Internet Applications

Lecture 12/13 – VoiceXML 2.0

See: http://www.w3.org/TR/voicexml20

Steve Young, Feb 2003
Modified by Jason D. Williams, Feb 2004

Cambridge University Engineering Department
Machine Intelligence Lab
A Simple Example

```
<vxml version="2.0" xmlns="http://www.w3.org/2001/vxml">
  <form>
    <field name="service">
      <prompt> Do you want news, sport or weather information? </prompt>
      <grammar src="service.gram" type="application/srgs" />
    </field>
  </form>
</vxml>
```

S: Do you want news, sport or weather information?
U: Hmmm, football please
S: I did not understand you.
S: Do you want news, sport or weather information?
U: Sport
S: [continues in service.asp script]
Goals

- minimise number of client-server interactions
- shields application authors from low-level platform details
- separates user interaction code (VoiceXML) from service logic
- promote portability by shielding voice dialog authors from specific hardware implementations
- easy to use for simple dialogs, yet capable of supporting complex mixed-initiative dialogs

Features

- recognition of spoken input and dtmf tones
- message output using recorded audio and synthesis
- recording of spoken input
- basic telephony control such as call transfer, disconnect
- dialog state-machine with (state is a <form>)
  - mixed initiative within states
  - user-driven state switching
  - access to ECMAScript (aka Javascript)
Limitations

- dialogues are essentially state-based (vs. rule-based), good for system-driven dialogues but tangled for user-driven dialogues
- limited opportunity for implicit correction and recovery
- dialogue designer has limited control over active grammar complexity
- extension to multi-modal mark-up not obvious (but see XHTML+Voice)
- very little call control (but see CC-XML)
- no cookies
- proprietary extensions abound

Key message

- despite limitations, broadly welcomed & increasingly deployed

Goal of this lecture

- explain the spec (theory) and typical practices (real-world)
**VoiceXML Applications & Sessions**

**Application** consists of a set of VoiceXML documents plus an optional root document which is loaded automatically, and persists until the browser exits the application.

**Session** is the duration of the user interaction (e.g., a phonecall).
Dialogs

User is always in one conversational state, or “dialog”, at a time.

An application consists of a set of dialogs, each dialog is represented by a `<form>` element. Application progresses by making transitions from form to form, both within current document and by loading new documents.

```
<form>
  
  
<form>
</form>
```

```
<form>
  
  
<form>
  
</form>
</form>
```

```
<form>
  
  
</form>
```

```
<form>
  
</form>
```

- “Document order”
- The session (normally) ends when:
  - User initiated: Hang-up
  - System initiated: explicit command (e.g., `<disconnect/>`)
  - System initiated: implicit end – i.e., “Fall off” end of page, or no successor form specified

Form analogies:
- “State”
- “HTML Form”
- “Semantic Frame”
Forms

Forms are the basic building block of VoiceXML documents

A form is a sequence of items: blocks and fields.

Every item has a variable associated with it which is initially undefined.

Block items are set true on entry to the block.

Field items are set when the user supplies a value.

Form Interpretation Algorithm (FIA)

a) find first undefined item
b) enter it
Repeat until all items defined.
System output (prompts)

- Text between <prompt> ... </prompt> is replayed as given (as TTS)
- The text can be “marked-up” using
  - <break> insert a pause
  - <emphasis> to emphasise something
  - <prosody> to control prosody
  - <say-as> to specify a phone sequence or speak in a particular style
- Variable values can be inserted using the <value expr="varname"> </value>
- Audio files can be inserted using <audio src="url"> </audio>
  - Audio usually cached in a prompt cache to minimize fetches
- Prompts can be selected depending on number of replays
- Barge-in can be turned on and off

```
<prompt bargein="false">
  <audio src="thanksforshopping.wav">Thanks for shopping with us! </audio>
  I’m sending you <value expr="qty"> <value expr="colour"/> TeeShirts 
  size <value expr="size"/></value>. To check on delivery status, call 
  <say-as type="phone"> 01223 332752 </say-as>
</prompt>
```
Grammars

- A VoiceXML interpreter can support a range of grammar formats, however all interpreters must support the standard W3C grammar format “Speech Recognition Grammar Specification” – SRGS.
- SRGS comes in two flavours: **readable ABNF** and **XML compliant**
- Tools exist to map between them i.e. they are equivalent.
- The ABNF version is essentially a refinement of the Java Speech Grammar Format JSGF
- Right-branching recursion allowed

ABNF example – header & re-write rules

```abnf
#ABNF 1.0 ISO-8859-1;
language en-GB;
mode voice;
root $top;
$place = London | Manchester | Leeds;
$leavefrom = from $place;
$destination = to $place;
$top = i want to go $leavefrom $destination
```

Optional; also: `<grammar root="…">`

rule name: `$place` = London | Manchester | Leeds;

Token: `$leavefrom`, `$destination`
Rules can reference rules in other grammars via a URL:

$cities = \langle http://www.mygrammars.com/world-cities.gram#canada \rangle
$\langle http://www.gramsUS.com/US-cities.gram#vermont \rangle$

The usual BNF conventions apply for options and alternatives:

<table>
<thead>
<tr>
<th>Alternatives: a</th>
<th>or</th>
<th>b</th>
<th>or</th>
<th>c</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>b</td>
<td>c</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Options: a | c | or | a | b | c

Alternatives can also have weights

/0.3/ a | /0.5/ b | /0.2/ c

Rules can be repeated by adding a repeat specifier:

$id = \langle digit <1-10> \rangle$; 1 to 10 digits
$amexcard = \langle digit <15> \rangle$; exactly 15 digits
$visacard = \langle digit <16> \rangle$; exactly 16 digits
$password = \langle letter <6-> \rangle$; 6 or more letters
$number = \langle digit <1-> \rangle$; 1 or more digits

Alternatives can also have weights

/0.3/ a | /0.5/ b | /0.2/ c
DTMF Grammars

In VoiceXML DTMF tones can also be recognised as an alternative to voice. Usually, a DTMF grammar is placed in parallel with a regular grammar allowing the user the option of either speaking or pressing keys. Eg

```
<grammar mode="speech" type="application/srgs">
  [dark] red {red} | [dark|light] blue {blue} | ......
</grammar>
<grammar mode="dtmf" type="application/srgs">
  1 {red} | 2 {blue} | 3 {green} ......
</grammar>
```

Here the user can either say “red” or press key 1 on telephone, returned value is the same in either case.

A DTMF grammar is similar to a regular grammar except that the only vocabulary “words” allowed are the keys 1, 2, ..., 9, 0, *, #
Field Grammars - three options

1) Specify grammar source in a separate file
   eg
   `<grammar src="colour.gram" type="application/srgs"/>

2) Specify grammar source in-line grammar, eg
   `<grammar type="application/srgs">
     [dark] red {red} | [dark|light] blue {blue} | ......
   </grammar>

3) Use one of the built-in field types
   - boolean eg “yes”, “no”, “true”, “that is correct”
   - date eg “tomorrow”, “12th June”, “next monday”
   - digits eg “two one nine four one”
   - number eg “three thousand six hundred and nine”
   - currency eg “six pounds fifty pence”
   - phone eg “double three two six five four”
   - time eg “twelve thirty pm”, “eleven o’clock”

   eg
   `<field name="size" type="number">
   or
   `<grammar src="builtin:grammar/number"/>
   `<grammar src="builtin:dtmf/number"/>
Example: input/output/field grammars

```html
<form id="getorder">

...  

  <field name="size">
    <prompt> What size do you want? </prompt>
    <grammar src="sizes.gram" type="application/srgs"/>
    <filled>
      <prompt> Ok, size <value expr="size$.utterance"/>
      </prompt> </filled>
  </field>

  <field name="colour">
    ...
  </field>

  <field name="confirm" type="boolean">
    <prompt> Please confirm that you require a 
             <value expr="colour"/> Tee Shirt in size 
             <value expr="size"/>.
    </prompt>
    <filled>
      <if cond="confirm">
        <submit next="checkout.asp" namelist="size colour"/>
      </if>
      <prompt> Oh, sorry, let me try again. </prompt>
      <clear namelist="colour size confirm"/>
    </filled>
  </field>

</form>
```

Reference to grammar source

- `<filled>` executed when its field has been filled.
- **Note:** shadow variable

Field “type” (built-in grammar): No reference to grammar source

- If all ok, submit info & move on

Otherwise, clear everything and start again. (NB: prompt queue)
Rules can have **semantic interpretation tags**:

```javascript
$time = ($hour $min) { $.hour = $hour; $.min = $min } |
    ($qual $hour) {$.hour = $hour + $qual.hour; $.min = $qual.min };

$qual = quarter (to {$.hour = -1; $.min = 45} | past {$.hour = 0; $.min = 15} ) |
    half past { $.hour = 0; $.min = 30 };

$hour = one {1} | two {2} | three {3} | …
$min = [oh] five {5} | ten {10} | fifteen {15};
```

- Every rulename is also a variable referenced implicitly by $ or explicitly via $name
- Rule vars can be simple variables eg $min = 30 or structs eg $.hour = 1
- When no explicit assignment is given, {x} is short-hand for $ = x

When applied to “quarter to six”, the above would return

```
{ hour: 5; min: 45; }
```

*Tag contents a subset of ECMAscript*
Binding a field grammar semantic tag to its field name

- Grammar tag structure not necessarily congruent with field names
- By default, field name = slot
- However, forcing 1:1 mapping limits re-use of grammars

Grammar can return:
{ a : ValA } or
{ b : ValB } or
{ a : ValA, b : ValB
or something else

Associate the <field> with a tag using the slot attribute

<form id="...">
<field name="myField" slot="...">
<grammar ...
...

RULES:
1. If a top-level tag (struct) = "slot" is filled, field = its value.
2. If not, field = whole result

<table>
<thead>
<tr>
<th>If slot = ...</th>
<th>and field gram returns ...</th>
<th>... then myField =</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>{ a : ValA }</td>
<td>ValA</td>
</tr>
<tr>
<td>a</td>
<td>{ a : ValA, b : ValB }</td>
<td>ValA</td>
</tr>
<tr>
<td>a</td>
<td>{ b : ValB }</td>
<td>{ b : ValB}</td>
</tr>
<tr>
<td>X</td>
<td>{ a : ValA, b : ValB }</td>
<td>{ a : ValA, b : ValB }</td>
</tr>
<tr>
<td>x.y</td>
<td>{ x : {y : ValY, z : ValZ }}</td>
<td>ValY</td>
</tr>
<tr>
<td>x</td>
<td>{ x : {y : ValY, z : ValZ }}</td>
<td>{y = ValY, z = ValZ }</td>
</tr>
</tbody>
</table>
**Events**

**Events** are provided to handle error and normal situations. They are generated
a) by the underlying platform (e.g., user doesn’t respond, or requests help)
b) explicitly by VoiceXML scripts using the `<throw>` element
c) by the interpreter when errors occur

eg:

- `noinput`: User didn’t say anything before timeout
- `nomatch`: Reco result below confidence threshold
- `help`: User asked for help
- `connection.disconnect.hangup`: User hung up
- `error.badfetch`: Couldn’t load URI
- `error.semantic`: Run-time VoiceXML error

Events are handled by `<catch>` elements:

```
<catch event="noinput">…</catch>
<catch event="nomatch">…</catch>
<catch event="help">…</catch>
<catch event="error">…</catch>
```

**Syntactic sugar** for commonly-caught events
Example: events

```xml
<form id="getorder">
  <block>
    Welcome to Super-cool Tee-Shirts
  </block>
  <field name="size" type=number>
    <prompt>What size do you want?</prompt>
    <help>Please say the size in range 8 to 16.</help>
    <noinput count="1">
      Sorry, I didn’t hear anything. <reprompt/>
    </noinput>
    <noinput count="2">
      Sorry, I still didn’t hear anything. Say a number from 8 to 16.
    </noinput>
    <noinput count="3"><throw event="maxnoinput"/></noinput>
  </field>
  <block>
    <submit next="getline.asp" namelist="size"/>
  </block>
</form>
```

Catches user-defined event raised anywhere on this page. (Could have also put this in the application root document.)

Using `<catch>` shorthand

Forms & form items maintain counts for each event raised within it. **Count** attribute gives us different handlers depending on how many times we’ve seen the event in this form item.
Executable Content

VoiceXML supports two compatible scripting environments:

a) a built-in language based on a set of executable tags
b) support for ECMAScript via <script> tags

The built-in language is designed to be highly secure. Implementors wishing to minimise security risks may disable ECMAScript.

The built-in language is based on the following tags:

```
<var name="x[0]" expr="1">
<assign name="x" expr="y+1">
<if cond="x &gt; 10">... <else/>..</if>
<goto next="url">
<return namelist="x" >
```

var $x[0] = 1;

x = y+1;

if (x>10) { } else { };

goto "url";

return x;

In addition <clear>, <prompt>, <reprompt>, <submit>, <exit> and <disconnect> can appear within executable content.
Variable Scope

The scope of a variable depends on where it is declared. To see if a variable $x$ is in scope from some point, the scope chain is traced back upwards until a declaration for $x$ is found.

If a variable $x$ in a higher layer is hidden by a similar declaration in a lower layer, then the levels can be named explicitly e.g.

document.$x$, application.$x$, session.$x$, etc
Example: Executable content & var scope

```xml
<form id="getsize">
  <field name="size" type="number">
    <prompt>What size?</prompt>
    <filled>
      Got it, size <value expr="size"/>
      <assign name="document.size" expr="size"/>
      <goto next="#submit"/>
    </filled>
  </field>
</form>

<form id="submit">
  <block>
    <prompt>
      Submitting your order for a size
      <value expr="size"/>
    </prompt>
  </block>
</form>
```

- `<script>` code – initializes document-scope `size`
- `<field>` name is also `size` – hides `document.size`
- To access document-scope `size`, need to use `document.size`. Need to store (or submit) value of size in a higher-level scope for it to be available in the next form
- When no scope is specified, use the most locally scoped variable.
Adding mixed-initiative behaviour

To this point, the “form” construct has provided little value. The real value of a form lay in creating multi-field, mixed-initiative interactions.

Idea:
• Begin with an open question like “What’s your T-Shirt order?”
• Within form, allow a user to drive how we visit <field>’s (e.g., colour, size)

Needs:
• Way of asking an initial question ... <initial>
• Specifying a grammar “shared” by several fields ... form-level grammar
• Algorithm to control how we visit fields ... Form Interpretation Algorithm

Proviso: Multi-field, mixed-initiative dialogs are used very rarely in industry... at least today
Mixed-initiative Forms

By default, each field of a form is visited in turn, and each field has its own grammar. VoiceXML also allows a form level grammar to be specified

```
<form id="form name">
  <grammar src="form.srgs" />

  <initial>
    <prompt> "form level prompt" </prompt>
  </initial>

  <field name="fieldA">
    <prompt> ... </prompt> <grammar src="fieldA.srgs" />
  </field>

  <field name="fieldB">
    <prompt> ... </prompt> <grammar src="fieldB.srgs" />
  </field>

  etc
</form>
```

form level grammar allowing user to specify one or more of fieldA, fieldB, etc in a single utterance.
initial element specifies what to say when prompting at form level

Basic operation

1. prompt at form level; and recognise using “form.srgs”
2. if any fields still unfilled, then visit each field in turn using field level prompts & grammars
The Form Interpretation Algorithm (FIA)

Field-level AND form-level grammars

1. **Select phase**: Find the first form item for which guard condition = “false”.
2. **Collect phase**: “visit” that item.
   - **<initial>** :
     a. Load & compile grammars at <form> and higher levels
     b. Execute contents in document order (including <prompt>)
     c. Start playing prompt queue & activate recogniser; wait for result or event
   - **<block>** :
     a. Execute contents in document order (including <prompt>)
     b. Define the name of the <block>.
     c. Go to (1).
   - **<field>** :
     a. Load & compile any <grammar>’s within this <field> and <form> & higher levels
     b. Play <prompt>’s & activate recogniser; wait for result or event

3. **Process phase**:
   - If any form or field level grammar item is matched, define the name of <initial>.
   - If a grammar item is matched, execute the contents that or those <field>’s <filled>
     - **May be multiple <field>’s which have been matched**
   - If an event is thrown, handle that event.
   - Go to (1)
**Tee-Shirt example with mixed-initiative**

```xml
<form id="getorder">
  <grammar src="teeshirt.srgs" type="application/srgs"/>
  <initial>
    <prompt>
      Welcome to Super-cool Tee-Shirts.
      What size and colour do you want?
    </prompt>
  </initial>
  <field name="size" type=number>
    <prompt> What size do you want? </prompt>
  </field>
  <field name="colour">
    <prompt> What colour do you want? </prompt>
    <grammar src="colour.srgs" type="application/srgs"/>
  </field>
  <block>
    <submit next="checkout.asp" namelist="size colour" />
  </block>
</form>
```

### Example 1:

S: Welcome to Super-cool Tee-Shirts
What size and colour do you want?

U: Red in size ten please.

S: [ continues in checkout script ]

### Example 2:

Or

S: Welcome to Super-cool Tee-Shirts
What size and colour do you want?

U: Red please.

S: What size do you want?

U: Ten

S: [ continues in checkout script ]
Handling failures in mixed-initiative forms

S: Welcome to Super-cool Tee-Shirts
   What size and colour do you want?
U: Errmmm.
S: Sorry I didn't understand that. Please say something like
   “Red size 12” or “Size 10, colour red”
U: Cool, what do you have?
S: Sorry I still don't understand. I will prompt you
   one item at a time.
   What size do you want?
U: Ten
S: What colour do you want?
U: Red
S: [ continues in checkout script ]
Binding a *form* grammar semantic tag to a field name

- Still associate the `<field>` with a tag using the slot attribute
- However, different matching algorithm than field-level grammars!
- An utterance may not fill a particular field

```javascript
{ drink: "coke",
  pizza: {
    size: "large",
    topping: ["ham", "veg"]
  }
}
```

**RULES:**

1. If a top-level tag (struct) = “slot” is filled, field = its value.
2. If not, field *is not matched*

<table>
<thead>
<tr>
<th>If <code>utt</code> is as above, and slot = …</th>
<th>… then <code>myField</code> =</th>
</tr>
</thead>
<tbody>
<tr>
<td>drink</td>
<td>“coke”</td>
</tr>
<tr>
<td>pizza</td>
<td>{size: &quot;large&quot;, topping: [“ham”, “veg”]}</td>
</tr>
<tr>
<td>pizza.size</td>
<td>“large”</td>
</tr>
<tr>
<td>pizza.topping</td>
<td>[“ham”, “veg”]</td>
</tr>
<tr>
<td>sandwhich, pizza.number, size</td>
<td><em>Does not match field</em></td>
</tr>
<tr>
<td><em>undefined</em> (ie, name &amp; slot blank)</td>
<td><em>Does not match field</em></td>
</tr>
</tbody>
</table>
Other tags

• Associate a grammar with a `<goto>`: `<link next="URI"> <grammar ... /> </link>`

• Cause a grammar to fire an event: `<link event="eventName"> <grammar ... /> </link>`

• Record (don’t recognise) an utterance: `<record name="..." ... > ... </record>`

• A higher-level, simpler version of a `<form>` for choosing 1 of N options: `<menu>`
  • Choices: `<choice next="URI"> grammar text </choice>`
  • List choices in a prompt: `<enumerate/>`

• Platform-specific components: `<object name="oName" classid="..." data="...">`
  • Like sub-dialog, pass in arguments using `<param name="..." expr="..."/>`
  • Result returned in `oName` ECMAScript variable

• Control platform features (e.g., timeout): `<property name="..." value="..."/>`

• Generate a debug message in the log: `<log> ... </log>`

• App-specific meta-information: `<meta> ... </meta> and `<metadata> ... </metadata>`

• Cause `<prompt>` content to play again after event handler: `<reprompt/>`
APPENDIX

Students are not responsible for new material beyond this point
When is document order important?

- `<form>`’s: by default, user is in the first `<form>` on page. Override with http://server.com/page.vxml#form2

- Form items like `<field>` and `<block>`: FIA looks considers these in document order

- Within a `<block>` and `<filled>`: execute code in document order

- Within a `<field>`: ordering of `<prompt>` elements with respect to each other is important.

- Within a `<field>`: all other ordering is not important. (FIA looks at WHOLE `<field>` before launching recognition). For example, these two fields are equivalent:

```xml
<field name="number">
  <prompt>What’s your account number?</prompt>
  <grammar src="accountnum.gram" ...>
  <filled>
    <prompt>Got it.</prompt>
    <goto next="#password"/>
  </filled>
</field>

<field name="number">
  <filled>
    <prompt>Got it.</prompt>
    <goto next="#password"/>
  </filled>
  <grammar src="accountnum.gram" ...>
  <prompt>What’s your account number?</prompt>
</field>
```
Grammars

- Grammar specifies what a user can say at a given time
- To perform recognition, the textual description of a grammar must be compiled
- Most VoiceXML grammars are rule-based
  - Platform must support compilation-on-demand for rule-based grammars
  - Most grammars are re-used: cache of compiled grammars (GDB) is crucial
- Proprietary extensions support client-side, pre-compiled SLMs
- Built-in grammars: Pre-compiled; available to all applications; parameterisable
12.8 Mixed-initiative at Document Level

In addition to mixed-initiative within forms, VoiceXML also allows mixed-initiative between forms. Consider:

```xml
<form id="sell">
    <grammar src="items.srgs" scope="document" />
    <field name="mops"> ....
    <field name="buckets"> ....
    <field name="cloths"> ....
    <field name="checkout"> ....
</form>

<form id="checkout">
    <grammar src="checkout.srgs" />
    <field name="card_number"> ....
    <field name="expiry_date"> ....
</form>
```

The grammar is active throughout the document.

In checkout form, both the “items” and the “checkout” grammar are active.

S: Any more items to buy?
U: Two mops
S: Any more items to buy?
U: No checkout please
S: What is your card number?
U: Oh, and one bucket.
S: Any more items to buy?
U: No checkout please
S: What is your card number?

Jump back to sell form.
Grammar Scope

Note form scope can be overridden for each grammar.
Sub-Diagslogs

Dialogs can be structured into sub-dialogs and invoked in a hierarchical fashion as needed.

A subdialog call is represented by a `<subdialog>` tag and it replaces a `<field>` element within a form.

call dialog stored in cardnum.vxml passing value of “cardType” as a parameter

result is returned as an object, with properties set by cardnum

<filled> element checks return values and assigns to form variables as required
Example sub-dialog

Called subdialog collects parameters, and then executes in an environment which is identical to that of the calling <subdialog> element.

getcardnum.vxml

```vxml
<form id="getcardnum">
  <var name="type"/>
  <field name="number" type="digits">
    <prompt>
      Please give your <value expr="type"/> card number
    </prompt>
    <filled>
      <var name="status" expr="CheckNumber(number,type)"/>
      <return namelist="number status"/>
    </filled>
  </field>
</form>
```

set automatically via callers param “type”

Check number is valid for given type of card, return result of check in status

return number and status to the caller
Sub-dialogs are not sub-routines

• Called subdialog collects parameters, and then executes in an environment which is completely independent of the calling <subdialog> Element.
  • Independent variable space
  • Independent event hierarchy
  • Independent application root document
  • …

• This means that higher-scope variables are not visible, and events are handled in their own space.
• This is true even if the subdialog & calling form are on the same page!

Remember that the user doesn’t experience “subdialogs” – they experience a conversation

Use sub-dialogs with care!
Dynamic Components

A VoiceXML application can access dynamic components using the <Object> tag. This is similar to the <Object> tag in HTML.

```xml
<object name="objname" classid="..." data="...">
    <param name="arg1" expr="val1" />
    ...
</object>
```

Object behaves like a field, it returns a value as field item which can then be used to access the objects properties and methods. For example,

```xml
<field name="cardNum" type="number">
    <prompt> What is your card number? </prompt>
</field>
<object name="credcard" classid="CLSID:3452 ...">
    <param name="holder" expr="callerName"/>
    <param name="number" expr="cardNum"/>
    <if cond="credcard.valid">
        <goto next="#checkout"/>
    </fi>
    <clear namelist="cardNum credCard" />
</object>
```

NB content of <object> tag is executed cf HTML
Links

Links can be added to a dialogue to allow the user to go to a specific point by a single voice command. They add a further mechanism for the user to take the initiative.

A link consists of a grammar and a target. The scope of the grammar is the same as the scope of the enclosing element. The target can be a URL or an event to “throw”. For example,

root document

```
<link event="help">
  <grammar type="application/srgs">
    help [me] | what can I do now | i'm lost | ...
  </grammar>
</link>

<link next="http://www.cl.cam.ac.uk/cstit">
  <grammar type="application/srgs">
    I need to learn about speech and language | 
    what is speech and language | 
    what is a hidden markov model
  </grammar>
</link>
```

- throw the help event if user appears to be lost
- if the user needs educating, jump to a new web site
Menus

Menus are a special type of form consisting of a single anonymous field. They add nothing new to the expressive power of VoiceXML, but are included for convenience. Example,

```xml
<menu id="service">
  <prompt> What service do you want? </prompt>
  <choice next=http://www.infovox.com/weather.vxml > Weather </choice>
  <choice next=http://www.infovox.com/sport.vxml > Sports </choice>
  <noinput> Please say one of <enumerate/> </noinput>
</menu>
```

Shorthand for
```
<catch event="noinput">
  creates a grammar consisting of a list of options
  enumerates the list of choices
</catch>
```

Shorthand for
```
<noinput> Please say one of <enumerate/> </noinput>
```
The Form Interpretation Algorithm (FIA)

Only field-level grammars, including handling of <reprompt>

1. **Select phase**: Find the first form item for which guard condition = “false”.

2. **Collect phase**: “visit” that item.
   - If (this item == different item to last visit) \{newitem = true\} else \{newitem = false\}
   - <block> :
     a. Execute contents in document order.
     b. If (newitem || repromptflag) \{ add <prompt>’s to prompt queue \}
     c. Define the name of the block.
     d. repromptflag = false
     e. Go to (1).
   - <field> :
     a. If (newitem || repromptflag) \{ add <prompt>’s to prompt queue \}
     b. repromptflag = false
     c. Load & compile any <grammar>’s within this <field>
     d. Start playing prompt queue & activate recogniser; wait for result or event

3. **Process phase**:
   - If a grammar item is matched, execute the contents of <filled> (if present).
   - If an event is thrown, handle that event.
     a. If a <reprompt/> is present, set repromptflag = true
     b. Go to (1)
The Form Interpretation Algorithm (FIA)

Field & Form level grammars, including handling of <reprompt> tag

1. **Select phase:** Find the first form item for which guard condition = “false”.

2. **Collect phase:** “visit” that item.
   - If (this item == different item to last visit) {newitem = true} else {newitem = false}
   - **<initial>**:
     - Execute contents in document order.
     - If (newitem || repromptflag) { add <prompt>'s to prompt queue }
     - repromptflag = false
     - Load & compile grammars at <form> and higher levels
     - Start playing prompt queue & activate recogniser; wait for result or event

   - **<block>**:
       - If (newitem || repromptflag) { add <prompt>'s to prompt queue }
     - b. Define the name of the <block>.
     - c. repromptflag = false
     - d. Go to (1).

   - **<field>**:
     - a. If (newitem || repromptflag) { add <prompt>'s to prompt queue }
     - b. repromptflag = false
     - c. Load & compile any <grammar>'s within this <field>
       - a. Unless <field> has attribute (modal = true), Load & compile grammars at <form> & higher
     - d. Start playing prompt queue & activate recogniser; wait for result or event

3. **Process phase**:
   - If any form or field level grammar item is matched, Define the name of <initial>.
   - If a grammar item is matched, execute the contents that or those <field>'s <filled> (if present; may be multiple <field>'s).
   - If an event is thrown, handle that event.
     - a. If a <reprompt/> is present, set repromptflag = true
   - Go to (1)