Grammar Reference

Version 2.2
March 2002

BeVocal, Inc.
1380 Bordeaux Drive
Sunnyvale, CA 94089

Table of Contents

Preface

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.

A VoiceXML application references a grammar with the `<grammar>` element. The grammar can be either:

- An inline grammar description (included directly in your VoiceXML code).
- An external grammar described in a separate file.

BeVocal VoiceXML supports grammars specified in the following forms:

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

This document describes GSL and XML grammars.

See http://www.w3.org/TR/speech-grammar/ for a description of the ABNF grammar syntax. See http://www.w3.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.

Note: BeVocal currently provides Beta-level support for JSGF grammars, with several limitations. These limitations will be removed in a future release. We do not yet support the `import` statement or semantic tags, and we do not currently handle inline grammar fragments. If you use inline JSGF grammars, you must use a complete, syntactically-correct JSGF grammar.

Audience

This document is for software developers using the BeVocal Café development environment. 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:

- VoiceXML Programmer's Guide. BeVocal.
  (http://cafe.bevocal.com/docs/vxml/index.html)
- VoiceXML Version 2.0 Specification. VoiceXML Forum.
  http://www.voicexml.org under Specs for PDF
  (http://www.w3.org/TR/voicexml20/ for HTML)
- Nuance Grammar Developer's Guide. Nuance.
  Under developers documentation after logging on at the Nuance web site (http://www.nuance.com).

1 Nuance Grammars

This chapter describes the Nuance Grammar Specification Language (GSL):

GSL Basics

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 field item variables. Grammars can range from a simple list of possible words to a complex set of possible phrases.

Grammar Rules

A grammar in GSL consists of one or more rules.

A grammar *rule* can have two parts:

1. A *rule name* (optional) that identifies the rule for use in other rules. A rule name must begin with a capital letter.
2. A *grammar expression* (required), which defines the possible utterances associated with that rule.

You create grammar expressions using a set of *grammar operators* to combine words and/or other expressions. To understand the examples in this section, you just need to know three of the grammar operators:

- `[]` – disjunction. For example, `[a b]` means a or b.
- `()` – sequence. For example, `(a b)` means a followed by b.
- ? – optional. For example, ?a means a is optional.

Since most grammars in VoiceXML identify a set of possible words that a user might say, the top-level grammar expression in a grammar rule is usually enclosed in square brackets `[ ]` to represent disjunction.

The following example shows a simple named grammar rule:

```
SummerMonths [june july august]
```

The rule name is `SummerMonths`, and the grammar expression is the disjunction `[june july august]`. This grammar is matched if the user says “june”, july”, or “august”.

Assignment Commands

Any grammar expression—either the top expression in the rule, or a component expression—can be followed by an *assignment commands* enclosed in curly braces `{ }`.

An assignment command specifies what values to assign to field item variables if the preceding grammar expression matches a user utterance. Assignment commands are not needed in every grammar. For example, you 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 assignment commands are present, they are ignored.

An assignment commands is associated with the grammar expression that immediately precedes it. 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.

Rules for grammars that appear within `<field>` or `<form>` elements generally have assignment commands of the form:

`<slotname value>`

A *slot name* identifies the field item variable to be assigned the specified value when a user utterance matches the grammar expression.

The following example shows a named grammar rule with assignment commands.

```
PrimaryColors [
  [red pink burgundy]         { <color red> }
  [(?sky blue) turquoise]     { <color blue> }
  [(?forest green chartreuse] { <color green> }
]
```

The grammar expression in this rule is a disjunction of three subexpressions. Each of the subexpressions is itself a disjunction and has an associated assignment command. If the user says "red", "prink", or "burgundy", this utterance matches the first subexpression and the assignment command `<color red>` is executed, assigning the value `red` to the field item variable `color`.

Subgrammars

Complex grammars can contain references to *subgrammars*. A subgrammar is a named grammar rule that is referenced (by name) from another rule. Subgrammars let you modularize your grammar descriptions. For example, this version of the `PrimaryColors` grammar rule refers to the subgrammar `Shades`.

```
PrimaryColors [
  ( ?Shades [red pink burgundy])         { <color red> }
  ( ?Shades [(?sky blue) turquoise])     { <color blue> }
  ( ?Shades [(?forest green chartreuse]) { <color green> }

Shades [
  dark
  light
]
```

When a grammar rule contains the name of a subgrammar, the effect is the same as if the referenced rule's grammar expression (and any assignment commands) appeared in place of the rule name. Thus, the preceding `PrimaryColors` grammar rule is equivalent to:

```
PrimaryColors [
  ( ?[dark light] [red pink burgundy])
      { <color red> }
  ( ?[dark light] [(?sky blue) turquoise])
      { <color blue> }
  ( ?[dark light] [(?forest green chartreuse])
      { <color green> }
]
```

A subgrammar can itself reference another subgrammar. Thus, you can create hierarchies of grammar rules.

Variables

A reference to a subgrammar can specify a variable to be set to the value returned from the subgrammar; the subgrammar can return a value with an assignment command of the form:

`return (`*`value`*`)`

You declare a variable by following the subgrammar name with a colon (`:`) and a variable name. You get the value of the variable by preceding the variable name with a dollar sign ($). For example:

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

City [
  atlanta { <return "Atlanta"> }
  chicago { <return "Chicago"> }
  dallas  { <return "Dallas"> }
]
```

Field Grammars

For grammars contained in `<field>` elements, only one field item variable can be set by the grammar. As a consequence, you can choose the slot name arbitrarily.

The following grammar rule identifies month names and sets the month field to the corresponding three-letter month abbreviation.

```
Month [
  january  {<month jan>}
  february {<month feb>}
  march    {<month mar>}
  …
]
```

Suppose this rule is in a field grammar and the user says "February". Then the field item variable is filled with `feb`, the value associated with the grammar expression for the utterance "february".

If the field is named `date` (that is, the field has the attribute `name="date"`), you can obtain the value of the field item variable in the field's `<filled>` element with:

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

If the field also has attribute `slot="foo"`, the `date` variable still evaluates to `feb`. A `<field>` element's `slot` attribute is ignored for field level grammars.

If the grammar expression is a disjunction of a set of simple words, you can use a shorthand notation for field grammars that omits explicit assignment commands, as shown in the following example. Any matching word will be assigned to a default slot name.

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

If a field grammar contains this grammar expression and the user says "February", then the field item variable will be assigned the value `february`.

Form Grammars

If you are writing a grammar for a mixed-initiative form, you may use multiple slot names, each identifying a field item variable to be filled. In this case you can name each field according to the corresponding slot name in the grammar.

Suppose the following rule is defined as the grammar for a mixed-initiative form. If the user says, "May I talk to a person?" and the form contains a field named `support`, then `<value expr="support">` will evaluate to `true`.

```
Help [
  (i don't understand)
    {<help "true">}
  [what huh help]
    {<help "true">}
  (?(talk [to with])[someone (a ?live person)])
    {<support "true">}
  [(?good bye) asta_la_vista]
    {<exit "true">}
]
```

If the field names aren't the same as the grammar's slot names, you can use the `slot` attribute on the `<field>` elements to map the grammar slot values to the appropriate field item variables. To show this, suppose a mixed initiative form contains the preceding grammar and a field named `x` with the attribute `slot="support"`. If the users says, "May I talk to a person?" then `<value expr="x">` will evaluate to `true`.

Note that only the field's `name` attribute represents a variable; the `slot` attribute is simply a mapping from the grammar assignment command to a particular field. As a result, in this case, `<value expr="support">` evaluates to `undefined`.

Multiple fields can have the same slot name, that is, they can all have the same value for their `slot` attributes. If a form grammar sets a value for a particular slot, *all* fields with that slot name are filled with the value.

Ambiguous Grammars

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

```
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 the chapter on using multiple recognition in the VoiceXML Programmer’s Guide (http://cafe.bevocal.com/docs/vxml/multiresults.html).

Referencing Grammars

A VoiceXML application references a grammar with the `<grammar>` element. The grammar can be either:

- An inline grammar description (included directly in your VoiceXML code).
- An external grammar described in a separate file.

Inline Grammars

To include a grammar description directly in your VoiceXML code, embed it inside the `<grammar>` tag. You can use CDATA escapes when you want to include special characters not normally permitted in XML documents, such as angle brackets (< and >).

This example shows an inline form grammar that consists of a single unnamed rule:

```
<form>
  <grammar>
    <![CDATA[
      [
        (atlanta ?georgia)  { <city Atlanta> <state Georgia> }
        (chicago ?illinois) { <city Chicago> <state Illinois> }
        (dallas ?texas)     { <city Dallas> <state Texas> }
      ]
    ]]>
  </grammar>
  ...
</form>
```

The grammar must consist of one of the following:

- A single rule, which need not be named.
- Multiple named rules; the first rule is assumed to be the root rule of the grammar.

Source Grammar Files

A source grammar file is a text file that contains one or more named rules. For example, the file `colors.grammar` contains the following three rules:

```
ShadeAndColor (Shades Colors)

Colors [
  red  {<color red>}
  blue {<color blue>}
]

Shades [
  dark  {<shade dark>}
  light {<shade light>}
]
```

Note: The `<grammar>` tags and CDATA escapes must not be included in the grammar description given in the external file.

URI of Grammar File

To reference a grammar description that is contained in an external file, you set the `src` attribute of the `<grammar>` element to the URI of the grammar file. 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

  `http://myCompany.com/vxml/mygram.grammar`
- Relative to the host:

  `/vxml/mygram.grammar`

  Note the initial forward slash.
- Relative to the location of the VoiceXML document:

  `vxml/mygram.grammar`

  Note the lack of the initial forward slash.

In addition, you can use an at sign (`@`) in the URI to refer to the BeVocal hosting platform. For example:

```
@/bevocal/grammars/numbers.grammar
```

The `src` attribute must consists of the URI for the external grammar file followed a pound sign (#) and the name of the root rule of the grammar. The grammar consists of the root rule and all subgrammars referenced directly or indirectly from it.

For example, the `color` field uses the `Colors` rule as the root (and only) rule of its grammar:

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

The `paint` field uses the `ShadeAndColor` rule as the root rule of its grammar, which also contains the subgrammars `Shades` and `Colors`.

```
<field name="paint">
  <grammar src="colors.grammar#ShadeAndColor"/>
  ...
</field>
```

Compiled Grammar Files

The first time you reference a particular source grammar file, that grammar file is compiled and the compiled file is cached. Subsequent references to the same source grammar file actually use the compiled file instead. 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 takes 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 time-out error. To avoid this problem, you can:

1. Compile the grammar file before running the application.
2. Refer to the compiled grammar file (instead of the source grammar file) from your application.

Compiling a Grammar File

You use the Grammar Compiler tool to submit a request for offline compilation of a GSL grammar file. 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 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.

For performance reasons, you should compile a grammar file that is larger than 100 kilobytes. Café customers *must* compile a grammar file that is larger than 150 kilobytes, because their applications are not allowed to reference that large a source grammar file. (This size restriction does not apply to hosting customers.)

Typically, you develop and test your application to reference a small subset of your grammar in an 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.

Referencing a Compiled Grammar File

To reference the grammar in a compiled grammar file, 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.

Grammar Scopes

VoiceXML grammars are scoped. By default, the scope of a grammar is set by the element that contains the grammar. There are four basic scopes.

Scope	Grammar is in scope when execution is:	Default scope for a grammar defined in:
Field	In the <field> where it is defined or referenced.	A <field> element.
Dialog	In the dialog where it is defined or referenced.	A <form> element. A <choice> element.
Document	In the document where it is defined or referenced.	A <link> element directly under a <vxml> element.
Application	Anywhere in the application.	Any element with document scope that appears in the application root document.

A grammar is *active* when it is in scope. The speech-recognition engine will recognize utterances from any and all active grammars. 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.

In general, we recommend that you keep the scope of your grammars as narrow as possible, which allows the speech-recognition engine to be most efficient.

Grammar Syntax

GSL Grammar descriptions are composed of three components: *rule names*, *grammar expressions*, and *assignment commands*. This chapter gives the syntax for each of these components.

Rule Names

A rule name identifies the following grammar expression. Every *rule name* must start with an upper-case letter. You can use the following characters within rule names:

- upper case or lower case letters (but the rule name cannot be all uppercase letters)
- digits
- special characters, limited to:

  - (hyphen), _ (underscore), ' (single quote), @ (at sign), . (period)

Grammar Expressions

Grammar expressions consist of word tokens, rule references, and grammar operators. *Word tokens* correspond to the actual words a user might speak. *Rule references* correspond to subgrammars.

Word tokens and rule references must be separated from each other by whitespace. Whitespace is optional between the grammar operators and their operands (word tokens or rule names). You can add comments using the semicolon (`;`) character; everything on the line after a semicolon is ignored.

Word Tokens

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

- lower case letters *only*
- digits
- special characters, limited to:

  `-` (hyphen), `_` (underscore), `'` (single quote), `@` (at sign), `.` (period)

If you enclose a word 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 word token, but `"new york"` is not.

Rule References

A rule reference has one of the following two forms:

- *`ruleName`*
- *`ruleName`*`:`*`variableName`*

Both forms reference the rule named `ruleName` as a subgrammar. The second form additionally specifies a variable named *`variableName`* to be set to any value returned from the subgrammar.

A grammar expression can reference any grammar rule defined in the grammar that contains the grammar expression.

Grammar Operators

Word tokens and rule references are combined into grammar expressions by *grammar operators*. In the table below, a, b, c, and d represent word tokens, rule references, or more complex grammar expressions.

Grammar Operator	Example Expression	Meaning
( ) sequence	`(a b c … d)`	a and b and c and … and d (in that order)
[ ] disjunction	`[a b c … d]`	One of a or b or c or … or d
? optional	`?a`	a is optional
+ positive closure	`+a`	One or more repetitions of a
* kleene closure	`*a`	Zero or more repetitions of a

The following table list some example grammar expressions and gives the set of possible spoken phrases that each expression represents.

Grammar expression	Matching utterances
`[large regular]`	• “large” • “regular”
`([large regular] fries)`	• “large fries” • “regular fries”
`([large regular] (?french fries))`	• “large French fries” • “regular French fries” • “large fries” • “regular fries”
`(it’s +very good)`	• “it’s very good” • “it’s very very good” • “it’s very very very good” • …
`(it’s *very good)`	• “it’s good” • “it’s very good” • “it’s very very good” • “it’s very very very good” • …

Transition Weights

You can assign *transition weights*, or probabilities, to most grammar expressions using the ~ operator. You specify a probability for a grammar expression as follows:

`grammarExpression~probability`

The probability must be a non-negative number. The following expression means that a has a 40% probability of occurring:

```
a~.4
```

Probabilities are mostly used in disjunct (OR) constructs. For example:

```
Softdrinks [
  coke~.5
  sprite~.3
  orange~.2
]
```

All probabilities in the disjunct are normalized to add up to 1.0. If you don’t specify any probabilities, all expressions in the disjunct are considered to be equally likely.

DTMF Tokens

You can express valid touchtone sequences in a grammar expression using the following DTMF notation.

Key Press	DTMF Notation
0	`dtmf-0`
1	`dtmf-1`
2	`dtmf-2`
3	`dtmf-3`
4	`dtmf-4`
5	`dtmf-5`
6	`dtmf-6`
7	`dtmf-7`
8	`dtmf-8`
9	`dtmf-9`
*	`dtmf-star`
#	`dtmf-pound`

The following grammar expression 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 assign a value of `john_smith` to a field item variable.

Grammar Assignment Commands

You use grammar *assignment commands* to associate an assignment operation to a particular grammar expression, either the top expression in the rule, or a component expression. The assignment commands for a grammar expression are enclosed in curly braces `{ }` and immediately follow the associated grammar expression. The curly braces can enclose one or more individual assignment commands.

Command Types

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

- A *slot-filling command* specifies a value for a field item variable; it has the form:

  `<slotname value>`

  The slot name indicates the field item variable to be assigned a value.

- A *return command* returns a value from a subgrammar; it has the form:

  `return (`*`value`*`)`

  A return command is useful when the rule reference to the subgrammar specifies a variable. When the subgrammar 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 subgrammar. If the rule is ever used as a root grammar rule it must not include a `return` command.

Value Expressions

The *`value`* expression in an assignment command can be a literal value, a variable expression, or a function call.

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 strings 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, for example:

```
DigitString [ …
  ([zero oh] one) {<return("01")>}
  ...
]
```

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

Variable Expression

If a grammar includes a rule reference to a subgrammar and that rule reference specifies a variable *`variableName`*, an assignment command of the containing grammar 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 grammar expression; its value can be obtained with expression:

```
$string
```

For example, the following grammar accepts the name of a day of the week and sets the `day` field item variable 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 functions 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.

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

The following example illustrates how the functions can be used to compute the value of the `number` field item variable from values returned by subgrammars.

```
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 grammar expression has an optional element. If the user said “twenty”, the value of `n2` would be undefined. Since the default value for the `add` function is zero, the result will be correct (20 + 0).

2 XML Speech Grammars

This chapter describes the XML Speech Recognition Grammar Format:

XML Grammar Basics

An XML grammar, like XML 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; and 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 container 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 `"/>"`.

Rules

An XML grammar consists of rules. Each rule specifies some user input that can be recognized and, optionally, specifies how to interpret a valid expression in terms of values for field item variables.

The `<rule>` tag defines a rule in an XML grammar. Each rule definition has a name, specified by the `id` attribute. A rule's name must be unique within the scope of the grammar that contains the rule. A rule name names must be a legal XML ID.

A rule's `scope` attribute indicates where it can be used. The scope may be either:

- `"private"`—local to the grammar that contains it.
- `"public"`—available to be referenced from other grammars.

A rule with no `scope` attribute is private by default.

Rules specify user input with tokens are rule references.

- A *token* corresponds to a word that a user might actually speak. 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 contain in a `<token>` element.
- A *rule reference* refer to another grammar rule specifying user input. The `<ruleref>` tag references another rule.

Inline and External Grammars

An inline XML grammar consists of one or more `<rule>` elements inside the VoiceXML `<grammar>` tag.

An external XML grammar is an XML file with a `<grammar>` element containing `<rule>` elements. Like any XML file, an external XML grammar file begins with the header:

```
<?xml version="1.0" ?>
```

Note that the version number refers to the version of XML, not the version of the grammar it contains.

Default Rule

Every XML grammar has a *default rule*. If the `<grammar>` element containing the rule has a `root` attribute, that attribute names the default rule. Otherwise, the default rule is constructed implicitly as the logical OR (disjunction) of all public rules in the grammar.

Language and Pronunciation

The `xml:lang` attribute of a `<grammar>` element specifies its language of the spoken input. This alerts the speech-recognition engine should use pronunciation rules, phonetic inventory and acoustic models for the specified language. Currently, English is the only supported language.

Various components within a grammar can have a `lang-list` attribute, which specifies one or more languages in which the spoken input is expected. A single language is analogous to the language for the grammar. If multiple languages are specified, the speech-recognition engine should use the pronunciation, phonetic inventory and acoustic models of the different languages in parallel.

The language is specified with an identifier that designates the language and, optionally, the country whose local pronunciation and vocabulary should be used. For example, the identifier `en-US` designated United States English.

Tag Summary

The following table classifies the tags in the XML form of the W3C Speech Recognition Grammar Format according to their purpose.

Purpose	Tags
Grammar definition	<grammar>
Rule definition	<rule>
Rule reference	<ruleref>
Input elements	<token> <one-of> <item> <tag>
Example	<example>

Tag Index

The following table lists the tags used in the XML form of the W3C Speech Recognition Grammar Format. The <grammar> tag described here is used only in and external XML grammar file. Other tags are used both in external grammar files and in VoiceXML documents in inline grammars.

Tag	Description
<example>	Example phrase that matches the containing grammar rule.
<grammar>	Defines the grammar in an external XML grammar file.
<item>	Input element that indicates optional or repeated user input.
<one-of>	Input element that indicates alternative user inputs.
<rule>	Defines a grammar rule.
<ruleref>	Input element that references another rule.
<tag>	Specifies how to interpret the user input.
<token>	Input element that specifies words to be spoken by the user.

Tag Descriptions

The remainder of this chapter contains tag descriptions is alphabetical order.

<example>

Example phrase that matches the containing grammar rule.

Syntax

```
<example>
  Example Input
</example>
```

Description

An <example> element encloses a sequence of tokens corresponding to user input that matches the containing rule. It is illustrative, for the benefit of a developer reading the grammar; the speech-recognition engine ignores the element.

Usage

Parents	Children
<rule>	None.

<grammar>

Defines the grammar in an external XML grammar file.

Syntax

```
<grammar
    version="1.0"
    xml:lang="language"
    mode="voice"|"dtmf"
    root="string" >
  Rules
</grammar>
```

Description

Top-level element in each XML grammar file.

Attribute	Description
`version`	XML Speech Recognition Grammar Format version used in this document. *Optional* (default is `"1.0"`). The only accepted value is `"1.0"`.
`xml:lang`	The language identifier for the grammar; see "Language and Pronunciation" on page 22. *Optional* (default is `"en-US"`) The accepted language identifiers are: • `en`—English • `en-US`—United States English
`mode`	The mode of the contained or referenced grammar. *Optional* (default is `"voice"`). • `voice`—Spoken input • `dtmf`—DTMF input.
`root`	The name of the explicit default grammar rule. *Optional* (if omitted, an implicit default rule is used; see "Default Rule" on page 22).

Usage

Parents	Children
None	`<rule>`

<item>

Input element that indicates optional or repeated user input.

Syntax

```
<item
    repeat="M"|"M-N"|"M-"|"0-1" >
  Content
</item>
```

Description

An `<item>` element can contain any number of input elements and tag elements. The input elements indicate a sequence that must be matched in order. The `repeat` attribute applies to the entire sequence, indicating that it is optional or that it may be repeated.

Any contained `<tag>` elements apply to the entire optional or repeated sequence. If the user input matches the input elements and `repeat` attribute, the `<tag>` elements are interpreted to assign values to field item variables.

Attribute	Description
`repeat`	Indicates the number of times that the contained expansion may be repeated. *Optional* (if omitted, the sequence of input elements must occur exactly once). • *N* The contained expansion is repeated exactly "n" times. "n" must be "0" or a positive integer. • *M-N* The contained expansion is repeated "m" times or more (inclusive). "m" must be "0" or a positive integer. For example, "3-" declares that the contained expansion can occur three, four, five or more times. • *M-* The contained expansion is repeated "m" times or more (inclusive). "m" must be "0" or a positive integer. For example, "3-" declares that the contained expansion can occur three, four, five or more times • *0-1* The contained expansion is optional.
`repeat-prob`	The probability of the optional repetitions specified in the `repeat` attribute. This attribute is ignored if the `repeat` attribute is not specified. *Optional*.

Usage

Parents	Children

`<rule>`
`<item>`
	`<token>`
	`<ruleref>`
	`<item>`
	`<one-of>`
	`<tag>`

<one-of>

Input element that indicates alternative user inputs.

Syntax

```
<one-of
    lang-list="languages" >
  Alternatives
</one-of>
```

Description

The contained items are alternatives; any one of them may be matched by the user input. Each alternative is an <item> element.

Attribute	Description
lang-list	A comma-separated list of the identifiers for languages in which the alternative inputs may be spoken; see "Language and Pronunciation" on page 22. *Optional* (default is "en-US") The accepted language identifiers are: • en—English • en-US—United States English

Usage

Parents	Children
<rule> <item>	<item>

<rule>

Defines a grammar rule.

Syntax

```
<rule
    id="string"
    scope="private"|"public" >
  Other Content
</rule>
```

Description

An <rule> element can contain any number of input elements, examples, and tag elements. The input elements indicate a sequence that must be matched in order.

Any contained <tag> elements apply to the entire sequence. If the user input matches the input elements, the <tag> elements are interpreted to assign values to field item variables.

Any <example> elements are ignored by the speech-recognition engine.

Attribute	Description
id	The name of the rule; must be unique within the containing grammar (this is enforced by XML). The rule must not be the name of one of the special rules "NULL", "VOID", or "GARBAGE".
scope	The scope in which this rule can be used. *Optional* (default is "private"). • private—The rule can be used only by the containing grammar • public—The rule can be reference by another grammar (in a <ruleref> element in the referencing grammar. **Note:** Do not confuse the scope of a rule with the scope of a containing grammar. The scope of the grammar indicates where in the VoiceXML application the grammar is active.

Usage

Parents	Children
<grammar>	<token> <ruleref> <item> <one-of> <tag> <example>

<ruleref>

Input element that references another rule.

Syntax

```
<ruleref
    lang-list="languages"
    special="NULL"|"VOID"|"GARBAGE"
    uri="URI" >
  Optional Tags
</ruleref>
```

Description

The referenced rule specifies user input to be matched.

If the user input matches the referenced rule, any contained <tag> elements are interpreted to assign values to field item variables.

Attribute	Description
lang-list	The language and optional country local identifier for the referenced rule. *Optional* (default is "en-US") The accepted language identifiers are: • en—English • en-US—United States English
special	The referenced special rule. *Optional* (as an alternative to uri). • NULL—Rule that is automatically matched, that is, matched without the user speaking any word. • VOID—Rule that can never be spoken. Inserting VOID into a sequence automatically makes that sequence unspeakable • GARBAGE—Rule rule that matches any speech up until the next rule match, the next token or until the end of spoken input.
uri	The URI of the referenced rule. *Optional* (as an alternative to special). May be one of the following: • *#ruleName*—References the local rule *ruleName* in the contained grammar. • *grammarFileURI#ruleName*—References the public rule *ruleName* in the grammar defined in the XML grammar file whose URI is *grammarFileURI*. • *grammarFileURI*—References the default rule of the grammar defined in the XML grammar file whose URI is *grammarFileURI*. See "Default Rule" on page 22.

Usage

Parents	Children
<rule> <item>	<tag>

<tag>

Specifies how to interpret the user input.

Syntax

```
<tag>
  slotName="value"
</tag>
```

Description

Within a <tag> element, *slotName* identifies the field item variable to be assigned the specified *value* when a user utterance matches the input specified by the containing element.

Usage

Parents	Children
<rule> <item>	None.

<token>

Input element that specifies words to be spoken by the user.

Syntax

```
<token
    lang-list="languages">
  Content
</token>
```

Description

If the user input matches the contained words, any contained <tag> elements are interpreted to assign values to field item variables.

Attribute	Description
lang-list	A comma-separated list of the identifiers for languages in which the user input may be spoken; see "Language and Pronunciation" on page 22. *Optional* (default is "en-US") The accepted language identifiers are: • en—English • en-US—United States English

Usage

Parents	Children
<rule> <item>	<tag>