SPEECH UNIVERSITY LEARNING MODULE GUIDE

XML GRAMMARS

Notes:

Date	Version	Summary of Changes	Author
9/27/04	1.1.0	Based on OS Basic XML Grammars Overview, which was based on previous version of XML Grammar SLM	Marie McCarthy
10/21/04	1.1.1	Added PPTs and INs; added updated demos; made a few corrections	Marie McCarthy
11/21/2005	1.1.2	Rebranded for Nuance	DHarper

Notice

Published by:

Nuance Communications, Inc.
1 Wayside Road
Burlington, MA 01803, U.S.A.

MODULE OBJECTIVES

Notes:

Upon successful completion of this module, you should be able to:

- Recognize the rules and rule references you use to build XML grammars
- Build basic XML grammars
- Utilize key/value pairs to streamline semantic interpretation
- Use SpeechWorks OpenSpeech Recognizer Tools to test XML grammars

Notes:

INTRODUCTION

Notes:

A grammar is a text file that specifies what the speech recognition engine can understand. Grammars consist of vocabulary words and the rules that govern the patterns in which those words can be spoken. Grammars are compiled into machine readable code that a recognizer uses to perform the recognition.

This module will describe the syntax, semantics and use of the **SRGS XML grammar form,** the standard for speech recognition engines. We will demonstrate tools for testing and compiling grammars. We will also explain how to add semantic interpretation to grammars.

For purposes of this course, we will focus on the OpenSpeech Recognizer, though since the syntax is largely standards based, the information may be easily redirected to the Nuance 8.x recognizer.

The XML grammar form is one of two grammar forms described in the **Speech Recognition Grammar Specification (SRGS)** published by the **World Wide Web Consortium (W3C).** The XML grammar form has received industry-wide acceptance and its support is required by the VoiceXML 2.0 SRGS specification published by the W3C. When reading this module, assume that all information and examples are common to all three versions of the SRGS. Exceptions are clearly labeled. As the 6/02 SRGS candidate recommendation and the 12/03 SRGS proposed recommendation are essentially identical, and the 12/03 SRGS proposed recommendation is fully compatible with the 7/02 SRGS candidate recommendation, the abbreviation SRGS CR&PR will be used to label examples that are common to both versions.To obtain a copy of the SRGS, visit the World Wide Web Consortium’s web site at www.w3.org.

Notes:

What is a grammar?

A grammar is created as a text file that specifies what the speech recognition engine can understand. It consists of vocabulary words and patterns in which those words occur. Think of a grammar as words and rules that describe those words. The grammar includes synonyms and optional words that callers may say. Distinct grammars are needed for almost every collection context in an application.

A grammar has several functions.

- By setting up rules that govern the order and frequency of vocabulary words, it constrains the sentences the recognizer considers. This example lets the caller set the destination for routing their phone calls:

```
<rule id="Home">
      <count number="optional"> to my </count>
      home
     <count number="optional"> phone </count>
 </rule>
```

- Because an application usually needs to know the meaning of what was said, rather than the raw text that was spoken, a good grammar returns a **semantic meaning** to the application as a set of key/value pairs.

```
<rule id="Home" tag="DESTINATION='home'">
  <count number="optional"> to my </count>
  home
  <count number="optional"> phone </count>
</rule>
```

- A grammar may also filter out some sentences before they reach the n-best list. For example, a grammar that allows a caller to say "from <source phone> to <destination phone>" might exclude the sentence "from my home phone to my home phone"

In other words, a grammar informs the speech recognizer to

- Listen for **words** that may be spoken
- Listen for **patterns** in which those words occur
- Return a **semantic meaning** to the application

Notes:

OSR processes the grammar and the speech input as follows:

Inputs to OSR	o Grammar(s): text file with the words and patterns of words to listen for o Audio stream: the speech input from the caller.
OSR Processing	o Compiles the grammar (if necessary) o Performs "best-match" between audio and grammar
Outputs from OSR	o Recognition result: N-best list of hypotheses (results) with confidence score, semantic meaning and raw utterance o Parsing errors and other information. Examples: input mode={dtmf or voice}; no input; no match.

Speech recognition is possible only when Caller input matches words or phrases identified in the grammar. A good grammar is effective in several ways:

- It provides for *all* reasonable Caller utterances
- It does not over-generate phrase options or include non-grammatical phrases
- It passes concise semantic meanings to the application
- It limits grammar/application dependencies
- It avoids inefficient use of memory and CPU
- It avoids synonymous entries from the n-best list

Notes:

eXtensible Markup Language

The **XML grammar form** is based on Extensible Markup Language, or XML. A brief introduction to XML makes the XML grammar form easier to understand and use.

XML is a markup language for documents containing structured information.

In general, structured information contains both content (words, pictures, etc.) and some indication of what role that content plays (for example, section heading content is different from footnote content or the content in a database table, etc.).

A markup language is a mechanism to identify structures in a document. The **XML specification** defines a standard way to add markup to documents.

```
<?xml version="1.0" encoding-"iso-8559-1"?>
<!-- database record -->
<customer_record>
   <last_name> Smith </last_name>
   <first_name> Josie </last_name>
   <mail_address line="1"> Josie Smith </mail_address>
   <mail_address line="2"> PO box 1234 </mail_address>
   <mail_address line="3"> Boston, MA </mail_address>
   <mail_address line="4"> 01201 </mail_address>
  <phone location="cell" preferred="false">
         617-428-4444
    </phone>
  <phone location="home" preferred="true">
         617-424-1234
   </phone>
</customer_record>
```

XML Structure

XML elements defines the structure of an XML document. The content may be text, more XML elements, or both.

- *Elements* (also called tags) are used to mark up the text
  `<last_name>` `Smith` **`</last_name>`**
- *Initial elements* - the XML document must start with the xml header:
  `<?xml version="1.0" encoding-"iso-8559-1"?>`
- *Attributes* annotate the tag element:
  `<mail_address` **`line="1"`**`> Josie Smith </mail_address>`

Notes:

- *Content* of an element is the code between the opening and closing tags of the element.
  ```
  <last_name> Smith </last_name>
  ```
- Comments:
  ```
  <!-- database record -->
  ```

Notes:

Examples of XML Code

The following four examples illustrate important elements of XML: the role of the header, the element tree of tags, the relationship of text to code, and the use of attributes.

Initial Elements

Every XML document requires an **XML version** declaration. This must begin at the first character of the first line of the file. The encoding attribute is optional.

```
<?xml version="1.0" encoding="iso-8559-1"?>
<!-- this is SSML -->
   <speak>
     <paragraph>
       <sentence>The first message is from
          <say-as type="name">Stephanie</say-as>
          and arrived at
          <break size="medium"/>
          <say-as type="time">3:45pm</say-as>
       </sentence>
     </paragraph>
   </speak>
```

Element Tree

XML documents follow a tree-like hierarchical structure that incorporates inheritance considerations. The XML-specific language specification dictates the tree-level at which each of its element may occur.

Level 1	Initial elements `<speak> </speak>`
Level 2	`<paragraph> </paragraph>`
Level 3	`<sentence> </sentence>`

Level 4	`<say-as> </say-as>`
	`<break/>`

In the example above,

- `<paragraph>` is called a ‘child’ of `<speak>`
- `<paragraph>` is called a ‘parent’ of `<sentence>`
- `<say-as>` and `<break/>` are called ‘siblings’

Notes:

Text

Parsed Character Data is the raw text that occurs as content of XML elements. It designates XML-language specific information.

In this case the raw-text represents words to be "spoken" using text-to-speech:

```
<?xml version="1.0" encoding="iso-8559-1"?>
<!-- this is SSML -->
  <speak>
    <paragraph>
      <sentence> The first message is from
        <say-as type="name">Stephanie</say-as>
          and arrived at
          <break size="medium"/>
        <say-as type="time">3:45pm</say-as>
      </sentence>
    </paragraph>
  </speak>
```

In an XML grammar form, the raw text represents words to be recognized by the speech recognition engine.

```
<!-- this is an XML grammar fragment -->
<rule id="Home">
   <count number="optional"> to my </count>
   home
   <count number="optional"> phone </count>
</rule>
```

Attributes

Attributes modify elements. The attribute tags in the above example are placed after the name of the tag:

```
<say-as type="name">

<break size="medium"/>
```

Notes:

XML Syntax Concepts

XML has many specific syntax rules. Following are some of the most important XML rules that are necessary in the XML grammar form.

- Quote marks are required around attribute values:

  Incorrect: `<cost currency=euro>26.54</cost>`

  Correct: `<cost currency="euro">26.54</cost>`

- Closing elements must be noted:

  Incorrect: `<cost currency="euro">26.54`

  Correct: `<cost currency="euro">26.54</cost>`

- There is a shorthand for empty elements:

  `<tag>... </tag> == <tag/>`

- Exact nesting is required (child tags must close before their parent tags close):

  Incorrect: `<a><b></a></b>`

  Correct: `<a><b></b></a>`

- Certain symbols that resemble those identifying pieces of code require that you "escape." Use an ampersand to distinguish them from symbols interpreted as code and with a semicolon. For example:

Symbol	Meaning	Escape - write as
>	greater than	**&gt;**
>=	greater than or equal to	**&gt;=**
<	less than	**&lt;**
<=	less than or equal to	**&lt;=**
"	quotation mark	**&quot;**
'	apostrophe	**&apos;**
&	ampersand	**&amp;**
&&	ampersand ampersand	**&amp;&amp;**

XML Language Concepts

Notes:

- Well-formed XML documents follow rules for XML syntax and style.
- Valid documents must follow a Document Type Definition (DTD) or a schema
  - a DTD defines legal elements, content and attributes
  - a schema is an alternative to a DTD, and defines the legal elements, content, and attributes of an XML language using XML

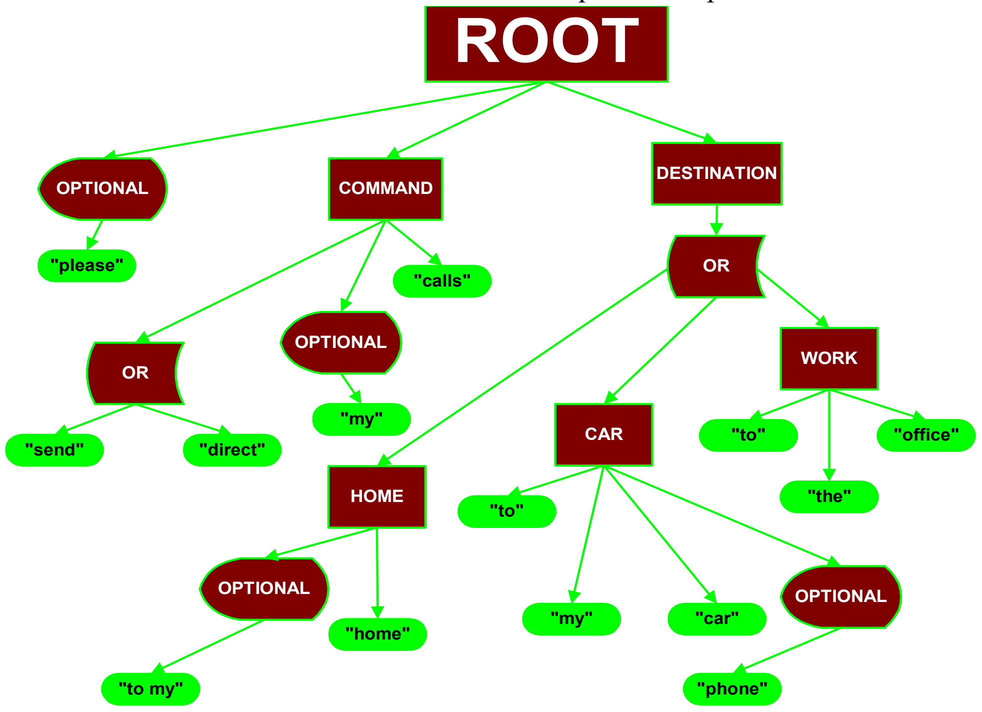
The XML grammar specification provides an informative DTD and a normative schema.

Notes:

Grammar Tree

In general, a grammar can be represented as an inverted tree composed of a **root**, **terminals** and **nonterminals**.

- The *root node* is at the top and all other nodes expand from it.
- *Nonterminals* are like branches because they can be expanded into other values.
- *Terminals* are leaves because no further expansion is permitted.



In XML form grammars:

- The *root node* is the **root rule** that will be 'activated' by the recognizer during speech recognition. It consists of all intermediate rule references, operators and sequences of words that can be recognized.
- *Nonterminals* represent all other **rules** in the grammar. These allow for modularizing and reusing code, as well as for enhanced readability.
- *Terminals* represent the **words** the caller can say

Sample XML Form Grammar - 1/01 SRGS

Notes:

This tree may be represented by the following grammar which conforms to the 1/01 SRGS working draft:

```
<?xml version="1.0" ?>
<grammar xml:lang="en-US" version="1.0" root="ROOT">

  <rule id="ROOT" scope="public">
    <count number="optional"> please </count>
    <ruleref uri="#Command"/>
    <ruleref uri="#Destination" />
  </rule>

  <rule id="Command">
    <one-of>
      <item> send </item>
      <item> direct </item>
    </one-of>
    <count number="optional"> my </count>
      calls
  </rule>

  <rule id="Destination">
    <one-of>
      <item> <ruleref uri="#Home"/> </item>
      <item> <ruleref uri="#Car"/> </item>
      <item> <ruleref uri="#Work"/> </item>
    </one-of>
  </rule>

  <rule id="Home">
    <count number="optional"> to my </count>
      home
    <count number="optional"> phone </count>
  </rule>

  <rule id="Car">
    to my car
    <count number="optional"> phone </count>
  </rule>

  <rule id="Work">
    to the office
  </rule>

</grammar>
```

Notes:

Sample XML Form Grammar SRGS CR&PR

This tree may be represented by the following grammar which conforms to the SRGS CR&PR candidate recommendation:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<grammar xml:lang="en-US" version="1.0" root="ROOT"
  xmlns="http://www.w3.org/2001/06/grammar"
  tag-format="swi-semantics/1.0">
  <rule id="ROOT" scope="public">
    <item repeat="0-1"> please </item>
    <ruleref uri="#Command"/>
    <ruleref uri="#Destination" />
  </rule>
  <rule id="Command">
    <one-of>
      <item>direct</item>
      <item>send</item>
    </one-of>
    <item repeat="0-1"> my </item>
    calls
  </rule>
  <rule id="Destination">
    <one-of>
      <item> <ruleref uri="#Home" />
      </item>
      <item> <ruleref uri="#Car" />
      </item>
      <item> <ruleref uri="#Work" />
      </item>
    </one-of>
  </rule>
  <rule id="Home">
    <item repeat="0-1"> to my </item>
    home
  </rule>
  <rule id="Car">
    to my car
    <item repeat="0-1"> phone </item>
  </rule>
  <rule id="Work"> to the office </rule>
</grammar>
```

Composing XML Grammar Documents

Notes:

Three elements provide essential scaffolding in any XML grammar document:

- Header and Character Encoding
- Root Rule Declaration
- Comments

<xml> Heading and Character Encoding

Every XML grammar begins with an XML declaration and the XML version="1.0" attribute. XML form grammars should include character encoding to indicate the set of symbols used in the document. If omitted, the recognizer or platform will determine encoding.

```
<?xml version="1.0" encoding="ISO8859-5"?>
```

<grammar> and Root Rule Declaration

Following the XML declaration, is the `<grammar>` element and its attributes. There is one `<grammar>` element per grammar file. The **content** of this root grammar element is the set of rules defining what the caller may say.

```
<?xml version="1.0" encoding="ISO8859-5"?>
<grammar xml:lang="en-US" version="1.0" xmlns="http://
www.w3.org/2001/06/grammar" root="ROOT">
 <rule id="ROOT" scope="public">
 ...
 </rule>
...
</grammar>
```

The grammar specification allows and optional root attribute of the <grammar> element. An XML grammar is not required to declare a root rule, but it is good practice to do so. A root declaration must identify one public rule defined elsewhere in the same grammar.

The root rule is used to pass information up to the recognizer. use of the root rule ensures an explicit path to what is happening during recognition.

Notes:

Some important attributes of grammar include:

Attribute	Description

`xml:lang` (recommended)	Language models used for recognition. Default is US English: `en-US`
	Examples:
	`xmlg="en-US"`
	`xml:lang="fr-CA"`

`version` (required)	XML form grammar version. Current `version="1.0"`
`root` (optional)	Identifies a rule that is at the root of the grammar tree
`xmlns` (required)	XML namespace provides a method for qualifying elements and attribute names used in different XML-based documents by associating them with namespaces `xmlns="http://www.w3.org/2001/06/grammar"`

Comments

XML comments may be placed almost anywhere in a grammar document. The syntax is as follows:

```
<?xml version="1.0" encoding="ISO8859-5"?>
<grammar xml:lang="en-US" version="1.0" xmlns="http://
www.w3.org/2001/06/grammar" root="ROOT">
<rule id="ROOT" scope="public">

 <!-- the default grammar language is US English -->
...
</grammar>
```

RULE DEFINITIONS

Notes:

The building blocks of an XML form grammar are **rules**. Each rule contains a legal rule expansion that specifies what the caller may say to match that rule. Grammars are composed of many rules.

- Use **<rule id="*rulename*">** to name the rule. The content of <rule> is the rule expansion that defines what the caller can say to match this rule.

```
<rule id="city">
 <one-of>
    <item> Boston </item>
    <item> San Francisco </item>
    <item> Chicago </item>
 </one-of>
</rule>
```

- The `<rule>` element has three requirements:
  - The `id` attribute indicates the name of the rule (rulename) and *must* be unique within the grammar.
  - The content of `<rule>` may be any legal **rule expansion.**
  - The rule name (id) must be unique within the scope of the grammar that references it.
- Use the **`scope=`** attribute of the `<rule>` element to specify the scope of the rule. The scope designates the rule as **`"private"`** (can only be used locally within its parent `<grammar>`) or **`"public"`** (may be referenced by grammar rules defined in other files).
  - Rules default to "private" if the scope attribute is omitted.
  - These two rules are equivalent:

`<rule id="Work" scope="private">`
`to the office`
`</rule>`
	`<rule id="Work">`
	`to the office`
	`</rule>`

- The root rule is often named "ROOT" and should be included as an attribute of the <grammar> element. There is only one root per grammar.
  - The **root** rule should be declared with scope "**public**".

```
<grammar xml:lang="en-US" version="1.0" root="ROOT">
   <rule id="ROOT" scope="public">
      <count number="optional"> please </count>
     <ruleref uri="#Command"/>
    <ruleref uri="#Destination" />
   </rule>
```

Notes:

RULE EXPANSIONS

The content of <rule> is called the **rule expansion,** which defines the sequence and pattern of words the caller must say to match that rule. The following elements can be combined to create the rule expansion:

- token
- rule reference
- sequence
- alternatives
- counts

Tokens

- A token (or *terminal*) is that part of any grammar that defines the actual **word** or **words** that may be spoken and recognized. A token is any string that the recognizer can convert to a phonetic representation.
- Tokens are separated by white space. Multiple words may be combined into one token by enclosing the phrase within quotation marks or separating the words with underscores.

  `office (one token)`

  `to the office` (sequence of three tokens)

  `to_the_office, "to the office"` (one token each)
- Any plain or unmarked text (other than the content of <example>) that occurs as content of <rule> is a token.
- Tokens may also be indicated as content of the element <token> (rarely used)
- The following code sample contains four tokens: Boston, San_Francisco, New, and York

```
<rule id="dest_city">
 <count number="optional"> I’d like to fly to </count>
 <one-of>
    <item> Boston </item>
    <item> San_Francisco </item>
    <item> New York </item>
 </one-of>
</rule>
```

Alternatives

Notes:

Rule expansions can contain alternative items that callers may say. Only one of the items will be recognized.

- Use the **`<one-of>`** element to delimit the set of alternatives.
- Use the **`<item>`** element as a child of `<one-of>` to denote each alternative.
- The content of `<item>` must be a rule expansion that indicates an alternative. Each alternative expansion is contained in an `<item>` element. Only one of the <item>s will be recognized.

```
<one-of>
 <item>Michel</item>
 <item>Yukio</item>
 <item>Sylvia</item>
 <item>Duke</item>
 <item><ruleref uri="#otherNames"/></item>
</one-of>
```

Each alternative is equally likely unless you assign weights. Weights are indicated by the **`weight`** attribute of `<item>` as a child of `<one-of>`:

```
<one-of>
 <item weight="3.0">coke </item>
 <item weight="1">root beer </item>
 <item weight="0.1">sarsaparilla </item>
</one-of>
```

A weight of 1 is equivalent to no weight. A weight greater than 1 is a positive weight, and a weight less than 1 is a negative weight. A weight of 3 means that the item is 3 times as likely to be spoken as an item with a weight of 1, and 2 times as likely to be spoken as an item with a weight of 1.5. It is best to base weights on real speech input from a live system, and not on preconceived notions of what callers are likely to say.

Notes:

Rule References

A rule expansion may reference other rules by their names (identifiers). This improves readability, modularization and reuse of the grammar rules.

- The <ruleref/> element may occur anywhere a token may occur within the rule.

- <ruleref/> has no content so it is always self-closing.

- Use **<ruleref uri="*rulename*"/>** to reference a rule by its name. You may reference a rule that resides in a different grammar file, but only if that rule has been declared of scope "public".

```
<rule id="travel_info">
   from
   <ruleref uri="#city_name"/>
   to
   <ruleref uri="#city_name"/>
<rule>

<rule id="city_name">
 <one-of>
    <item> Boston </item>
    <item> "San Francisco" </item>
    <item> New York </item>
 </one-of>
</rule>
```

The following table summarizes the various forms of rule reference in the XML grammar form, though **only the local rule reference** is relevant in this course:

Reference Type	XML Form
Local rule reference	`<ruleref uri="#rulename"/>`
Reference to a rule in a different grammar file identified by the URI (that rule must be scope public)	`<ruleref uri="grammaruri#rulename"/>`
Reference to the **root** rule in a different grammar file identified by the URI	`<ruleref uri="grammaruri"/>`

Sequences

Notes:

Rule expansions can be made up of a sequence of many rule expansion elements (`<ruleref>, <item>, <one-of>, <count>, <token>`) and text.

The sequence implies the temporal order in which the expansions must be spoken by the user and recognized by the speech recognizer. Sequences are read from left to right.

If necessary, use an `<item>` element to surround elements of a sequence that you want to annotate with additional tags.

```
<!-- sequence of tokens -->
<rule id="test"> this is a test </rule>

<!-- sequence of rule references -->
<rule id="ROOT">
 <ruleref uri="#Command"/>
 <ruleref uri="#Destination"/>
 </rule>

<!--sequence of tokens and rule references-->
 <rule id="travel_info">
    from
    <ruleref uri="#city_name"/>
    to
    <ruleref uri="#city_name"/>
 <rule>

<!-- sequence container -->
<rule id="travel_dest">
 <item count="optional"> fly to </item>
 <ruleref uri="#city"/> </item>
</rule>
```

Notes:

Optional and Repeatable Elements

The <count> element - 1/01 SRGS working draft

The `count` element is used to designate that the enclosed rule expansion is either optional or repeatable two (2) or more times.

```
<rule id="Home">
    <count number="optional"> to my </count>
    home
    <count number="optional"> phone </count>
</rule>
```

The `<count>` element is found in the 1/01 SRGS working draft only, and is supported by OSR 1.x, OSR 2.x and OSR 3.0.

The **`number`** attribute of `<count>` indicates the number of times the contained expansion may be repeated. The allowed values are

- <count number="**optional**"> (optional)
- <count number="**?**"> (optional)
- <count number="**0+**"> (optional, but repeatable)
- <count number="**1+**"> (required, but repeatable)

Example,

```
<!-- "pizza" -->
<!-- "big pizza with pepperoni" -->
<!-- very big pizza with cheese, onions and pepperoni" --
>
<rule id="pizza_order">

 <count number="optional">
    <count number="optional"> very </count>
    big
 </count>
 pizza
 <count number="0+">
   <count number="optional">
     <one-of>
       <item> with </item>
       <item> and </item>
     </one-of>
   </count>
   <ruleref uri="#topping"/>
</count>
```

Repeat - SRGS CR&PR

Notes:

The **`repeat`** attribute is found in the 6/02 SRGS candidate recommendation and subsequent versions of the SRGS and is supported by OSR 2.0 and OSR 3.0.

```
<rule id="Home">
    <item repeat="0-1"> to my </item>
    home
    <item repeat="0-1"> phone </item>
</rule>
```

`repeat` is an attribute of **`item`** and is used to define a legal rule expansion as being optional, repeatable zero or more times, repeatable one or more times, or repeatable some number or range of times.

The `repeat` attribute of the `item` element has the following values:

- repeat="**n**" (repeat exactly n times, n must be 0 or a positive integer)
- repeat="**m-n**" (repeat from m to n times, inclusive, m must be greater than n)
- repeat="**m-**" (repeat m or more times)
- repeat="**0-1**" (the rule expansion is optional or may be said once)

```
<!-- Examples of the following expansion -->
<!-- "pizza" -->
<!-- "big pizza with pepperoni" -->
<!-- "very big pizza with cheese, onions and sausage" -->
<item repeat="0-1">
  <item repeat="0-1"> very </item>
   big
 </item>
 pizza
<item repeat="0-">
  <item repeat="0-1">
    <one-of>
      <item>with</item>
      <item>and</item>
    </one-of>
  </item>
  <ruleref uri="#topping"/>
</item>

<!-- allow the caller to say 1 or more digits -->
<item repeat="1-"> <ruleref uri="#digit"/> </item>

<!-- the caller must say between 4-6 digits -->
<item repeat="4-6"> <ruleref uri="#digit"/> </item>
```

Notes:

Example Phrases

For documentation purposes, you may include the <**example**> tag in the grammar file to provide one or more example phrases of what callers may say to match the grammar.

Examples are special **comments** ignored during recognition. Some recognition software uses it to create example lists or printed versions of grammar with automated testing.

The <example> element must be the **initial** content within a <rule>. The content of <example> is a sample of valid caller input.

```
<rule id="command" scope="public">
 <!-- A simple directive to execute an action -->
 <example> open the window </example>
 <example> close the door </example>

 <ruleref uri="#Action"/>
 <ruleref uri="#Object"/>
</rule>
```

Steps to Create a Grammar

Notes:

Follow these steps for creating a grammar:

- Use any plain text editor to write and edit grammar rules.
- Test the grammar using OSR tools. It is very important to test the grammar before compiling it and using it within an application. Command line tools are available with all versions of OSR.
- (optional) Compile the grammar from the command line using **sgc**.
- After all previous testing is completed, run a speech application that uses the written grammar. Call in and test.

Notes:

TESTING GRAMMARS WITH SPEECHWORKS TOOLS

There are two kinds of grammar testing, **off-line** and **on-line**.

- **Off-line** testing allows you to verify that the grammar allows precisely the set of intended responses.
  - Off-line testing uses tools (example, *parseTool and test_parser*) to verify that a grammar will return the appropriate results when passed the text equivalent of an utterance.
- **On-line** testing allows you to test the speech recognition.
  - On-line testing involves loading a grammar into a recognition engine via an IVR application and then calling the application to see if words or phrases are recognized when spoken.

Always do off-line testing with OSR tools before performing on-line testing.

During development, every grammar should be tested repeatedly **before attempting to compile and run** the grammar inside of an application.

The test cycle is very rapid and you can perform quick development iterations on your grammar while avoiding the more inefficient cycle of compiling the grammar, running the application, and testing application with speech input.

Testing a Grammar with parseTool

Notes:

Parsing is the process of disassembling the words or phrases spoken by a caller and matching them to rules in a grammar.

The **parseTool** is a command-line utility installed with OSR for testing that a grammar allows precisely the words and phrases required and no others

If an expected sentence doesn't parse, or if it parses more than once, then the grammar must be edited.

- To test if a phrase parses, use parseTool with the **-test_sentences** (-t_s) flag.

  `parsetool sample.grxml –test_sentences`

  - The following shows the result of using parseTool on one sentence that parses, and one that doesn't. The grammar used is the phone call redirect sample grammar found earlier in this module. **Items in bold are entered from the command line by the tester.**

```
C:\parseTool sample_grammar.grxml -test_sentences

Oct 21 10:55:16.60| 200|||| ** WARNING **| 0|  SWI_
SUCCESS| success| SWIconfigGetDefaultLanguage | Default
language 'en-us' autocomputed!
PROG parsetool:
  arg <spec-filename> == 14.grxml
  arg <-test_sentences> == -test_sentences

next sentence: send calls to my car phone
Parsing 'send calls to my car phone' with uri
'14.grxml'...
<?xml version='1.0'?>
   <result>
      <interpretation grammar="ParseToolGrammar"
confidence="100">
         <input mode="speech">
            send calls to my car phone
         </input>
         <instance>
            <SWI_literal>
               send calls to my car phone
            </SWI_literal>
            <SWI_grammarName>
               ParseToolGrammar
            </SWI_grammarName>
            <SWI_meaning>
               {SWI_literal:send calls to my car phone}
            </SWI_meaning>
         </instance>
      </interpretation>
   </result>
```

Notes:

```
Parse successful, line 1
```

- To generate a list of sentences allowed by the grammar, use parseTool with the **`-gen_sentences (-g_s)`** flag. Use the **`-max_gen`** parameter to specific the number of sentences to generate (default 10).

```
C:\parseTool sample_grammar.grxml -gen_sentences

Oct 21 10:57:26.81| 3056|||| ** WARNING **| 0|  SWI_
SUCCESS| success| SWIconfigGetDefaultLanguage | Default
language 'en-us' autocomputed!
PROG parsetool:
  arg <spec-filename> == 14.grxml
  arg <-gen_sentences> == -gen_sentences
send my calls home phone
please direct calls to the office
direct calls to the office
direct my calls to the office
please direct my calls to the office
direct calls to my car
please send calls to the office
please send my calls home
please direct calls to my car phone
please direct calls to the office
```

SEMANTIC INTERPRETATION

Notes:

To simplify and isolate the application’s dependency upon the grammar, a well designed grammar distinguishes between what the Caller actually speaks (the raw text that was recognized) and the semantic meaning of that speech (the answer).

For example, if there are five ways to say "direct my calls," the grammar should pass only one meaning to the application.

The industry push is toward Natural Language, allowing callers to speak "naturally" and in complete sentences. Natural Language Grammars must be written to extract and return the Caller’s meaning to the application through a set of key/value pairs. These key/value pairs are defined in the grammar by using **tags**.

- A **tag** is an arbitrary string that gives meaning to a rule or rule expansion
- A tag does not affect the legal word patterns defined by the grammars or the speech recognition process.
- Tags provide information that is typically used in post-processing speech recognition results, often to assign a semantic meaning in the form of key-value pairs ***keyname*=’*value*’**.
- Usually, the key (*keyname*) represents a variable name in the application and the value is assigned into that variable.
- Tags contain the scripting language ECMAScript. The value of the tag attribute can be a sequence of ECMAScript expressions, separated by semi-colons. ECMAScript is similar to JavaScript, but is nonproprietary.

Keys returned to application	Values returned to the application into the corresponding keys
key1=LOCATION	‘car’ or ‘home’ or ‘office’
key2=ACTIVITY	‘direct calls’

In the 1/01 SRGS working draft, tag= was an attribute. In the 6/02 candidate recommendation and beyond, <tag> is an element.

Notes:

Tags in the 1/01 SRGS working draft - the tag attribute

In the 1/01 SRGS working draft, a `tag` attribute may be attached to any of the rule expansion elements: `<ruleref>, <one-of>, <item>, <count>;` but may NOT be an attribute of `<rule>.`

- In this example, the key LOC gets one of three values ('home', 'car', or 'office') based on which rule is recognized

```
<rule id="Destination" >
  <one-of>
    <item>
      <ruleref uri="#Home" tag="LOC='home' "/>
    </item>
    <item>
      <ruleref uri="#Car" tag="LOC='car' "/>
    </item>
    <item>
      <ruleref uri="#Work" tag="LOC='office' "/>
    </item>
  </one-of>
</rule>
```

Tags in the SRGS CR&PR- the <tag> element

Notes:

In the 6/02 SRGS candidate recommendation and subsequent versions of the SRGS< a `<tag>` element is a rule expansion that can **ONLY be a child of the `<item>` and `<rule>`** elements**.**

The content of the `<tag>` element can be any arbitrary string of text. Tags may contain content for semantic interpretation, including ECMAScript.

Tags do not affect what is recognized by the grammar and do not affect the speech recognition process. Rather, they are typically used in post-processing speech recognition results that match a grammar's rule definitions and rule expansions.

```
<rule id="Destination" >
  <one-of>
    <item>
      <ruleref uri="#Home"/>
      <tag> LOC='home' </tag>
    </item>
    <item>
      <ruleref uri="#Car"/>
      <tag> LOC='car' </tag>
    </item>
    <item>
      <ruleref uri="#Work"/>
      <tag> LOC='office' </tag>
    </item>
  </one-of>
</rule>
```

Notes:

The tag-format Attribute (SRGS CR&PR)

The `tag-format` is an optional attribute of the `<grammar>` element that specifies the content of the `<tag>` element.

OSR supports the grammar scripting languages standardized by the W3C. However, the specification of the use of ECMAScript inside tags is not yet complete. To ensure compatibility with future versions of OSR, we encourage the use of the <grammar> attribute **tag-format="swi-semantics/1.0"**

```
<grammar xml:lang="en-US"
         version="1.0"
         root="ROOT"
         tag-format="swi-semantics/1.0"
         xmlns="http://www.w3.org/2001/06/grammar">
```

SEMANTIC INTERPRETATION EXAMPLES

Notes:

To understand the following grammars, you need to understand the following points:

- Each rule has a corresponding ECMAScript object, referred to as the **rule object.**
- Each rule object has **properties** that are set by scripts. The scripts run if the rule is included in the sentence parse.
- The script inside a rule can access keys/values returned by rule references (child objects) of that rule. **Always test existence of a key/value set by a child ruleref, accessing a nonexistent key/value causes runtime errors!**
- Rules are executed left-to-right. Scripts are executed by rules in left-to-right rule order after its parent rule executes.
- The root rule is special since each of its properties corresponds to a key/value pair **returned by the grammar to the application**.

Notes:

Grammar with Semantic Interpretation - 1/01 SRGS

```
<?xml version="1.0"?>
<grammar xml:lang="en-US" version="1.0" root="ROOT">
  <rule id="ROOT" scope="public">
    <count number="optional"> please </count>
    <ruleref uri="#Command"
    tag="ACTIVITY=Command.ACTIVITY"/>
    <ruleref uri="#Destination"
tag="LOCATION=Destination.LOC"/>
  </rule>
  <rule id="Command">
    <one-of tag="ACTIVITY='direct calls'">
    <item>direct</item>
    <item>send</item>
    </one-of>
    <count number="optional"> my </count>
    calls
  </rule>
  <rule id="Destination">
    <one-of>
    <item><ruleref uri="#Home" tag="LOC='home'"/> </item>
    <item><ruleref uri="#Car" tag="LOC='car'"/> </item>
    <item><ruleref uri="#Work" tag="LOC='work'"/></item>
    </one-of>
  </rule>
  <rule id="Home">
    <count number="optional"> to my </count>
    home
  </rule>
  <rule id="Car">
    to my car
  <count number="optional"> phone </count>
  </rule>
  <rule id="Work"> to the office </rule>
</grammar>
```

Grammar with Semantic Interpretation - SRGS CR&PR

Notes:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<grammar xml:lang="en-US" version="1.0" root="ROOT"
  xmlns="http://www.w3.org/2001/06/grammar"
  tag-format="swi-semantics/1.0">

  <rule id="ROOT" scope="public">
    <item repeat="0-1"> please </item>
    <ruleref uri="#Command"/>
    <tag> ACTIVITY=Command.ACTIVITY </tag>
    <ruleref uri="#Destination" />
    <tag>LOCATION=Destination.LOC </tag>
    <tag> SWI_meaning = ACTIVITY + ' ' + LOCATION </tag>
  </rule>

  <rule id="Command">
    <one-of>
      <item>direct</item>
      <item>send</item>
    </one-of>
    <item repeat="0-1"> my </item>
    calls
<tag> ACTIVITY='direct calls' </tag>

  </rule>
  <rule id="Destination">
    <one-of>
      <item> <ruleref uri="#Home" />
             <tag>LOC='home'</tag>
      </item>
      <item> <ruleref uri="#Car" />
             <tag>LOC='car'</tag>
      </item>
      <item> <ruleref uri="#Work" />
             <tag>LOC='office'</tag>
      </item>
    </one-of>
  </rule>
  <rule id="Home">
    <item repeat="0-1"> to my </item>
    home
  </rule>
  <rule id="Car">
    to my car
    <item repeat="0-1"> phone </item>
  </rule>
  <rule id="Work"> to the office </rule>
</grammar>
```

Notes:

SPECIAL KEYS

SpeechWorks OpenSpeech Recognizer (OSR) uses special keys to create meaning. Each of these keys begins with "SWI_".

Among the keys used with OSR are:

- SWI_literal - the raw text answer of what was recognized
- SWI_meaning - used to designate the same meaning for synonyms. This improves recognition confidence scoring when 2 synonyms sound alike.

SWI_literal

SWI_literal contains the raw text answer (the actual recognized text). It can be used within the ECMAScript and is available for every rule, including the root rule.

This example is compatible with the 1/01 SRGS working draft:

```
<rule id="Savings">
 <item>
    savings account number
     <ruleref uri="#AcctNumber"/>
 </item>
</rule>
<rule id="AcctNumber" tag="ACCT=SWI_literal">
 <ruleref uri="#Three_Digits"/>
 <ruleref uri="#Three_Digits"/>
</rule>
```

This example is compatible with the SRGS CR&PR:

```
<rule id="Savings">
 savings account number
 <ruleref uri="#AcctNumber"/>
</rule>

<rule id="AcctNumber">
 <tag> ACCT=SWI_literal </tag>
 <ruleref uri="#Three_Digits"/>
  <ruleref uri="#Three_Digits"/>
</rule>
```

SWI_meaning

Notes:

The SWI_meaning key is set only for the root rule.

Alternative phrases that mean the same thing must be assigned to the same SWI_meaning. This is important for two reasons.

1. Most platform integrations of OpenSpeech Recognizer use SWI_meaning to determine the result of recognition. Note, by default the value of SWI_ meaning equals the value of SWI_literal.

2. When a recognized phrase sounds like another phrase in the grammar, the recognizer often assigns a low confidence score, since it is "unsure" of which phrase is correct. If both phrases have the same value of SWI_meaning, they are considered synonyms, and OSR will only include one of the synonyms on the n-best list and not degrade the confidence value.

N-best list

Recognition results are returned in the form of an n-best list. The n-best list is composed of the n sentences that best match what the recognizer hears. An n-best list will have one or more items arranged from highest to lowest confidence value.

Consider the following yes/no grammar where synonyms for "yes" are represented with the SWI_meaning option.

This example is compatible with the 1/01 SRGS working draft:

```
<?xml version="1.0"?>
<grammar xml:lang="en-us" version="1.0" root="ROOT">
<!-- for OSR 1.x and the 1/01 SRGS working draft -->
 <rule id="ROOT" scope="public">
    <one-of>
      <item tag="SWI_meaning='true' ">yes</item>
      <item tag="SWI_meaning='true' " >yep</item>
      <item tag="SWI_meaning='false' ">no</item>
    </one-of>
 </rule>
</grammar>
```

Notes:

This example is compatible with the SRGS CR&PR:

```
<?xml version="1.0"?>
<grammar xml:lang="en-us" version="1.0"
tag-format="swi-semantics/1.0"
xmlns="http://www.w3.org/2001/06/grammar" root="ROOT">
<!-- for OSR 2.0 and OSR 3.0 and the SRGS CR&PR -->
 <rule id="ROOT" scope="public">
    <one-of>
      <item> <tag> SWI_meaning=’true’ </tag> yes </item>
      <item> <tag> SWI_meaning=’true/ </tag> yep
      </item>
      <item> <tag> SWI_meaning=’false’ </tag> no </item>
    </one-of>
 </rule>
</grammar>
```

Without SWI_meaning, the n-best list might include the following:

Text
yes
yep
no

When SWI_meaning is properly used, OSR filters out redundant answers so that entries are truly distinct:

Text	Top-level SWI_meaning key
yes	true
yep	
no	false

- If SWI_meaning is not explicitly defined on the root, it is constructed by concatenating all the keys defined in the root and their values.
- If there are no keys, SWI_meaning is set as SWI_literal.
- If SWI_meaning is an object, it is converted to a string.

TESTING KEY/VALUE PAIRS WITH PARSETOOL

Notes:

You can use **parseTool** with **-test_sentences** and the **-debug_output** flags to follow a sentence as OSR parses it through the grammar and assigns keys and values.

```
parseTool sample.grxml -test_sentences -debug_output
parseTool sample.grxml -t_s -d_o
```

For example:

```
C:\parseTool sample_grammar.grxml -test_sentences -debug_
output

Oct 21 11:00:31.79| 1940|||| ** WARNING **| 0|  SWI_
SUCCESS| success| SWIconfigGetDefaultLanguage | Default
language 'en-us' autocomputed!
PROG parsetool:
  arg <spec-filename> == 34.grxml
  arg <-test_sentences> == -test_sentences
  arg <-debug_output> == -debug_output

next sentence: send calls to my car phone
Parsing 'send calls to my car phone' with uri
'34.grxml'...
Step  0: Command
 Enviro: {SWI_vars:{}}
 Input : {}
 Script: ACTIVITY='direct calls'
 Result: {ACTIVITY:direct calls }

Step  1: ROOT
 Enviro: {SWI_vars:{}Command:{ACTIVITY:direct calls SWI_
literal:send calls SWI_spoken:send calls SWI_confidence:1
}}
 Input : {}
 Script: ACTIVITY=Command.ACTIVITY
 Result: {ACTIVITY:direct calls }

Step  2: Destination
 Enviro: {SWI_vars:{}Car:{SWI_literal:to my car phone
SWI_spoken:to my car phone SWI_confidence:1 }}
 Input : {}
 Script: LOC='car'
 Result: {LOC:car }

Step  3: ROOT
 Enviro: {SWI_vars:{}Command:{ACTIVITY:direct calls SWI_
literal:send calls SWI_spoken:send calls SWI_confidence:1
```

Notes:

```
}Destination:{LOC:car SWI_literal:to my car phone SWI_
spoken:to my car phone SWI_confidence:1 }}
 Input : {ACTIVITY:direct calls }
 Script: LOCATION=Destination.LOC
 Result: {ACTIVITY:direct calls LOCATION:car }

Parse 0: {{send calls Command} {{to my car phone Car}
Destination} ROOT}
<?xml version='1.0'?>
  <result>
    <interpretation grammar="ParseToolGrammar"
confidence="100">
      <input mode="speech">
        send calls to my car phone
      </input>
      <instance>
        <ACTIVITY confidence="100">
          direct calls
        </ACTIVITY>
        <LOCATION confidence="100">
          car
        </LOCATION>
        <SWI_literal>
          send calls to my car phone
        </SWI_literal>
        <SWI_grammarName>
          ParseToolGrammar
        </SWI_grammarName>
        <SWI_meaning>
          {ACTIVITY:direct calls LOCATION:car}
        </SWI_meaning>
      </instance>
    </interpretation>
  </result>
Parse successful, line 1

next sentence:
```

Notes:

TESTING KEY/VALUE PAIRS WITH TEST_ PARSER.EXE

Another OSR tool for testing a set of input sentences and associated key/value pairs is called **test_parser**. Use test_parser to perform a regression test that compares the correct key/value pairs with those actually generated. The tool operates on a text file containing a set of test sentences. Each line of the file defines a test or directive.

- Syntax:

  ```
  test_parser <sample text file>
  ```

- Each test line in the test file is of the form:

  ```
  <xml grammar file> <test sentence> <key name> <correct
  value for key name>
  ```

This sample file can be used to test the above example grammar (when the same grammar file is used on multiple lines, you may insert a hyphen instead of the grammar name):

```
sample.grxml  "send calls to my car phone"   LOCATION   "car"

-              "send calls to the office"     LOCATION   "car"
```

The second line of the test file is should not parse. This file would generate the following result, which shows that 2 lines were tested, and one generated an error:

```
C:\test_parser test_set.txt

Oct 21 11:12:41.33| 3236|||| ** WARNING **| 0|  SWI_
SUCCESS| success| SWIconfigGetDefaultLanguage | Default
language 'en-us' autocomputed!
PROG test_parser:
  arg <spec-filename> ==
  arg <-test_file filename> == test_set.txt
  arg <-test_sentences> == -test_sentences
  arg <-batch> == -batch
Error: Unmatched value, wanted 'car', got 'work' for
LOCATION, line 2.
1 errors!!!
2 0.000000
```

Notes:

Module Review

Now that you have completed this module, you should be able to:

- Recognize the rules and rule references you use to build XML grammars
- Build basic XML grammars
- Utilize key/value pairs to streamline semantic interpretation
- Use SpeechWorks OpenSpeech Recognizer Tools to test XML grammars