> } ]

Digit [  
[zero oh] { return(0) }  
NonZeroDigit:d { return($d) } ]

NonZeroDigit [  
one { return(1) }  
two { return(2) }  
...  
nine { return(9) } ]

TeenAndTen [  
ten { return(10) }  
eleven { return(11) }  
...  
nineteen { return(19) } ]
DecadeFromTwenty [ 
  twenty { return(20) }
  thirty { return(30) }
  ...
  ninety { return(90) }
]

Notice in the TwoDigit rule, the third rule expansion has an optional element. If the user said “twenty”, the value of \( n_2 \) would be undefined. Since the default value for the add function is zero, the result will be correct \((20 + 0)\).
Currently, the BeVocal VoiceXML interpreter also supports the Java Speech Grammar Format (JSGF).

**Note:** Support for this grammar format may be deprecated in a future release.

See [http://www.w3c.org/TR/jsgf/](http://www.w3c.org/TR/jsgf/) for a description of the Java Speech Grammar Format. See Appendix D of the VoiceXML 1.0 specification for a discussion of JSGF as a VoiceXML grammar format.

BeVocal VoiceXML currently provides Beta-level support for JSGF grammars, with several limitations. The implementation does not support the `import` statement or semantic tags and does not handle inline grammar fragments. If you use inline JSGF grammars, you must use a complete, syntactically-correct JSGF grammar. For example:

```xml
<grammar>
  <![CDATA[
    #JSGF 1.0;
    grammar topping;
    public <topping> =
    ( pepperoni | sausage | mushrooms );
  ]]>]
</grammar>
```

The recognized extension for an external JSGF grammar file is `.jsgf`. 
Nuance SayAnything grammars allow users to speak freely to an application and to have their sentences interpreted without the application developer having to write complex grammar rules covering the entire sentence. A SayAnything grammar is created using a combination of statistical language models (SLM) and robust natural language interpretation technologies.

The BeVocal VoiceXML interpreter supports using SayAnything grammars in your application and provides several properties to improve recognition performance with a SayAnything grammar. However, the BeVocal Café development environment does not provide tools to help you create these grammars. To create a SayAnything grammar, you must do so with the Nuance 8.0 tools. Once you have created your SayAnything grammar, you can use it within your VoiceXML application by specifying the grammar as the value of the `<grammar>` tag’s `src` attribute.

**Note:** Creating an efficient SayAnything grammar is a complex task, requiring a lot of speech expertise and in-depth knowledge of Natural Language Semantics. There are many steps involved in creating and fine-tuning these grammars. Please contact Bevocal Professional Services for details or for help creating commercial-grade grammars specific to your application.

**Introduction to SayAnything Grammars**

Your application may be best served with an open-ended prompt such as "How can I help you?". In this situation, not only can the user’s response be highly variable and hard to predict, but it can also contain disfluencies—things such as restarts, filled pauses (um and uh), and ungrammatical sentences. In addition, the grammar for this state must fill several slots from only one utterance. The challenge is to write a grammar that lets callers say an arbitrary phrase within the domain of the task, fills many slots, and still achieves high accuracy.

Using a normal grammar and devising a comprehensive set of grammar rules may be impractical. In most cases, the out-of-grammar rate obtained with handcrafted grammar rules is very high and any attempt to add more rules often leads to poor in-grammar recognition accuracy.

For an example of using a SayAnything grammar, see the VoiceXML Samples page.

**SLM Grammars**

A language model that assigns probabilities to sequences of words is called a statistical language model (SLM).

A very simple form of an SLM is a list of words with probabilities assigned to each; this is commonly called a unigram. A unigram is overly simplistic model for many reasons, primarily because the probability of a word is independent of its position in a sentence. In this model, for example, a sentence such as "I want to travel to Boston next Monday" is considered as likely as any permutation of its words, such as "To want to travel to Boston I next Monday".

In contrast, an n-gram SLM is one in which the probability of a word depends on the previous N-1 words; N is called the order of the model.

Unlike a normal grammar, an n-gram SLM grammar is not manually written but instead is trained from a set of examples that models the user’s speech. To train an SLM grammar, you pass this set of examples (and optionally a domain-specific vocabulary) to a utility, which estimates the model probabilities. Because the
probability assigned to a phrase depends on the context and is driven by data, an SLM provides a grammar in which more plausible phrases are given higher probabilities.

SLMs are useful for recognizing free-style speech, especially when the out-of-grammar rate is high. SLMs are not meant to replace normal grammars, which are quite suitable when the application’s prompts are sufficient to restrict the user’s response. Since SLMs need a large set of examples to train, a data collection system or a pilot based on a normal grammar must often be developed to gather training examples.

Robust Natural Language Interpretation

SLMs get started on the understanding free-style speech by recognizing an wider range of user utterances. However, recognizing the utterance is only half the battle. The other half is interpreting the meaning of the utterance. To do that, you still need to fill slots with the appropriate values. Writing these grammar rules can be a tedious task and can defeat the advantages of using an SLM in the first place.

To address this problem, BeVocal platform offers a robust natural language (NL) parsing capability that lets you write slot-filling grammars that only consider the meaningful phrases in an utterance, ignoring the parts that don’t matter. The robust NL interpretation engine spots meaningful phrases in the text output of the recognizer and fills appropriate slots.

Natural language interpretation has 2 modes—full and robust. In the conventional operating mode, the full mode, the recognition engine and the NL engine are driven by the same grammar. This grammar both defines the valid phrases and how the slots are filled.

In robust mode, you can use two different grammars. The first grammar drives the recognition phase and the second drives the interpretation phase. For example, the recognition could use an SLM grammar, allowing the application to recognize a large range of user’s speech. The text output by the recognition engine could then by processed by the NL engine running a grammar that parses certain phrases only—the meaningful ones—and fills the appropriate slots.

While the conventional NL parser requires a “full” parse of the speaker’s sentence by a top-level grammar rule—all the words in the sentence must be matched by a single grammar rule—the robust parser eliminates this requirement. If a full parse is not found, the robust NL parser attempts to fill as many slots as it can from partial parses, by using subsentence phrase fragments.

For example, consider a grammar for a typical flight reservation application and the following user’s sentence:

I’d like to um I want to go to boston tomorrow.

The speech recognition engine, driven by the SLM, recognizes the sentence and sends the result to the NL engine, which tries to interpret the text. In full mode, the NL engine would not parse the text, because the sentence cannot be completely parsed by the grammar; consequently, it would reject the sentence and fill no slots. However, in robust mode, the engine could ignore the babbling at the beginning of the sentence and fill the destination and day slots with boston and tomorrow, respectively.
Controlling SayAnything Grammars

Once you have your SayAnything grammar ready, you can use the following properties of the VoiceXML interpreter to help improve recognition within the grammar.

**bevocal.grammar.interpretationtype**

*Extension*. The `bevocal.grammar.interpretationtype` property specifies whether to use the NL engine in standard mode (`full`) or in robust mode (`robust`). Setting this property to `robust` facilitates the NL engine’s interpretation of more spontaneous utterances from SLM grammars.

**Note:** This property is relevant only when recognizing against SLM grammars. It should always be set to its default value when recognizing against conventional grammars.

The default value is `full`.

**bevocal.grammar.phoneticpruning**

*Extension*. The `bevocal.grammar.phoneticpruning` property specifies whether the recognizer should perform phonetic pruning. For SLM grammars, set this parameter to `true` except for grammars with small vocabularies.

**Note:** This property is relevant only when recognizing against SLM grammars. It should always be set to its default value when recognizing against conventional grammars.

The default value is `false`.

**bevocal.grammar.weightfactor**

*Extension*. The `bevocal.grammar.weightfactor` property controls the relative weighting of acoustic and linguistic scores during recognition.

As this value increases, the recognizer runs faster and hence the value of the `speedvsaccuracy` property should be increased to get better recognition.

The corresponding speech engine property is in the range between 0 and 100. For well-trained SLM grammars, the optimum value is between 0.58 and 0.6, corresponding to the range of 9-10 in the speech engine.

**Note:** This property is relevant only when recognizing against SLM grammars. It should always be set to its default value when recognizing against conventional grammars.

The default value is 0.5. This maps to a setting of 5 in the speech engine.

**bevocal.grammar.wordtransitionpenalty**

*Extension*. The `bevocal.grammar.wordtransitionpenalty` property controls the word transition weight. This is the trade-off between inserted and deleted words. For SLM-based grammars, the optimal value is in the range 0 to –50. For conventional grammars, you should leave this property set to its default value.

**Note:** This property is relevant only when recognizing against SLM grammars. It should always be set to its default value when recognizing against conventional grammars.

The default value is -200.
The BeVocal VoiceXML Voice Enrollment facility allows the interpreter to convert the user’s utterance directly to an ABNF grammar, without going through the intermediate text representation, and to recognize utterances against that grammar. This chapter describes:

- Text-Based Grammars
- Limitations of Text-based Grammars
- Enrollment Basics
- Usage Model
- Recognizing Enrolled Grammars
- Deleting Enrolled Grammars

Note: The Voice Enrollment facility is an experimental extension to VoiceXML; its implementation and behavior are subject to change. The current BeVocal VoiceXML implementation contains the feature before it has been standardized so that developers may provide feedback. If this capability becomes a standard part of a future version of VoiceXML, the BeVocal VoiceXML implementation will change as necessary to match the VoiceXML standard.

Text-Based Grammars

When a VoiceXML application interacts with its user, the application typically plays some prompts, waits for the user to speak, and then recognizes the user’s utterance against one or more active grammars. As described earlier, these grammars can be explicitly specified by you in one of the standard grammar languages or can be implicitly provided by a built-in field or grammar type. For example, the following snippet of VoiceXML uses an inline ABNF grammar followed by a built-in number grammar.

```
<form>
  <field>
    Say a color
    <grammar>
      #ABNF 1.0 en-US;
      root $color;
      $color = red | green | blue;
    </grammar>
  </field>
  <field>
    Say a number
    <grammar src="builtin:grammar/number"/>
  </field>
</form>
```

These types of text-based grammars give enough flexibility for the vast majority of applications. Because grammars can be loaded from external URLs and those external URLs can point to JSPs or CGIs that generate the grammars dynamically, you can write an application even when the exact grammars that are needed won’t be known until runtime.
Limitations of Text-based Grammars

The one limitation to text-based VoiceXML grammars is that they can’t handle recognition of phrases for which you do not have a text representation. For example, imagine the following user interaction:

<table>
<thead>
<tr>
<th>User</th>
<th>Add to address book</th>
</tr>
</thead>
<tbody>
<tr>
<td>Computer</td>
<td>What name would you like to add?</td>
</tr>
<tr>
<td>User</td>
<td>Jane Smyth</td>
</tr>
<tr>
<td>Computer</td>
<td>What is the phone number for Jane Smyth?</td>
</tr>
<tr>
<td>User</td>
<td>four zero eight five five one two three four</td>
</tr>
<tr>
<td>User</td>
<td>Call Jane Smyth</td>
</tr>
</tbody>
</table>

[Computer dials 408-555-1234]

In this application, the user’s address book grammar needs to be generated dynamically at runtime. However, there’s a catch. The application can’t simply generate a grammar with the text “jane smyth” in it, because it doesn’t know that’s the text it should use.

The application could use the `<record>` tag to get a WAV file containing the name of the person to be added to the address book, but there is still no way to transcribe the sounds in the WAV file into the text to insert in a grammar. In general, VoiceXML provides no speech-to-text facility for converting arbitrary recordings into text.

What is needed is a way to convert the user’s utterance directly to a grammar, without going through the intermediate text representation.

You may know that speech grammars are statistical models of low-level phonemes matching phrases which the user is expected to say. Similar models can be created in response to a user’s utterance. The Voice Enrollment facility puts these pieces together and lets you create grammars based on user utterances.
To use voice-generated grammars, you first create a grammar in your application. When the application runs, you have the user create (or enroll) a phrase in the grammar. (The phrase is often a name, for example an address book entry.) At that time, the interpreter collects two or more utterances of each phrase from the user and uses those utterances to build a statistical model of the phrase. The interpreter associates a phrase id with each phrase; the phrase id identifies the phrase if it is recognized later when the grammar is activated.

You use the `<bevocal:enroll>` tag to enroll phrases in a grammar:

```xml
<bevocal:enroll name="enroll"
    grammarname="addresses"
    speakeridexpr="'1234'"
    phraseidexpr="'firstname'">
    <prompt count="1">
        Please say the name you want to add</prompt>
    <prompt count="2">Please say the name again</prompt>
</bevocal:enroll>
```

With this snippet, the system will prompt the user until it gets enough utterances to have a good statistical model of the phrase. If the model doesn’t converge, an error will eventually be thrown.

When you want to perform recognition with a grammar containing enrolled phrases, you refer to the grammar using a special syntax in the ABNF grammar format. (Currently, enrolled grammar access is supported only in ABNF; in a future release, the interpreter will support the XML grammar format.) For example:

```xml
<field>
    Who do you want to call?
    <grammar>
        <![CDATA[
            #ABNF 1.0 en-US;
            root $call;
            $call = call $<enrolled:/addresses?speaker=1234>
                [on [his | her] (cell | home) phone];
        ]]>  
    </grammar>
</field>
```

This ABNF grammar contains a reference to the enrolled grammar named “addresses”. In addition, it identifies the current speaker’s ID as 1234. Because enrolled phrases are speaker-dependent, the speaker ID is a required parameter. For more information, see “Recognizing Enrolled Grammars” on page 91.

### Usage Model

See the `<bevocal:enroll>` tag in the *VoiceXML Programmer’s Guide* for details of the syntax, attribute definitions, and definitions of shadow variables and exceptions for this tag.

The `<bevocal:enroll>` tag must collect several utterances until it has a consistent statistical model of the phrase the user is trying to enroll. In this it is unlike other VoiceXML input items, which only collect a single piece of input from the user. This could have been implemented as one atomic operation; that is, the tag would begin execution, collect as many utterances as it needs, and then return control to the Form.
Interpretation Algorithm. Instead, the implementation causes collection of each utterance to be done with a separate iteration of the FIA.

The first time a `<bevocal:enroll>` item is visited, it will collect one utterance and then return control to the FIA. However, unless that single utterance is sufficient for a consistent enrollment (the minimum value of `minconsistencies` is 2), the input variable will not be set. Therefore, the FIA's next iteration will select the same `<bevocal:enroll>` item, which will then collect a second utterance. This behavior will continue until a consistent enrollment is achieved, `maxtries` is reached, or an error occurs.

A major advantage of this approach is that it gives you very fine-grained control over the enrollment behavior. In particular, you can use the `<bevocal:enroll>` item’s prompt counter to supply tapered prompts for different iterations. For example:

```xml
<bevocal:enroll ... maxtries="5">
  <prompt count="1">
    Please say the name you want to add
  </prompt>
  <prompt count="2">
    Please say the name again
  </prompt>
  <prompt count="5">
    I'm having trouble understanding you.
    Please try one more time.
  </prompt>

  <catch event="error.enrollment.max_tries">
    ...
  </catch>
</bevocal:enroll>
```

You need to be aware that a `<bevocal:enroll>` item will typically be executed several times before it is successful. You will have to take this behavior into account if you want to manipulate the item’s variable yourself, or if you use the item’s `cond` attribute to control when it is executed.

Once enrollment succeeds, the input variable is filled with the audio from one of the user’s consistent utterances. Your application can send the audio to its back-end server using `<submit>` or `<data>` and store the audio in a database or file for later use in the user interface. Since there is no way to retrieve a text representation of an enrolled phrase, the audio recording is very useful for user interface purposes, for example in messages like "Now calling Jane Smyth".

When an enroll utterance clashes with an existing phrase in the enrolled grammar, an `error.enrollment.clash` event is thrown. You can use the `<bevocal:enroll>` tag’s shadow variables, `name$.clash` and `name$.clashedPhraseIds` to get information about the number of clashes and which phrase IDs the enroll utterance clashed with.

## Security Considerations

The `<bevocal.security.key>` property controls access to enrollment grammars. In this case, a security key can be thought of as a namespace that qualifies the `grammarname` attribute of the `<bevocal:enroll>` tag. Applications using one security key cannot access enrollment grammars created by an application using a second key, because their grammars live in separate namespaces.

When you develop applications for one of BeVocal's commercial hosting services such as Enterprise Hosting, you will need a security key in order to use enrollment.

When you develop on Café, you can use enrollment without a key; however there are limitations. First, there will be an implied key derived from your Café account number. This means that even if you use the same enrollment grammar name from two different Café accounts, you will not be able to access the same enrolled phrases. Even though the grammar names will appear to be the same, they will be two separate grammars in two separate namespaces. Second, when you are using enrollment in Café without a security key...
key, each grammar is limited to 10 enrolled phrases. Attempting to enroll more than 10 phrases will cause an error.

noauthorization event to be thrown.
Enrollment Example

Here is a more complex example of the `<bevocal:enroll>` tag:

```xml
<?xml version="1.0" ?>
<!DOCTYPE vxml PUBLIC "-//BeVocal Inc//VoiceXML 2.0//EN"
 "http://cafe.bevocal.com/libraries/dtd/vxml2-0-bevocal.dtd">
<vxml version="2.0" xmlns="http://www.w3.org/2001/vxml">
<form id="enroll_names">
  <block>
    Welcome to the address book demo.
    Let's add some names to the address book.
  </block>

  <!-- This event is thrown when there is a clash with one -->
  <!-- of the existing phrases in the enrolled grammar. -->
  <catch event="error.enrollment.clash">
    Oops! There was a clash for the enrollment sample.
    <exit/>
  </catch>

  <!-- This event is thrown when then minimum number of -->
  <!-- consistent utterances are not obtained within maxtries. -->
  <catch event="error.enrollment.max_tries">
    Maximum tries reached. Please try again.
    <exit/>
  </catch>

  <catch event="error.noauthorization">
    Maximum phrases enrolled.
    <exit/>
  </catch>

  <catch event="noinput">
    <prompt> In the noinput handler </prompt>
    <reprompt/>
  </catch>

  <!-- Prompts for a phrase to be enrolled. -->
  <!-- Executes this item at least twice to get 2 consistent -->
  <!-- samples for the phrase; that value is controlled by. -->
  <!-- minconsistencies. The grammarname and speakeridexpr uniquely -->
  <!-- identify an enrollment grammar. The phraseidexpr uniquely -->
  <!-- identifies a phrase in enrolled grammar and is returned -->
  <!-- when recognized against the enrollment grammar. -->
  <bevocal:enroll name="en1" 
              minconsistencies="2" maxtries="4"
              grammarname="ADDRESSBOOK" speakeridexpr="'speaker10'"
              phraseidexpr="'tom'" type="audio/wav">
    <prompt count="1"> Say a name </prompt>
    <prompt count="2"> Say the name again. </prompt>
    <prompt count="3"> Please say the name again. </prompt>
  </bevocal:enroll>
</form>
</vxml>
```
Recognizing Enrolled Grammars

Once you have enrolled phrases in a grammar, the next step is to perform speech recognition using that grammar. Currently, this is done by inserting a reference to the enrollment grammar in an ABNF grammar. For example:

```xml
<grammar>
  <![CDATA[
    #ABNF 1.0 en-US;
    root $call; $call = call $<enrolled:/addresses?speaker=1234>
      [on [his | her] (cell | home) phone]
  ]]> 
</grammar>
```

The syntax is:

```
$<enrolled:/grammarname?speaker=speakerid;key=securitykey>
```

The parameters are:

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>grammarname</code></td>
<td>The name of the enrollment grammar. This corresponds to the <code>grammarname</code> attribute of <code>&lt;bevocal:enroll&gt;</code>. <strong>Required.</strong></td>
</tr>
<tr>
<td><code>speaker</code></td>
<td>The ID of the current speaker. This must match a speaker ID that was previously used to enroll a phrase via the <code>&lt;bevocal:enroll&gt;</code> tag. <strong>Required.</strong></td>
</tr>
<tr>
<td><code>key</code></td>
<td>The security key for accessing this enrolled grammar. <strong>Optional</strong> when running on Café. <strong>Required</strong> if running in a non-Café environment such as Enterprise Hosting.</td>
</tr>
</tbody>
</table>

When a grammar that refers to an enrolled phrase is matched, a slot whose name is the same as the enrollment grammar name will be filled with the phrase ID of the phrase that was recognized. When the
As an alternative, you can use the grammar as a form grammar and perform the recognition using mixed initiative. This way, each slot returned by the recognition will be used to fill a field whose name (or slot attribute) matches a slot name in the grammar. This lets you use enrollment in complex grammars where you want to recognize not only an enrolled name but also other actions (for example, an action to perform, a modifier for the action, and so on).

Finally, note that in all other respects, grammars using enrollment behave just like any other grammar. Fields containing an enrollment grammar can also contain other grammars. When such a field is active, grammars in the enclosing form, document, and application are also active unless you have explicitly set the field's modal attribute to true. This gives you the flexibility to enable universal commands during your applications that use enrollment. In the address book example, in a single recognition state all of the following utterances might be valid:

<table>
<thead>
<tr>
<th>Utterance</th>
<th>Recognizing Grammar</th>
</tr>
</thead>
<tbody>
<tr>
<td>Call Jane Smith on her cell phone</td>
<td>Form-level mixed initiative grammar using enrollment</td>
</tr>
<tr>
<td>go back</td>
<td>Interpreter's predefined &quot;go back&quot; universal</td>
</tr>
<tr>
<td>go to address book</td>
<td>Link grammar in application root document</td>
</tr>
</tbody>
</table>

### Recognition Example

The following example performs recognition with the grammar that was defined in the example in “Enrollment Example” on page 90. Since the enrollment grammar is used inside a field and defines no other slots, the phrase ID of the recognized enrollment entry is used to fill the field f1 in the example.

```xml
<?xml version="1.0" ?>
<!DOCTYPE vxml PUBLIC "-//BeVocal Inc//VoiceXML 2.0//EN"
"http://cafe.bevocal.com/libraries/dtd/vxml-bevocal.dtd">

<vxml version="2.0" xmlns:bevocal="http://www.bevocal.com/">
  <link next="addressbook.vxml">
    <grammar>
      #ABNF 1.0;
      root $abook;
      $abook = address book;
    </grammar>
  </link>

  <form id="recognize_names">
    <field name="f1">
      Let's recognize the enrolled names.
      Say one of the enrolled names
      <grammar>
        <![CDATA[
          #ABNF 1.0;
          root $call;
          $call= [call] $<enrolled:/ADDRESSBOOK?speaker=speaker10>
            [on|at] [his|her|its|else] [home|work|cell] [phone];
        ]]>
      </grammar>
    </field>
  </form>
</vxml>
```
Deleting Enrolled Grammars

If you use enrollment to maintain a voice address book or other dynamic lookup mechanism, you need to be able to delete phrases from the grammar in addition to adding them. The BeVocal interpreter provides support for this via a JavaScript function that you can use to delete enrolled phrases.

`bevocal.enroll.removeEnrolledPhrase(grammer, speakerid, phraseid, key)`

**Parameters:**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>grammar</td>
<td>The name of the grammar containing the enrolled phrase to delete. <em>Required.</em></td>
</tr>
<tr>
<td>speakerid</td>
<td>The id of the speaker who enrolled the phrase to be deleted. <em>Required.</em></td>
</tr>
<tr>
<td>phraseid</td>
<td>The id of the phrase to be deleted. <em>Required.</em></td>
</tr>
<tr>
<td>key</td>
<td>The security key for accessing the specified enrollment grammar. <em>Optional</em> when running on the BeVocal Café. (You must pass a key argument, but it can be null or an empty string). <em>Required</em> when running in other environments such as Enterprise Hosting.</td>
</tr>
</tbody>
</table>