Speech Technology in CE Task Guide

Intelligent User Interfaces

Professor Charles Rich
Computer Science Department
rich@wpi.edu

Options for Speech Systems

- Dictation vs. Grammar-Driven
- Microphone:
  - open (typically adaptive)
  - push-to-talk
  - push-to-talk with automatic endpointing
- Speaker Trained vs. Independent

Sphinx4 Speech Engine

- a state-of-the-art speech recognition system written entirely in Java
- joint effort between Carnegie Mellon University, Sun Microsystems Laboratories, Mitsubishi Electric Research Labs, and Hewlett Packard, with contributions from the University of California at Santa Cruz and the Massachusetts Institute of Technology
- http://cmusphinx.sourceforge.net/sphinx4/

Sphinx4 Data Files

- Acoustic and language model
  - speech/WSJ_8gau_13dCep_16k_40mel_130Hz_6800Hz.jar
  - 20,000 words, 10.6MB (considered “medium size”)
  - trained on people reading the Wall Street Journal
- Dictionary
  - java/edu/wpi/cetask/guide/commands.dict
  - CMU dictionary: 125,000 words
  - add phonetic spelling of new words here, e.g.,
    HEINLEIN HH AY N L AY N

JSGF Grammars

- http://java.sun.com/products/java-media/speech/forDevelopers/JSGF
- restrict recognition to only words and order of words specified by grammar
- greatly increases effectiveness of recognition
- greatly restricts what person can say --- *how does person know what s/he can say?*
- output of successful recognition can be:
  - parse tree
  - sequence of "tags" (see later)
  - serves as input to semantic interpretation ("understanding")

JSGF Grammars in CE Task Guide

- java/edu/wpi/cetask/guide/commands.gram
  - generic grammar
  - includes commands, such as 'status', etc.
  - output sequence of tags directly interpreted as command line input
  - stubs for application-specific task names, parameters, etc.
- models/Library.gram
  - extension grammar associated with task model (e.g., Library.xml)
  - provides specialized expressions for tasks and parameters

commands.gram

```
public <commands> =  <task> | <by> | <next> | <done> | <execute> | <yes> | <no> | <status>
                   | <clear> | <quit> | <verbose> | <debug>;

<task> = ( ( Let's | I want to | Please ) [ perform ] ) { task } <task_name> [ <values> ];

<by> = by { by } <decomp_name>;

<values> = [ on | of ] <NULL> { / } <value> ( [ and ] <NULL> { / } <value> )*;

<value> = something |
          ( failed { false } | succeeded { true } | <digit> | <boolean> | <domain_value> );

<digit> = zero { 0 } | one { 1 } | two { 2 } | three { 3 } | four { 4 } | five { 5 }
                 | six { 6 } | seven { 7 } | eight { 8 } | nine { 9 };

/* note simple "failed" and "succeeded" only for proposed tasks with no declared slots */

<done> = ( done { done } [ <task_name> [ <values> ] ] )
         | failed { done / false } | succeeded { done / true };

<execute> = execute { execute } [ <task_name> [ <values> ] ];

<next> = [what] next { next };
```


command.gram (cont'd)

```
<yes> = ( yes | ok ) { yes };

<no> = no { no };

<status> = status { status };

<clear> = clear { clear };

<quit> = ( quit | exit | goodbye ) { quit };

<verbose> = verbose { verbose } [ <boolean> ];

<debug> = debug { debug } [ <boolean> ];

<boolean> = ( true | on ) { true } | ( false | off ) { false };

/* following are placeholders for task names and additional domain values defined
   in .gram files associated with task models */

<task_name> = <NULL>;

<decomp_name> = <NULL>;

<domain_value> = <NULL>;
```

Library.gram

```
<task_name> = ( borrow [ a book ] ) { Borrow } |
              ( go to the library ) { GoToLibrary } |
              ( choose [ a book ] ) { ChooseBook } |
              ( look [ a book ] up in the catalog ) { LookupInCatalog } |
              ( take [ a book ] ) { TakeFromShelf } |
              ( use search engine ) { UseSearchEngine } |
              ( check out [ a book ] ) { CheckOut } ;

<domain_value> =  ( Stranger [ in a Strange Land ] ) { stranger } |
                  Mindscan { mindscan } |
                  ( [ A ] Fire [ Upon the Deep ] ) { fire } |
                                Heinlein { "Heinlein" } |
                  Sawyer { "Sawyer" } | Vinge { "Vinge" } ;
```


Examples from Library2.guide

```
CE> say let's borrow Mindscan
    # task Borrow / mindscan

CE> say please use search engine on Vinge
    # task UseSearchEngine / "Vinge"
```

Limitations of Current Scheme

- Simple tags interpretation restricts order of output tags to correspond to input word order
- *Many* other ways of using tags
  - e.g., tags can be JavaScript
  - sphinx4-1.0beta/demo/jsapi/tags/FeatureValueDemo.gram

JavaScript Tags

```
public <order> = [I (want | would like) a]
                        ( <pizza>  { this.command = "buyPizza"; }
                    | <burger> { this.command = "buyBurger"; }
                    ) { this.item = $; };

<pizzaTopping> = cheese | pepperoni | mushrooms | mushroom | onions | onion | sausage;

<pizza> = <NULL> { this.toppings = new Array(); }
              ([and] <pizzaTopping> {
              this.toppings = this.toppings.concat($.$value);
          })*
          (pizza | pie) { this.itemType = "pizza"; }
              [with] ([and] <pizzaTopping> {
              this.toppings = this.toppings.concat($.$value);
          })*;

<burgerTopping> = onions | pickles | tomatoes | lettuce | cheese;

<burgerCondiment> = mayo | relish | ketchup | mustard | special sauce;

<burger> = <NULL> { this.toppings = new Array(); }
           ((burger | hamburger) { this.itemType = "burger"; }
            | cheeseburger { this.itemType = "burger";
                             this.toppings = this.toppings.concat("cheese"); }
           )
               [with]
               ([and] (<burgerTopping> | <burgerCondiment>) {
                      this.toppings = this.toppings.concat($.$value); }
           )*;
```